DEC GKS User Manual

Order Number: AA–HW45C–TE

April 1989

This document is an instructional manual that is supplementary to the *DEC GKS Reference Manual*, and contains information for both the novice and the moderately experienced GKS programmer. Before reading further, software users may wish to review the release notes by typing HELP GKS RELEASE_NOTES on the DCL command line.

Revision/Update Information:	This revised document supersedes the *VAX GKS User Manual* (Order No. AI–HW45B–TE).
Operating System and Version:	VMS Version 4.7 or higher. ULTRIX Version 3.0 or higher. VAXstation requirement: VAXstation Windowing Software Versions 3.3 or higher, or DECwindows Version 1.0.
Software Version:	DEC GKS Version 4.0

digital equipment corporation
maynard, massachusetts

First Printing May 1984
Revised March 1986, May 1987, April 1989

ZK4629

Contents

Preface

Manual Objectives

This document is instructional, is supplementary to the *DEC GKS Reference Manual*, and contains information for both the novice and the moderately experienced DEC GKS programmer. Since the focus of this book is programming technique as opposed to complete product description, you may wish to review the introductory sections of each of the chapters in the *DEC GKS Reference Manual* as you read this book.

NOTE

Before reading this manual, you should review the DEC GKS release notes by typing the following:

```
$ HELP  GKS  RELEASE_NOTES
```

Intended Audience

This manual is intended for experienced application programmers who need information supplementary to the *DEC GKS Reference Manual*. Readers should be familiar with one high-level language and the DIGITAL Command Language (DCL). (For more information concerning DCL, refer to the *VAX/VMS DCL Dictionary*.)

Document Structure

This manual contains the following components:

- Chapter 1, Introducing DEC GKS, provides a brief introduction to the GKS standard and to DEC GKS.
- Chapter 2, Programming With DEC GKS, introduces basic DEC GKS programming techniques.
- Chapter 3, Writing Device-Independent Programs, introduces the method of using inquiry functions to write device-independent programs.
- Chapter 4, Composing and Transforming Pictures, provides information concerning the DEC GKS coordinate systems, picture composition, and zooming in and out of a picture.
- Chapter 5, Generating Output, provides information concerning DEC GKS output primitives, individual and bundled attributes, aspect source flags, segment formation, segment transformation, segment clipping, segment attributes, surface regeneration, and output deferral.
- Chapter 6, Requesting Input, provides information concerning the logical input device classes, normalization viewport priority, input data records, and synchronous input.
- Chapter 7, Sampling Input and Generating Events, provides information concerning input process documentation, simultaneously active input devices, and asynchronous input.
- Appendix A, DEC GKS Glossary, provides definitions for DEC GKS terminology.
- Appendix B, Sample Programs, provides a listing of the sample DEC GKS program Starry Night (described in Chapter 3, Writing Device-Independent Programs), coded in all supported languages.

Associated Documents

You may find the following documents useful when using DEC GKS:

- *DEC GKS User Manual*—For programmers who need tutorial information or guides to programming technique.
- *DEC GKS FORTRAN Binding Reference Manual*—For programmers who need specific syntax and argument descriptions for the FORTRAN Binding.
- *DEC GKS GKS$ Binding Reference Manual*—For programmers who need specific syntax and argument descriptions for the GKS$ Binding.

- *DEC GKS C Binding Reference Manual*—For programmers who need specific syntax and argument descriptions for the C Binding.
- *DEC GKS Device Specific Reference Manual*—For programmers who need information about specific devices.
- *Building a DEC GKS Workstation Handler System*—For programmers who need to build DEC GKS workstation graphics handler.
- *Building a DEC GKS Device Handler System*—For programmers who need to provide support for a device unsupported by the DEC GKS graphics handlers.
- *DEC GKS Installation Guide*—For system managers who install the DEC GKS VMS software, including the Run-Time installation, on VMS and ULTRIX operating systems.

Conventions

Convention	Meaning
RETURN	The symbol RETURN represents a single stroke of the RETURN key on a terminal.
$ RUN GKSPROG RETURN	In interactive examples, the user's response to a prompt is printed in red; system prompts are printed in black.
INTEGER X . . .	A vertical ellipsis indicates that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.
option, . . .	A horizontal ellipsis indicates that additional arguments, options, or values can be entered. A comma that precedes the ellipsis indicates that successive items must be separated by commas.
READ(5,*)	The chapters in this manual add calls to subroutines contained in Example 3–2. This example serves as a base for all subsequent examples in the book. Code marked in blue is new code that you must add to Example 3–2 so you can execute the new subroutines for a given chapter.

Convention	Meaning
[output-source, . . .]	Square brackets, in function synopses and a few other contexts, indicate that a syntactic element is optional.
deferral mode	All names of the DEC GKS description table and state list entries, and of the workstation description table and state list entries, are italicized.

Chapter 1

Introducing DEC GKS

DEC GKS is a run-time library of graphical functions that are defined by the ANSI X3.124-1985 and ISO 7942-1985 Graphical Kernel System (GKS) standards. When this manual refers to the "GKS standard," the reference applies to either the ANSI or ISO standard.

This chapter provides a general overview of DEC GKS concepts. The remaining chapters show you how to apply the DEC GKS concepts presented in this chapter to actual application programs. The remaining chapters in this manual cover the following topics:

- Basic programming techniques
- Device-independent programming
- Transformations and picture composition
- Generating output
- Accepting input

This manual defines DEC GKS terminology as needed to describe examples and programming technique. (Key concepts are italicized and defined in Appendix A, DEC GKS Glossary.) Some descriptions contain only enough information to understand the topic of discussion. After reading a chapter in this manual, you may wish to review the corresponding chapter in the *DEC GKS Reference Manual*. The *DEC GKS Reference Manual* contains the complete DEC GKS product description.

1.1 DEC GKS

The DEC Graphical Kernel System (GKS) is a set of run-time functions that provides application programs with a standard method of producing graphics on a potentially large number of physical devices (such as workstations, terminal screens, pen plotters, or graphics printers). By using DEC GKS, you do not need to be concerned with the system-specific or device-specific requirements for producing graphical images. You can spend more time developing your particular application.

DEC GKS performs device-independent tasks in a body of code called the DEC GKS *kernel*. To produce graphical images on a physical device, DEC GKS uses bodies of code called *workstation handlers*. A workstation handler can manipulate one or more physical devices. DEC GKS calls a workstation handler using an integer value known as a *workstation identifier*.

For example, the workstation handler identified by the number 41 works with the VAXstation I, the VAXstation II, and the VAXstation II/GPX physical devices. These physical devices are DEC GKS *workstations* with both input and output capabilities. Section 1.1.4 describes the categories of DEC GKS workstations in further detail.

To produce images on a series of physical devices that use a particular graphics language (such as the PostScript™ graphics language), DEC GKS uses bodies of code called *graphics handlers*.

This manual refers to all types of handlers as graphics handlers.

1.1.1 The GKS Standard

The GKS standard specifies both a functional standard and a syntactical standard for GKS routines. A functional standard specifies the task that a function must perform, but does not impose a standard function name or syntax. A syntactical standard specifies the function name, syntax, and task.

DEC GKS implements the functional standard as a group of run-time functions whose identifiers begin with the prefix GKS$. These functions perform tasks as required by the functional GKS standard. You should use the GKS$ functions if you plan to run your applications only on machines that have the DEC architecture.

™ PostScript is a trademark of Adobe Systems, Inc.

DEC GKS also implements the syntactical standard (the FORTRAN and C bindings) as a group of functions for use only in FORTRAN and C application programs. These bindings offer a complete set of GKS functions with standard identifier and parameter names for each of the binding functions. By using the FORTRAN and C binding functions, you can transport your programs from one GKS implementation to another. The FORTRAN binding functions all begin with the letter G. The C binding functions all begin with the letter g. For complete information concerning the FORTRAN binding, refer to *DEC GKS FORTRAN Binding Reference Manual*. For complete information concerning the C binding, refer to *DEC GKS C Binding Reference Manual*.

If you are programming using a DEC GKS supported language other than FORTRAN or C, you must use the GKS$ functions. In the near future, there will be GKS standard language bindings approved for other languages.

1.1.2 DEC GKS Programming

The DEC GKS kernel and graphics handlers perform tasks according to values in DEC GKS internal data structures. The values in these data structures determine which DEC GKS functions you can call at a given point in your application. To be able to perform input and output, almost all DEC GKS programs need to call a small set of DEC GKS control functions.

The following FORTRAN example uses the GKS$ functions to illustrate control function calls used by most DEC GKS applications:

```
        INTEGER WS_ID, SEG_NAME
        DATA WS_ID / 1 /, SEG_NAME / 1 /

❶       CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
❷       CALL GKS$OPEN_WS( WS_ID, 'TTAO:', 13 )
❸       CALL GKS$ACTIVATE_WS( WS_ID )

  C     Create a segment.
❹       CALL GKS$CREATE_SEG( SEG_NAME )
❺       CALL GKS$TEXT( 0.1, 0.5, 'DEC GKS looks mah-velous!' )
        CALL GKS$CLOSE_SEG()

  C     Release the DEC GKS and workstation environments.
❻       CALL GKS$DEACTIVATE_WS( WS_ID )
        CALL GKS$CLOSE_WS( WS_ID )
        CALL GKS$CLOSE_GKS()
```

The following numbers correspond to the numbers in the previous example:

❶ The call to the control function GKS$OPEN_GKS establishes certain DEC GKS data structures necessary for all DEC GKS programming. The argument to this function (in this example, the logical name SYS$ERROR), specifies a file to which DEC GKS writes generated error messages.

You cannot perform many tasks unless you make calls to more DEC GKS control functions.

❷ The call to the control function GKS$OPEN_WS establishes certain DEC GKS data structures necessary for a program that requires user input. The first argument to GKS$OPEN_WS is a workstation identifier. You use the workstation identifier whenever you need to refer to that particular device. The second argument specifies the connection identifier used to connect the device to the system. The third argument is the workstation type identifier. DEC GKS predefines the number 13 to specify the graphics handler that works with the VT241 color terminal. (For a list of the DEC GKS devices and their corresponding workstation type values, refer to Appendix A, DEC GKS Supported Workstations, in the *DEC GKS Reference Manual.*)

You must call GKS$OPEN_GKS before you call GKS$OPEN_WS. Depending on the needs of your application, you can open more than one workstation at a time. After the call to GKS$OPEN_WS, you can request input from the device, but you cannot generate output.

❸ The call to GKS$ACTIVATE_WS alters values in the DEC GKS data structures necessary for a program that requires output generation. The argument passed to this function is the workstation identifier of an open workstation.

Once you call GKS$ACTIVATE_WS, DEC GKS produces any generated output on all workstations that are active at the time of output generation.

❹ The call to GKS$CREATE_SEG establishes DEC GKS data structures necessary for the creation of a *segment*. A segment is a group of output images that can be stored and manipulated as a group. Using DEC GKS output functions, an output *primitive* is an image produced by a single call to an output function.

Placing output primitives in a segment allows greater flexibility in the presentation of the image. For instance, you have the ability to scale and rotate segments, whereas you cannot scale or rotate primitives that are not stored in segments.

After the call to GKS$OPEN_SEG and before the call to GKS$CLOSE_SEG, DEC GKS stores any generated output. Once you close a segment, you cannot add or delete primitives from the segment. You must call GKS$ACTIVATE_WS for at least one workstation before you can create a segment.

The argument passed to GKS$CREATE_SEG is an integer value called the *segment name*. DEC GKS uses this segment name to identify a particular segment.

5 The call to the output function GKS$TEXT produces a text string on the surface of the workstation. The first two arguments are coordinate points used as the starting point for the text string. By default, DEC GKS accepts coordinate points in the square range with a lower left corner of (0.0, 0.0), and extending from 0.0 to 1.0 on both the X and Y axes. You plot your primitive within the default range, and DEC GKS *transforms* the square range to the largest square that your active workstation can produce, with the lower left corner of the range corresponding to the lower left corner of the workstation.

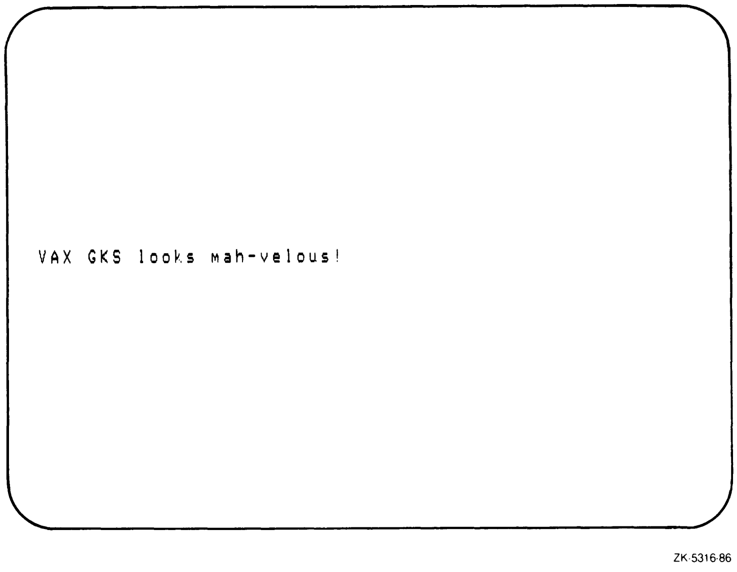
Figure 1-1 illustrates the screen of the VT241 after executing this program. If you are using any other type of terminal, substitute the integer value 0 as the second argument to GKS$OPEN_WS, and by default, DEC GKS uses your terminal connection. You also need to replace the workstation identifier 13 with the identifier appropriate for your workstation type.

6 The last three function calls release the DEC GKS and graphics handler data structures. You must deactivate a workstation before you close it. You must close all open workstations before you close DEC GKS.

NOTE

From this point on, this manual describes rectangular coordinate ranges as follows: ([0,1] x [1,10]). In this example, the coordinate range specifies a rectangular region whose X borders extend from 0.0 to 1.0, and whose Y borders extend from 1.0 to 10.0. The lower left corner of the rectangle is the point (0.0, 1.0) and the upper right point is (1.0, 10.0). For a pictorial explanation of this range notation, refer to Chapter 1, Introduction to DEC GKS, in the *DEC GKS Reference Manual.*

Figure 1–1: Generating a Text Output Primitive—VT241



NOTE

All figures in this manual depicting a workstation surface may appear slightly different than what you see on the surface of the actual workstations. The drawings provide you with the following information concerning the output primitives in a generated picture: relative position on the display surface, the picture's shape, approximate color representations, and approximate patterns.

1.1.3 DEC GKS Function Categories

The DEC GKS function categories are as follows:

- Control
- Output
- Output attribute
- Transformation
- Input
- Segment
- Metafile

- Error-handling
- Inquiry

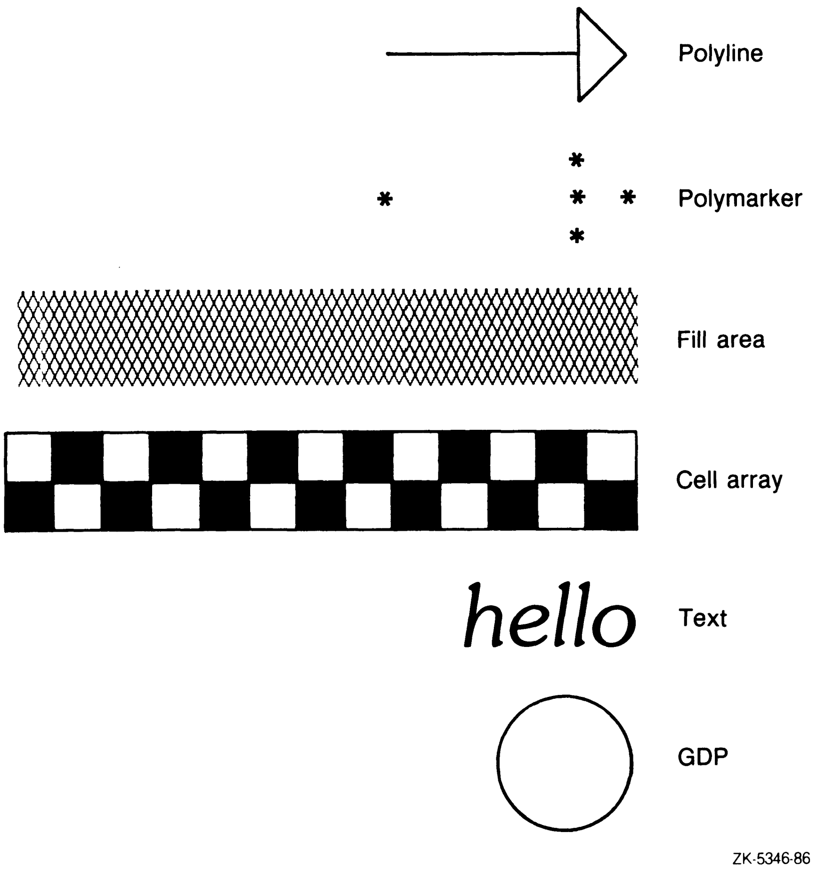
The control functions determine which DEC GKS functions you can call at a given point in your program. They also control the buffering of output and the regeneration of segments on the workstation surface.

The output attributes produce primitives of the following types:

- Polylines—Lines.
- Polymarkers—Symbols.
- Fill areas—Filled polygons.
- Text—Character strings.
- Cell Array—Filled cells of a rectangle.
- Generalized Drawing Primitives—A workstation-dependent image such as a circle.

Figure 1–2 illustrates possible output from each of the types of output primitives.

Figure 1–2: Possible DEC GKS Primitives—VT241



Output attributes affect the appearance of a primitive. For instance, by changing the line type attribute, you can produce solid, dashed, dotted, or dashed-dotted lines.

Transformations affect the composition of the graphical picture and the presentation of that picture. There are *normalization* and *workstation* transformations. The normalization transformations allow you to use various coordinate ranges for different primitives within a single picture. In this way, you can use a coordinate range that suits each particular primitive in a large picture.

The workstation transformations control the portion of the picture that you see on the workstation's surface, and the portion of the surface used to display the picture. Using workstation transformations, you can pan across a picture, zoom in to a picture, or zoom out of a picture.

The input functions allow an application to accept input from a user.

The segment functions store and manipulate segments.

The metafile functions allow you to store and to recall an audit of calls to DEC GKS functions. Using metafiles, you can store a DEC GKS session so that another application can interpret that session, thus reproducing the picture created by the original application. This manual does not discuss metafiles in detail. For more information concerning metafiles, refer to the *DEC GKS Reference Manual.*

The error-handling functions allow you to invoke a user-written error handler when a call to another DEC GKS function generates an error. This manual does not discuss error-handling in detail. For more information concerning error-handling, refer to the *DEC GKS Reference Manual.*

The inquiry functions return valuable default and current information about DEC GKS or about the device with which you are working. Chapter 3, Writing Device-Independent Programs, describes the DEC GKS inquiry functions and how you use them to write device-independent programs.

1.1.4 DEC GKS Workstation Categorization

The various capabilities of each physical device determine the *workstation category.* Most workstations fall into the following categories:

Category	Description
GKS$K_WSCAT_OUTPUT	A workstation of the category GKS$K_WSCAT_OUTPUT can only display graphical images on a single display surface. An example of a device placed in this workstation category is a printer, such as the LA210.
GKS$K_WSCAT_INPUT	A workstation of the category GKS$K_WSCAT_INPUT can only accept input by means of a mouse, a tablet, a keyboard, and so forth. None of the DEC GKS supported devices are GKS$K_WSCAT_INPUT.

Category	Description
GKS$K_WSCAT_OUTIN	A workstation of category GKS$K_WSCAT_OUTIN can display graphical images on the workstation surface as well as accept input. Examples of a device placed in this workstation category are terminals and workstations, such as the VT240 and the VAXstations.

DEC GKS also implements workstations of special categories. The workstation categories GKSK_WSCAT_MO, GKSK_WSCAT_MI, and GKS$K_WSCAT_WISS store metafiles and segments. Chapter 5, Generating Output, describes the category GKS$K_WSCAT_WISS. For more information concerning metafiles, refer to the *DEC GKS Reference Manual.*

In this manual, the term *workstation surface* applies to the portion of the workstation capable of displaying output. Using a VT241, the surface is the terminal screen. Using an LA210 printer, the surface is a single sheet of paper.

1.1.5 GKS Levels

The GKS standard defines 12 levels of GKS implementation defined by input and output capability. The input and output levels are mutually independent. The output levels are subsets of the next highest levels; likewise, the input levels are subsets of the next highest input levels.

Output levels are indicated in order of increasing capability by the characters m, 0, 1, and 2. The input levels are indicated in order of increasing capability by the characters a, b, and c.

The DEC GKS software is a level 2c implementation, incorporating all of the GKS output capabilities (level 2) and all input capabilities (level c). From this point on, this manual uses the term "DEC GKS" when describing the 2c level DEC GKS product.

Figure 1–3 defines the 12 upwardly compatible levels of GKS, and outlines the functionality offered by DEC GKS. DEC GKS implements the highest level of GKS (level 2c).

Figure 1–3: Functionality by GKS Levels

	Input Levels		
Output Levels	a	b	c
m	No input, minimal control, individual attributes, one settable normalization transformation, subset of output and attribute functions.	Request input, set operating mode and initialize functions for input devices, no pick input.	Sample and event input no pick.
0	Basic control, bundled attributes, multiple normalization transformations, all output and attribute functions, optional metafiles.	Set viewport input priority.	All of level mc, above.
1	Full output including settable bundles, multiple workstations, basic segmentation, no workstation independent segment storage, metafiles.	Request pick, set operating mode and initialize functions for pick input.	Sample and event input for pick.
2	Workstation independent segment storage	All of level 1b, above.	All of level 1c, above.

ZK-5027-86

The GKS input levels are determined by three types of input *operating modes.* The input operating modes are called request, sample, and event mode. In request mode, the application program waits for the user to enter input. Once the user signals the end of input, the application resumes. In sample and event mode, the application program and the input process operate asynchronously, so that the user enters input as the application program continues to execute.

1.1.6 DEC GKS Function Calls

To call either a GKS$, FORTRAN, or C binding function, use the calling sequence required by the language you use to write the application program. For instance, you precede the function identifier with the FORTRAN CALL statement when using VAX FORTRAN with the GKS$ functions. For an example, see the following code:

```
CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
```

As a second option, you can check the status of the returned condition value, as follows:

```
STATUS = GKS$OPEN_GKS( 'SYS$ERROR:' )
```

If you use the FORTRAN binding, code a function call as follows:

```
CALL GOPKS( 'SYS$ERROR:' )
```

Or, as follows:

```
STATUS = GOPKS( 'SYS$ERROR:' )
```

DEC GKS also provides language-specific definition files that you can include in your application programs. In most applications, you will need to include this file in order to take full advantage of the DEC GKS function calls. For instance, the definition files enable you to use the DEC GKS constants in conjunction with GKS function calls.

The following FORTRAN code example illustrates the use of the FORTRAN definition file:

```
C     Include the definition file...
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'

      CALL  GKS$CLEAR_WS( WS_ID, GKS$K_CLEAR_ALWAYS )
```

The definition file GKSDEFS.FOR contains the definition of the constant GKS$K_CLEAR_ALWAYS. Many of the DEC GKS constants begin with the prefix GKS$K. For more information concerning the DEC GKS constants and definition files, refer to Chapter 1, Introduction to DEC GKS, in the *DEC GKS Reference Manual.* For information concerning constants that are defined as error condition codes, refer to Appendix D, DEC GKS Error Messages, in the *DEC GKS Reference Manual.*

After you write and edit your FORTRAN DEC GKS application, you compile, link, and run your program, as follows:

```
$ FORTRAN  GKS_PROGRAM
$ LINK     GKS_PROGRAM
$ RUN      GKS_PROGRAM
```

For information unique to your language's program development, refer to the appropriate language documentation.

If you are programming using the FORTRAN language binding, the process is the same except for the linking of your object module. You need to link your FORTRAN binding program with the appropriate binding object library. There are several ways to do this. The following example presents one method:

```
$ DEFINE  GKSFORBND        SYS$LIBRARY:GKSFORBND
$ LINK    BND_PROGRAM.OBJ, GKSFORBND/LIBRARY
```

1.2 Program Examples Used in This Manual

With the exception of the program example in Chapter 2, Programming With DEC GKS, all program examples in this manual are based on the Starry Night program in Example 3–2. Additional examples slightly alter subroutines in the Starry Night program, or they call additional subroutines. You may want to key in this program so that you can follow program execution as you read the manual. If you are not programming in VAX FORTRAN, this manual presents the Starry Night program written in each of the DEC GKS supported languages, in Appendix B, Sample Programs.

All program examples in this manual are written in VAX FORTRAN for consistency. Where confusion may occur, VAX FORTRAN specific constructs are flagged. However, if you are unfamiliar with FORTRAN, you may wish to review the following list of FORTRAN-specific constructs used in the program examples in this manual.

Construct	Description
IMPLICIT NONE	This statement prevents the VAX FORTRAN compiler from implicitly declaring variable names that you have not declared.
C	This character, located in the first column of the line, signifies that the entire line contains a comment.

Construct	Description
*	This character, located in column six, is a continuation character. This character signifies that the previous line of code continues onto the line marked with the asterisk (*).
DATA	The DATA statement initializes program variables with data.
CHARACTER*80	This identifier is used to declare a character string of length 80.
INTEGER VAR(3)	This declaration declares a three-element array of type INTEGER.
%DESCR %VAL	These constructs are argument listpass arguments by descriptor, by value, and by reference.
LEN	This construct is a built-in function that returns the length of a string.
%LOC(array)	This built-in function returns the address of its argument.

Chapter 2

Programming With DEC GKS

This chapter provides an introduction to the following DEC GKS programming concepts:

- Control functions
- Picture coordinate points
- Output attributes
- Segments
- Surface regeneration

2.1 Using DEC GKS Control Functions

The first decision that you must make concerns physical devices. Once you choose a physical device of category GKS$K_WSCAT_OUTPUT or GKS$K_WSCAT_OUTIN, you need to locate the appropriate workstation type identifier. Appendix A, DEC GKS Supported Workstations, in the *DEC GKS Reference Manual,* lists the DEC GKS constants used to designate workstation types. In this chapter, to demonstrate the creation of a DEC GKS application program, the assumption is made that you are working at a VT241 color terminal.

Consequently, the shell of a DEC GKS program appears as follows:

```
❶       INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        IMPLICIT NONE
        INTEGER WS_ID
        DATA WS_ID / 1 /

❷       CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
        CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
        CALL GKS$ACTIVATE_WS( WS_ID )
            .
            .
            .
  C     The body of the program goes here...
            .
            .
            .
❸       CALL GKS$DEACTIVATE_WS( WS_ID )
        CALL GKS$CLOSE_WS( WS_ID )
        CALL GKS$CLOSE_GKS()
        END
```

The following numbers correspond to the numbers in the previous example:

❶ This code includes the DEC GKS FORTRAN definition file. By including the FORTRAN definition file, you can use the DEC GKS constants as arguments to the DEC GKS function calls. The use of DEC GKS constants allows quicker coding, better program documentation, and easier debugging.

This code also defines the variable WS_ID as containing the value 1. This variable is used as a workstation identifier.

❷ This code first opens the DEC GKS data structures and specifies the translation of the logical name SYS$ERROR as the destination of all error messages. When using a VT241, the default translation of SYS$ERROR is your terminal screen. For efficient debugging, you may wish to pass a VMS file specification to GKS$OPEN_GKS, so that you can store the error messages. If you choose this option, generated error messages no longer appear on your workstation surface.

This code also opens a workstation by assigning WS_ID (the value 1) as the workstation identifier, GKS$K_CONID_DEFAULT as the device connection (the physical connection identification between the device and the host computer), and GKS$K_VT240 as the workstation type (a color VT241 terminal). When programming with DEC GKS, you often want to use the surface of your terminal to create graphical images. By specifying GKS$K_CONID_DEFAULT, DEC GKS uses the translation of the logical name TT (the default device connection to your VT241 terminal) to establish the connection to your device. Chapter 3, Writing Device-Independent Programs, shows you how to use this constant to specify various device connections at the DIGITAL Command Language (DCL) level.

NOTE

> If you are using a VAXstation, DEC GKS does *not* use the connection identifier argument to GKS$OPEN_WS as a device connection. DEC GKS uses this string as a label that is placed at the top of the auxiliary window created for DEC GKS output. Notice that if you specify GKS$K_CONID_DEFAULT to GKS$OPEN_WS using a VAXstation, the logical name TT appears at the top of the newly created DEC GKS window.

After activating the open workstation, you can generate output to the workstation surface.

❸ This code releases the DEC GKS and the workstation environments. You must call these functions at the end of your DEC GKS programs in order to assure an orderly exit from your program.

If you are not using a VT241, many of the program examples written in this chapter will not execute properly. Before you can do further testing, you must look up the DEC GKS constant corresponding to your workstation in Appendix A, DEC GKS Supported Workstations, in the *DEC GKS Reference Manual*. Whenever a program in this chapter contains a call to GKS$OPEN_WS and passes the constant GKS$K_VT240, substitute the constant appropriate for your workstation. The only restriction is that your workstation must be of the DEC GKS category GKS$K_WSCAT_OUTPUT or GKS$K_WSCAT_OUTIN.

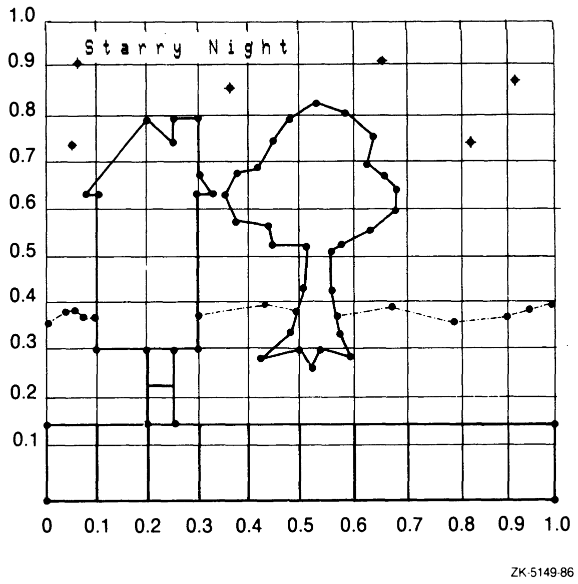
2.2 Plotting a Picture

Once you have called the DEC GKS control functions that establish the DEC GKS and workstation environments, you may want to produce a picture on the surface of the workstation. To do this, you need to plot your picture within the square coordinate range ([0,1] x [0,1]). This coordinate range is the default portion of the imaginary *world coordinate* range.

Once you establish your world coordinate points, you pass them to the desired output function. DEC GKS *transforms* the plotted picture through the DEC GKS coordinate systems and draws the picture on the largest square that your workstation can produce, with the lower left corner of the default world coordinate square corresponding with the lower left corner of the surface of your workstation. (Chapter 4, Composing and Transforming Pictures, describes the DEC GKS coordinate systems in detail.)

Figure 2–1 illustrates a picture plotted in the default range.

Figure 2–1: Plotting a Picture in Default World Coordinate Space



2.3 Generating Output

Once you plot your picture on the default portion of the world coordinate space, you pass the coordinate values to the appropriate DEC GKS output function. The DEC GKS output functions generate the basic components, or *primitives*, of all pictures.

The DEC GKS output functions are as follows:

- GKS$POLYLINE—Draws connected lines between requested points.
- GKS$POLYMARKER—Marks one or more locations with symbols.
- GKS$TEXT—Draws character strings.
- GKS$FILL_AREA—Fills a polygon.

- GKS$CELL_ARRAY—"Colors" cells of a specified rectangle.
- GKS$GDP—Draws a device-dependent primitive called a generalized drawing primitive (GDP).

To generate the plotted picture, you can use GKS$TEXT to generate the title, GKS$POLYMARKER to generate the stars, GKS$POLYLINE to generate the horizon, GKS$FILL_AREA to generate the house and tree, and GKS$CELL_ARRAY to generate the sidewalk and road.

When you generate DEC GKS primitives using the output functions, there are default attributes that affect the way in which the primitives appear on the workstation surface. For instance, if you call GKS$POLYLINE to output a line, the line is drawn solid (instead of dashed or dotted), in the *foreground* color (instead of other shades or colors that the workstation can produce), and at the smallest width that the device handler supports (instead of a wider width).

The foreground color is the color that your workstation uses, by default, to present text on the screen. The workstation uses the *background* color to fill the surface "behind" the text. On a VT241, the default background color is black and the default foreground color is green. Section 2.4 discusses color and other output attributes in detail.

The following sections add code to the control function example until it produces the desired picture. The blue lines in the code examples are lines you must add to the first code example in this chapter to generate the plotted picture. Example 2-1 is the complete program example that generates the initial Starry Night picture.

2.3.1 Generating Text

As the first task, you can add the text at the top of the picture in Figure 2-1. The following code example illustrates text generation:

```
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID
❶       REAL TEXT_START_X, TEXT_START_Y
        DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
     *  TEXT_START_Y / 0.9 /

        CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
        CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
        CALL GKS$ACTIVATE_WS( WS_ID )
```

```
❷       CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
       * 'Starry Night' )
            .
            .
            .
  C     The remaining body of the program goes here...
            .
            .
            .
        CALL GKS$DEACTIVATE_WS( WS_ID )
        CALL GKS$CLOSE_WS( WS_ID )
        CALL GKS$CLOSE_GKS()
        END
```

The following numbers correspond to the numbers in the previous example:

❶ This code defines two variables that specify the X and Y world coordinate values of the text starting point. By default, the base position of the first letter of the text string appears at (0.05, 0.9). This coordinate point is in the upper left corner of the default, square section of the world coordinate space. (Chapter 5, Generating Output, discusses the text attributes in further detail.)

❷ This code generates the text string at the specified world coordinates.

2.3.2 Generating Markers

Now, you can add the code that generates the stars in Figure 2–1, as follows:

```
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
❶       INTEGER WS_ID, NUM_STARS
        REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
       * STARS_Y_VALUES( 6 )
        DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
       * TEXT_START_Y / 0.9 /, NUM_STARS / 6 /
        DATA STARS_X_VALUES / 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 /
        DATA STARS_Y_VALUES / 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 /

        CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
        CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
        CALL GKS$ACTIVATE_WS( WS_ID )

        CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
       * 'Starry Night' )
```

```
❷       CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
      * STARS_Y_VALUES )
            .
            .
            .
  C     The remaining body of the program goes here...
            .
            .
            .
        CALL GKS$DEACTIVATE_WS( WS_ID )
        CALL GKS$CLOSE_WS( WS_ID )
        CALL GKS$CLOSE_GKS()
        END
```

The following numbers correspond to the numbers in the previous example:

❶ This body of code defines the variables that you need to pass as arguments to GKS$POLYMARKER. The variable NUM_STARS specifies the number of markers to generate.

Notice the two arrays STARS_X_VALUES and STARS_Y_VALUES. The array STARS_X_VALUES contains the X world coordinate values of all six markers to be generated, and STARS_Y_VALUES contains all six Y world coordinate values. So, the first marker location is (0.05, 0.7), the second location is (0.06, 0.86), and so forth. All output functions to which you pass a list of world coordinate values require arrays of this structure.

❷ This code generates the markers. By default, DEC GKS generates the smallest asterisk that the VT241 device handler supports.

2.3.3 Generating Polylines and Fill Areas

Once you know how to construct arrays containing the X and Y values of a list of world coordinate values, you can call GKS$POLYLINE and GKS$FILL_AREA to construct the horizon line, the house, and the tree, as follows:

```
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
      * NUM_HOUSE_PTS, NUM_LAND_PTS

        REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
      * STARS_Y_VALUES( 6 ), TREE_X( 29 ), TREE_Y( 29 ),
      * HOUSE_X( 12 ), HOUSE_Y( 12 ), LAND_X( 15 ),
      * LAND_Y( 15 )

        DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
      * TEXT_START_Y / 0.9 /, NUM_STARS / 6 /,
      * NUM_TREE_PTS / 29 /, NUM_HOUSE_PTS / 12 /,
      * NUM_LAND_PTS / 15 /

        DATA STARS_X_VALUES / 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 /
        DATA STARS_Y_VALUES / 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 /
```

```
       DATA TREE_X / 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
      * 0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
      * 0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
      * 0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
      * 0.515, 0.51, 0.495, 0.475, 0.425 /
       DATA TREE_Y / 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
      * 0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
      * 0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
      * 0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
      * 0.5, 0.425, 0.38, 0.33, 0.28 /

       DATA HOUSE_X / 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
      * 0.25, 0.25, 0.2, 0.075, 0.1, 0.1 /
       DATA HOUSE_Y / 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
      * 0.75, 0.7, 0.75, 0.6, 0.6, 0.3 /

       DATA LAND_X / 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
      * 0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 /
       DATA LAND_Y / 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
      * 0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
      * 0.385 /

       CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
       CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
       CALL GKS$ACTIVATE_WS( WS_ID )

       CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
      * 'Starry Night' )
       CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
      * STARS_Y_VALUES )
❶      CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )
       CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
       CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )
           .
           .
           .
 C     The remaining body of the program goes here...
           .
           .
           .
       CALL GKS$DEACTIVATE_WS( WS_ID )
       CALL GKS$CLOSE_WS( WS_ID )
       CALL GKS$CLOSE_GKS()
       END
```

The following number corresponds to the number in the previous example:

❶ These three lines of code generate the house, the tree, and the horizon line. By default, DEC GKS uses a hollow fill area for the tree and the house. When generating hollow fill areas, DEC GKS outlines the polygon by connecting the specified world coordinate points with lines.

2.3.4 Generating Cell Arrays

As the last step in producing the desired picture, you can call GKS$CELL_ARRAY to create the sidewalk and road. Example 2–1 shows the complete set of calls needed to generate the plotted picture in Figure 2–1, on a VT241.

Example 2–1: Generating the Plotted Picture

```
       IMPLICIT NONE
       INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
       INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
     * NUM_HOUSE_PTS, NUM_LAND_PTS, SIDE_OFF_COL,
     * SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
     * SIDE_COLORS( 1, 2 ), ROAD_OFF_COL, ROAD_OFF_ROW,
     * ROAD_NUM_COL, ROAD_NUM_ROW, ROAD_COLORS( 10, 1 )

       REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
     * STARS_Y_VALUES( 6 ), TREE_X( 29 ), TREE_Y( 29 ),
     * HOUSE_X( 12 ), HOUSE_Y( 12 ), LAND_X( 15 ),
     * LAND_Y( 15 ), SIDE_START_X, SIDE_START_Y, SIDE_DIAG_X,
     * SIDE_DIAG_Y, ROAD_START_X, ROAD_START_Y, ROAD_DIAG_X,
     * ROAD_DIAG_Y

       DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
     * TEXT_START_Y / 0.9 /, NUM_STARS / 6 /,
     * NUM_TREE_PTS / 29 /, NUM_HOUSE_PTS / 12 /,
❶    * NUM_LAND_PTS / 15 /, SIDE_START_X / 0.2 /,
     * SIDE_START_Y / 0.3 /, SIDE_DIAG_X / 0.25 /,
     * SIDE_DIAG_Y / 0.15 /, SIDE_OFF_COL / 1 /,
     * SIDE_OFF_ROW / 1 /, SIDE_NUM_COL / 1 /,
     * SIDE_NUM_ROW / 2 /, ROAD_START_X/ 0.0 /,
     * ROAD_START_Y / 0.15 /, ROAD_DIAG_X / 1.0 /,
     * ROAD_DIAG_Y / 0.0 /, ROAD_OFF_COL / 1 /,
     * ROAD_OFF_ROW / 1 /, ROAD_NUM_COL / 10 /,
     * ROAD_NUM_ROW / 1 /

❷      DATA SIDE_COLORS / 3, 2 /
       DATA ROAD_COLORS / 2, 3, 2, 3, 2, 3, 2, 3, 2, 3 /

       DATA STARS_X_VALUES / 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 /
       DATA STARS_Y_VALUES / 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 /
```

(continued on next page)

Example 2–1 (Cont.): Generating the Plotted Picture

```
      DATA TREE_X / 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
     * 0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
     * 0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
     * 0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
     * 0.515, 0.51, 0.495, 0.475, 0.425 /
      DATA TREE_Y / 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
     * 0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
     * 0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
     * 0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
     * 0.5, 0.425, 0.38, 0.33, 0.28 /

      DATA HOUSE_X / 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
     * 0.25, 0.25, 0.2, 0.075, 0.1, 0.1 /
      DATA HOUSE_Y / 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
     * 0.75, 0.7, 0.75, 0.6, 0.6, 0.3 /

      DATA LAND_X / 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
     * 0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 /
      DATA LAND_Y / 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
     * 0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
     * 0.385 /

      CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
      CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
      CALL GKS$ACTIVATE_WS( WS_ID )

      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
     * 'Starry Night' )
      CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
     * STARS_Y_VALUES )
      CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )
      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )

❸     CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
     * SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL, SIDE_OFF_ROW,
     * SIDE_NUM_COL, SIDE_NUM_ROW, %DESCR( SIDE_COLORS ) )

      CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
     * ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL, ROAD_OFF_ROW,
     * ROAD_NUM_COL, ROAD_NUM_ROW, %DESCR( ROAD_COLORS ) )

      CALL GKS$DEACTIVATE_WS( WS_ID )
      CALL GKS$CLOSE_WS( WS_ID )
      CALL GKS$CLOSE_GKS()
      END
```

The following numbers correspond to the numbers in the previous example:

❶ This code defines the variables to be passed as arguments to the GKS$CELL_ARRAY function. GKS$CELL_ARRAY divides a designated rectangular area into smaller rectangles, and displays each rectangle (or *cell*) in a specified color or shade. You must specify the colors in a two-dimensional array of integer values. The integer values are called *color indexes.* Some color indexes are predefined by the device handler to represent given colors (the VT241 defines the integer index value 3 to represent the color blue). Section 2.4 discusses color indexes in greater detail.

The GKS$CELL_ARRAY function requires a starting point and a diagonal point in world coordinate values. Using these points, DEC GKS creates a rectangular region on the world coordinate plane.

GKS$CELL_ARRAY requires an offset column and row number. These numbers determine an offset into the color index array from which to begin reading values. This code specifies that DEC GKS must begin reading index values at element (1, 1) in the color index array.

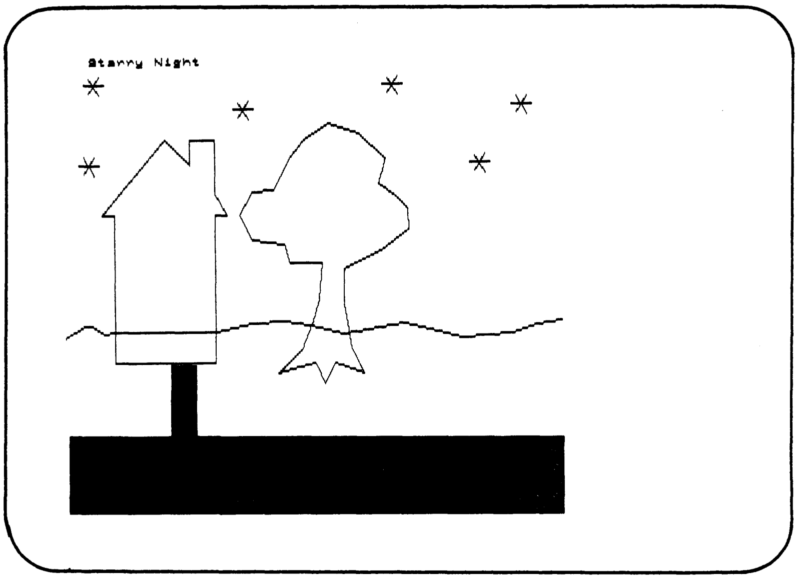
GKS$CELL_ARRAY also requires the number of rows and columns in which DEC GKS is to divide the cell array. DEC GKS divides the rectangle into rows and columns of equal size, and maps the corresponding color indexes from the index array to the cells of the rectangle. You can determine the maximum allowable number of cell array rows and columns by calculating the number of rows and columns from the offset starting element to the last element of the color index array.

❷ This code defines the color index values. By default, the VT241 handler defines the value 2 to represent the color red, and the value 3 to represent blue.

❸ This code generates the cell arrays. The sidewalk contains two alternating red and blue cells, and the road contains ten cells.

Figure 2–2 illustrates the workstation surface after you execute the code example.

Figure 2–2: Generating a Picture on the VT241



ZK-5141-86

2.4 Altering the Appearance of the Primitives

When reviewing the picture generated by the code examples presented in this chapter, think about how the overall presentation can be improved by altering the appearance of the output primitives. For instance, the polyline representing the horizon appears in front of the house and tree, when it should appear to be behind those objects; the house and tree could appear in a more striking manner; and, the default text size is difficult to read.

Consequently, you may wish to alter the appearance of the output primitives without having to change the plotting of the picture, or having to use different types of primitives. To do this, you can set the output primitive's individual output *attributes*. For instance, you distinguish between different polylines drawn in a single picture, you can change the color, thickness, and line type (solid, dashed, dotted, or dashed-dotted).

By default, DEC GKS uses the *individual* output attribute settings when generating primitives. Individual settings affect one single attribute associated with a primitive. The polyline individual attributes are line type, line width scale factor, and color. There is a way to specify attributes for a given primitive in *bundles,* so that you can control all attributes as a group. For instance, a single polyline bundle entry has values for line type, line width scale factor, and color. Bundles are discussed in detail in Chapter 5, Generating Output.

To see how altering default attributes can change the appearance of your picture, review the changes to the code example presented previously. Code marked in blue is the code you must add to each previous example to produce the specified results. (Example 2–2 presents the complete alteration of the Starry Night program.)

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
          .
          .
          .
     * BLUE
      REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
          .
          .
          .
     * LARGER, WIDER
      DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
          .
          .
          .
❶    * LARGER / 0.04 /, BLUE / 3 /,
     * WIDER / 3.0 /

      CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
      CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
      CALL GKS$ACTIVATE_WS( WS_ID )

❷     CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS )
      CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
      CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED )
      CALL GKS$SET_PLINE_LINEWIDTH( WIDER )

      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
     * 'Starry Night' )
      CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
     * STARS_Y_VALUES )
      CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )
      CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )
❸     CALL GKS$SET_FILL_COLOR_INDEX( BLUE )
      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
```

```
 CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
* SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL, SIDE_OFF_ROW,
* SIDE_NUM_COL, SIDE_NUM_ROW, %DESCR( SIDE_COLORS ) )
 CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
* ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL, ROAD_OFF_ROW,
* ROAD_NUM_COL, ROAD_NUM_ROW, %DESCR( ROAD_COLORS ) )

 CALL GKS$DEACTIVATE_WS( WS_ID )
 CALL GKS$CLOSE_WS( WS_ID )
 CALL GKS$CLOSE_GKS()
 END
```

The following numbers correspond to the numbers in the previous example:

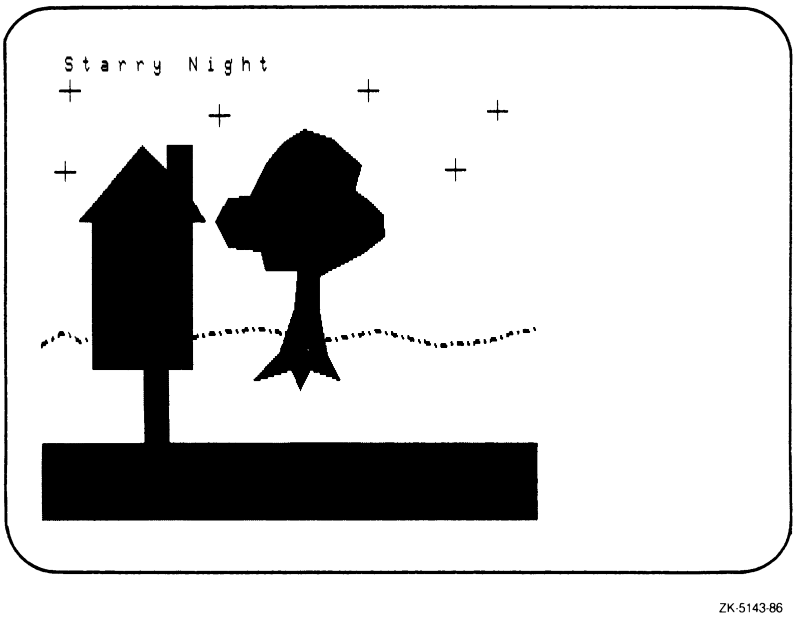
❶ This code defines variables to be passed as arguments to the output attribute functions. Notice that BLUE is defined to be the integer 3. The VT241 graphics handler predefines the index 3 to represent the color blue.

❷ This code sets the text height to the value 0.04 in world coordinate units (notice in the final picture how DEC GKS "adjusts" the character spacing according to the change in character height), changes the markers to plus signs (+), sets a fill area to solid (the default is hollow, which appears as an outline of the fill area), changes the polyline from solid to dashed-dotted, and increases the line width three times. These primitive attribute settings remain in effect for all subsequently generated primitives, unless you change the attribute settings.

❸ This code changes the fill area color index to the value specified by the variable BLUE (the index 3). This causes the house to be a different color than the tree.

Figure 2–3 illustrates the surface of the VT241 workstation after you execute the last code example.

Figure 2–3: Changing the Appearance of the Picture



2.5 Working with Segments

When you create a picture using DEC GKS, you may want to reproduce a graphical image at different positions within a single picture or you may want to treat a graphical image as a single unit. You can treat one or more output primitives as a unit by storing them in a *segment*. When working with segments, both the individual primitives and the segment have attributes.

You can use the segment attributes to highlight the primitives in a defined segment, to control the visibility of a segment, to set the priority of segments (used when two segments overlap on the display surface), to *transform* a segment, and so forth. Chapter 5, Generating Output, explains the segment attributes in detail.

The following code example places the land and the house into a single segment:

```
          IMPLICIT NONE
          INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
          INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
              .
              .
              .
         * LAND_HOUSE

          DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
              .
              .
              .
❶        * LAND_HOUSE / 1 /

          CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
          CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
          CALL GKS$ACTIVATE_WS( WS_ID )

          CALL GKS$SET_TEXT_HEIGHT( LARGER )
          CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS )
          CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
          CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED )
          CALL GKS$SET_PLINE_LINEWIDTH( WIDER )

          CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
         * 'Starry Night' )
          CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
         * STARS_Y_VALUES )
          CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )

          CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
         * SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL, SIDE_OFF_ROW,
         * SIDE_NUM_COL, SIDE_NUM_ROW, %DESCR( SIDE_COLORS ) )
          CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
         * ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL, ROAD_OFF_ROW,
         * ROAD_NUM_COL, ROAD_NUM_ROW, %DESCR( ROAD_COLORS ) )

❷         CALL GKS$CREATE_SEG( LAND_HOUSE )
          CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )
          CALL GKS$SET_FILL_COLOR_INDEX( BLUE )
          CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
          CALL GKS$CLOSE_SEG()

          CALL GKS$DEACTIVATE_WS( WS_ID )
          CALL GKS$CLOSE_WS( WS_ID )
          CALL GKS$CLOSE_GKS()
          END
```

The following numbers correspond to the numbers in the previous example:

❶ This code defines the segment name LAND_HOUSE to be the value 1. This program passes LAND_HOUSE to GKS$CREATE_SEG.

❷ This code defines a segment that contains the primitives representing the land and the house. At this point in the program, you cannot open another segment until you close LAND_HOUSE.

Notice that executing this program does not generate a different picture than the one generated by the previous example. This program simply illustrates how to store a segment.

To add primitives to the segment definition, you call DEC GKS output functions while the segment is open. DEC GKS stores the current output attribute settings with the primitive when you call the output function. If you desire, you can call a DEC GKS function while a segment is open to change a current individual primitive attribute setting. Bear in mind that a change to the current individual attribute setting only affects subsequently generated primitives.

If you use individual attributes settings for primitives stored in a segment (the default situation), then you cannot change the primitive's attributes for the remaining existence of the segment. You need to use bundled attributes if your application requires that you change attributes after segment creation. Chapter 5, Generating Output, explains how to use bundled attributes.

Manipulating segments is one of the most powerful features of a 2c implementation of GKS. Use of segments becomes crucial when controlling changes to the workstation surface. At times, you may need to redraw the picture on the surface of the workstation in order to reflect a change made by the application program. When this redrawing occurs, DEC GKS only redraws primitives contained in segments.

Consequently, you should place *retainable* graphical data in segments. Otherwise, this graphical data is lost when DEC GKS performs certain surface control operations. Section 2.6 discusses control of the workstation surface in greater detail.

2.6 Controlling the Workstation Surface

When you request certain changes to the workstation surface, those changes may occur *dynamically*, or the change may require an *implicit regeneration* of all segments on the workstation surface.

If a change is dynamic, the device handler makes the changes to the surface immediately, without losing output primitives not contained in segments.

If a change requires an implicit regeneration, the device may perform either of two operations as follows:

1. Take action immediately by clearing the surface and only redrawing the currently defined, visible segments.
2. Postpone the requested changes until you *update* the surface of the workstation.

When you update a workstation, you can perform one of two tasks depending on the argument you pass to GKS$UPDATE_WS. If you pass the flag GKS$K_POSTPONE_FLAG, you release any *deferred output* to the surface of the workstation. If you pass the flag GKS$K_PERFORM_FLAG and if the workstation surface is out of date, you force a surface regeneration to occur. (For more information concerning output deferral, refer to Chapter 3, Writing Device-Independent Programs.)

For example, consider the changing of a color *representation*. A color representation is a set of red, green, and blue intensities that a workstation associates with a given color index. By default, a VT241 associates the index 3 with color intensities that yield the primary color blue. If you reassociate the color intensities associated with the index 3, the VT241 makes the color representation change immediately, without requiring an implicit regeneration. (Note that if you are using a workstation other than a VT241, a change to the color representation may cause an immediate or a suppressed implicit regeneration of the surface.)

The following code example illustrates a change to the color representation, which changes the color of the sidewalk, road, and house:

```
       IMPLICIT NONE
       INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
       INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
           .
           .
           .
     * BLUE, LAND_HOUSE, RED

       REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
           .
           .
           .
     * ROAD_DIAG_Y, LARGER, WIDER, RED_INTENS_1, RED_INTENS_2,
     * GREEN_INTENS_1, GREEN_INTENS_2, BLUE_INTENS_1,
     * BLUE_INTENS_2

       DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
           .
           .
           .
❶    * RED_INTENS_1 / 0.56 /, GREEN_INTENS_1 / 0.0 /
     * BLUE_INTENS_1 / 0.0 /, RED_INTENS_2 / 0.8538 /,
     * GREEN_INTENS_2 / 0.6646 /, BLUE_INTENS_2 / 0.2862 /,
     * RED / 2 /, BLUE / 3 /

       CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
       CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
       CALL GKS$ACTIVATE_WS( WS_ID )

       CALL GKS$SET_TEXT_HEIGHT( LARGER )
           .
           .
           .
```

```
      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
     * 'Starry Night' )
        .
        .
        .
      CALL GKS$CREATE_SEG( LAND_HOUSE )
      CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )
      CALL GKS$SET_FILL_COLOR_INDEX( BLUE )
      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()

❷     CALL GKS$SET_COLOR_REP( WS_ID, BLUE, RED_INTENS_1,
     * GREEN_INTENS_1, BLUE_INTENS_1 )

      CALL GKS$SET_COLOR_REP( WS_ID, RED, RED_INTENS_2,
     * GREEN_INTENS_2, BLUE_INTENS_2 )

      CALL GKS$DEACTIVATE_WS( WS_ID )
      CALL GKS$CLOSE_WS( WS_ID )
      CALL GKS$CLOSE_GKS()
      END
```

The following numbers correspond to the numbers in the previous example:

❶ This code redefines the red, green, and blue intensities for the indexes 2 and 3. Consequently, all primitives with the associated color indexes 2 or 3 reflect the immediate color change.

You can use the color chart in Appendix H, DEC GKS Color Chart, of the *DEC GKS Reference Manual*, to match red, green, and blue color intensity combinations with the resulting color. In this example, the new representation of index 2 is associated with the color amber, and the new representation of index 3 is associated with the color brown.

❷ This code generates the immediate change to the color representation for all primitives whose color indexes are 2 or 3. Notice that DEC GKS does not implicitly regenerate the workstation surface; if a regeneration occurred, all primitives not contained in segments would have been cleared from the workstation surface.

As another example, you can change the rectangular area on the workstation surface containing the generated picture. Using a VT241, if you change this rectangle after generating output, the VT241 postpones an implicit regeneration causing the picture on the surface to be out of date, and you must update the surface in order to see the change. (Note that devices other than a VT241 may either implement the change dynamically, may cause an immediate regeneration, or, may postpone a regeneration causing the surface to be out of date.) Example 2–2 illustrates the entire process.

Example 2–2: Controlling the Surface of the VT241

```
       IMPLICIT NONE
       INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
       INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
      * NUM_HOUSE_PTS, NUM_LAND_PTS, SIDE_OFF_COL,
      * SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
      * SIDE_COLORS( 1, 2 ), ROAD_OFF_COL, ROAD_OFF_ROW,
      * ROAD_NUM_COL, ROAD_NUM_ROW, ROAD_COLORS( 10, 1 ),
      * LAND_HOUSE, RED, BLUE

       REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
      * STARS_Y_VALUES( 6 ), TREE_X( 29 ), TREE_Y( 29 ),
      * HOUSE_X( 12 ), HOUSE_Y( 12 ), LAND_X( 15 ),
      * LAND_Y( 15 ), SIDE_START_X, SIDE_START_Y, SIDE_DIAG_X,
      * SIDE_DIAG_Y, ROAD_START_X, ROAD_START_Y, ROAD_DIAG_X,
      * ROAD_DIAG_Y, RED_INTENS_1, RED_INTENS_2,
      * GREEN_INTENS_1, GREEN_INTENS_2, BLUE_INTENS_1,
      * BLUE_INTENS_2, LARGER, WIDER

       DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
      * TEXT_START_Y / 0.9 /, NUM_STARS / 6 /,
      * NUM_TREE_PTS / 29 /, NUM_HOUSE_PTS / 12 /,
      * NUM_LAND_PTS / 15 /, SIDE_START_X / 0.2 /,
      * SIDE_START_Y / 0.3 /, SIDE_DIAG_X / 0.25 /,
      * SIDE_DIAG_Y / 0.15 /, SIDE_OFF_COL / 1 /,
      * SIDE_OFF_ROW / 1 /, SIDE_NUM_COL / 1 /,
      * SIDE_NUM_ROW / 2 /, ROAD_START_X/ 0.0 /,
      * ROAD_START_Y / 0.15 /, ROAD_DIAG_X / 1.0 /,
      * ROAD_DIAG_Y / 0.0 /, ROAD_OFF_COL / 1 /,
      * ROAD_OFF_ROW / 1 /, ROAD_NUM_COL / 10 /,
      * ROAD_NUM_ROW / 1 /, LAND_HOUSE / 1 /,
      * RED_INTENS_1 / 0.56 /, GREEN_INTENS_1 / 0.0 /
      * BLUE_INTENS_1 / 0.0 /, RED_INTENS_2 / 0.8538 /,
      * GREEN_INTENS_2 / 0.6646 /, BLUE_INTENS_2 / 0.2862 /,
      * RED / 2 /, BLUE / 3 /, LARGER / 0.04 /,
      * WIDER / 3.0 /

       DATA SIDE_COLORS / 3, 2 /
       DATA ROAD_COLORS / 2, 3, 2, 3, 2, 3, 2, 3, 2, 3 /

       DATA STARS_X_VALUES / 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 /
       DATA STARS_Y_VALUES / 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 /
```

(continued on next page)

```
      DATA TREE_X / 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
     * 0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
     * 0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
     * 0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
     * 0.515, 0.51, 0.495, 0.475, 0.425 /
      DATA TREE_Y / 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
     * 0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
     * 0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
     * 0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
     * 0.5, 0.425, 0.38, 0.33, 0.28 /

      DATA HOUSE_X / 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
     * 0.25, 0.25, 0.2, 0.075, 0.1, 0.1 /
      DATA HOUSE_Y / 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
     * 0.75, 0.7, 0.75, 0.6, 0.6, 0.3 /


      DATA LAND_X / 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
     * 0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 /
      DATA LAND_Y / 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
     * 0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
     * 0.385 /


      CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
      CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
      CALL GKS$ACTIVATE_WS( WS_ID )


      CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS )
      CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
      CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED )
      CALL GKS$SET_PLINE_LINEWIDTH( WIDER )


      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
     * 'Starry Night' )
      CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
     * STARS_Y_VALUES )
      CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )
      CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
     * SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL, SIDE_OFF_ROW,
     * SIDE_NUM_COL, SIDE_NUM_ROW, %DESCR( SIDE_COLORS ) )
      CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
     * ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL, ROAD_OFF_ROW,
     * ROAD_NUM_COL, ROAD_NUM_ROW, %DESCR( ROAD_COLORS ) )

      CALL GKS$CREATE_SEG( LAND_HOUSE )
      CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )
      CALL GKS$SET_FILL_COLOR_INDEX( BLUE )
      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()
```

(continued on next page)

Example 2–2 (Cont.): Controlling the Surface of the VT241

```
      CALL GKS$SET_COLOR_REP( WS_ID, BLUE, RED_INTENS_1,
     * GREEN_INTENS_1, BLUE_INTENS_1 )
      CALL GKS$SET_COLOR_REP( WS_ID, RED, RED_INTENS_2,
     * GREEN_INTENS_2, BLUE_INTENS_2 )

❶     CALL GKS$SET_WS_VIEWPORT( WS_ID, 0.0, 250.0, 0.0,
     * 250.0 )
❷     CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )

      CALL GKS$DEACTIVATE_WS( WS_ID )
      CALL GKS$CLOSE_WS( WS_ID )
      CALL GKS$CLOSE_GKS()
      END
```

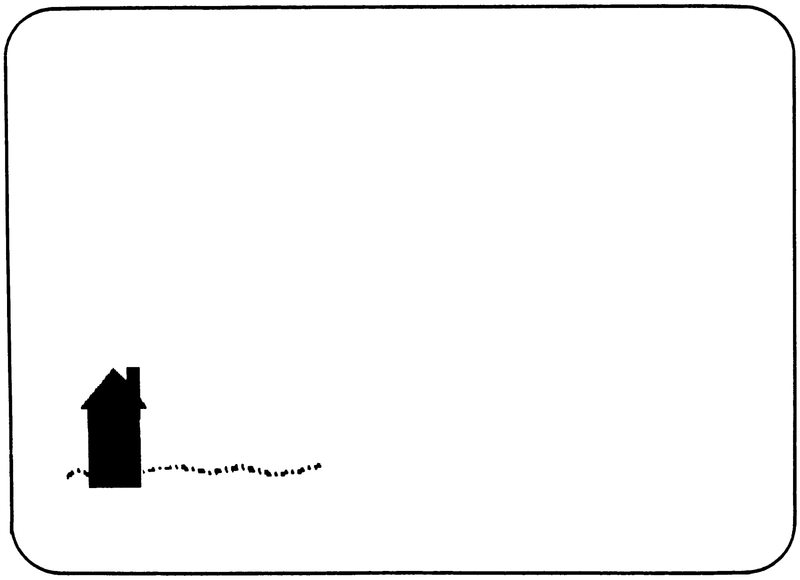
The following numbers correspond to the numbers in the previous example:

❶ The call to GKS$SET_WS_VIEWPORT changes the area of the workstation surface on which DEC GKS generates the picture. This code defines the lower left corner of the VT241 surface as the area on which to generate output. On a VT241, the maximum X value in device coordinates is 479.0, and the maximum Y value is 767.0.

❷ The VT241 handler does not change the area on which to generate output until you force a regeneration of the surface. Once you call GKS$UPDATE_WS and pass GKS$K_PERFORM_FLAG as an argument (*perform* the regeneration), DEC GKS clears the screen, makes the requested change(s), and redraws only the primitives in currently defined, visible segments. In this example, only the house and the horizon line are part of a segment definition; all other output primitives are lost.

Figure 2–4 illustrates the surface of the VT241 workstation after you execute the last code example.

Figure 2–4: Changing the Portion of the Surface Used for the Picture



After seeing the preceding code examples, you may have certain questions. You may want to know the following information:

- Whether your device is deferring output.
- How to obtain the device's maximum coordinate values.
- How you can tell whether your workstation postpones regenerations or performs them as soon as they are needed.
- How you can tell whether your picture is out of date so that you can force a picture regeneration.
- What to do if you are writing a single program that must run on many devices with different capabilities.

To provide information needed to control the workstation surface and to program in a device-independent manner, you can use the DEC GKS inquiry functions. The inquiry functions return information concerning deferred output, device coordinate values, postponed regenerations, and other useful information. Chapter 3, Writing Device-Independent Programs, discusses the inquiry functions and device-independent programming in greater detail.

Chapter 3

Writing Device-Independent Programs

This chapter provides an introduction to writing device-independent programs. The discussion of device independency involves the following topics:

- Device connection and type
- Deferral mode
- Workstation-dependent output attributes
- Color
- Implicit regenerations

The only way that you can access the information stored in the DEC GKS data structures is to use the DEC GKS inquiry functions. These structures contain information that you can use to make valuable programming decisions.

The following table presents an overview of the DEC GKS data structures.

Table/List	**Description**
GKS description table	This table contains constant information about the DEC GKS implementation you are using, such as the level of GKS (with DEC GKS, level 2c), the number of available workstation types, the list of workstation types, the maximum allowable open workstations, and so forth. If you are transporting your programs from one implementation of GKS to another, you may need to inquire about the implementation of GKS on a given system, so that your program does not call unsupported functions.

Table/List	Description
Workstation description table	This type of table contains constant information about one particular workstation, such as the workstation type, the workstation category, the device-specific maximum coordinate values, the different bundled output attribute values, and so forth. Each graphics handler contains a workstation description table describing that particular device. If your DEC GKS application uses more than one workstation at a time, or if you are unsure of the capabilities of your workstation, you may need to inquire about the values contained in the workstation description table.
GKS state list	This list contains entries that specify the current DEC GKS values such as the set of open workstations (if any), the current normalization transformation number, the current character height, and so forth. If you need to check the alterable DEC GKS values, you may need to inquire about the values contained in the DEC GKS state list.
Workstation state list	For each workstation you open, DEC GKS creates a workstation state list. This list contains entries that specify whether output is deferred, whether or not the surface has to be redrawn to fulfill an output request, whether the workstation surface is "empty" by GKS definition, whether the picture on the surface represents all of the requests for output made thus far by the application program, and so forth. If you need information concerning the current state of a particular workstation, you may need to inquire about the values contained in the workstation state list.
Segment state list	DEC GKS maintains a segment state list. The segment state list contains entries that specify the segment name, the set of associated workstations, the detectability of the segment, and so forth. If you need information concerning a particular segment, you may need to inquire about the values contained in the segment state list.

3.1 Writing Device-Independent Code

If you executed the program examples presented in this manual so far, and if you are not using a VT241, you may have experienced some difficulties. For instance, you had to replace the constant GKS$K_VT240 with the constant that corresponded to your workstation. Also, your workstation's graphics handler may or may not have responded to the calls to GKS$SET_COLOR_REP and GKS$SET_WS_VIEWPORT in exactly the same way as the VT240 handler. Perhaps your workstation did not need to perform an implicit regeneration to change the workstation viewport. Perhaps it did not associate the integers 2 and 3 with the colors red and blue by default.

Considering the effort of altering your program every time you run it using a different physical device, DEC GKS provides logical names, constants, and *inquiry* functions so that you can write device-independent programs. The inquiry functions also allow you to ask for values associated with both the default and current state of DEC GKS or of a particular supported device.

In Chapter 2, Programming With DEC GKS, Example 2–2 presented the complete program needed to generate output, to change two color representations, and to change the workstation viewport. To review code that performs the same tasks in a device-independent manner, see Example 3–1 in Section 3.1.6. The following subsections explain each device-independent programming technique individually.

3.1.1 Specifying the Connection and Device Type

When you write a program that is device-independent, your first step is to specify the proper device connection and type identifier. To see how this is accomplished, review the following lines of code from Example 3–1:

```
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
            .
            .
            .
❶       CALL GKS$OPEN_GKS( 'ERROR_FILE.TXT' )
            .
            .
            .
❷       CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT,
      * GKS$K_WSTYPE_DEFAULT )
            .
            .
            .
        CALL GKS$DEACTIVATE_WS( WS_ID )
        CALL GKS$CLOSE_WS( WS_ID )
        CALL GKS$CLOSE_GKS()
        END
```

The following numbers correspond to the numbers in the previous example:

❶ The call to GKS$OPEN_GKS is similar to the calls in previous code examples. However, this call specifies a file in the current directory to which DEC GKS writes any generated error messages. You may wish to do this so that you always have a copy of the errors generated during program execution.

❷ This call to GKS$OPEN_WS contains the argument constants GKS$K_CONID_DEFAULT and GKS$K_WSTYPE_DEFAULT. These constants tell DEC GKS that it must perform a series of logical name translations to obtain the device connection and the device type.

The constant GKS$K_CONID_DEFAULT (or, for most supported languages, the value 0) tells DEC GKS to translate the logical name GKS$CONID in order to determine the name of the device connection. The constant GKS$K_WSTYPE_DEFAULT tells DEC GKS to translate the logical name GKS$WSTYPE to determine the name of the workstation type. Consequently, you can use the DEFINE or ASSIGN command on the DCL command line to define the logical names to be the connection and type with which you are working, as follows:

```
$ FORTRAN PROGRAM RETURN
$ LINK  PROGRAM RETURN
$ DEFINE  GKS$CONID   ttb0 RETURN
$ DEFINE  GKS$WSTYPE  13   ! VT241 Color RETURN
$ RUN   PROGRAM RETURN
   .
   .
   .
$ DEFINE  GKS$CONID   tta3 RETURN
$ DEFINE  GKS$WSTYPE  12   ! VT125 Black and White RETURN
$ RUN   PROGRAM RETURN
   .
   .
   .
```

Before you attempt to define GKS$CONID, you need to perform the following tasks:

1. Make sure that you have allocated the device you need to access. The DCL command SHOW DEVICE provides a list of devices on your system node.
2. Allocate the terminal using the command ALLOCATE (you may need special privileges to allocate the device).
3. Use the command SHOW TERMINAL to make sure that the device's baud rate, parity, and other settings match the settings of the physical device.

4. Define the logical GKS$CONID to be the logical name of the appropriate device connection.

For more information concerning the terminal allocation process, refer to the appropriate commands in the *VAX/VMS DCL Dictionary.*

There may be times when you do not wish to define the DEC GKS logical names. In this case, or if you define an invalid value, DEC GKS translates several logical names in the following order:

1. If the logical name GKS$CONID is undefined, DEC GKS translates the logical name TT.
2. DEC GKS then translates TT, which always defaults to your process' default device connection.

If the logical name GKS$WSTYPE translates to the value 0 (GKS$WSTYPE being undefined), then DEC GKS sets the device type to be GKS$K_VT240BW (the value 14, a black and white VT240).

It is possible to specify a number for a device type that is unsupported by the implementation of GKS. For instance, if a user specified the value 999 as the translation for GKS$WSTYPE, the call to the function GKS$OPEN_WS would cause an error, since the workstation type 999 is not a DEC GKS supported graphics handler.

To see the function calls that prevent this from happening, refer to the following code from Example 3-1:

```
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
            .
            .
            .
     * INQUIRY_OKAY, ERROR_STATUS, CATEGORY
            .
            .
            .
        DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
            .
            .
            .
     * INQUIRY_OKAY / 0 /
```

```
❶      CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
     * ERROR_STATUS, CATEGORY )

  C    Make sure that the workstation type is valid.
❷      IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
     *    (( CATEGORY .NE. GKS$K_WSCAT_OUTPUT ) .AND.
     *    ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ))) THEN
              WRITE(6,*)
     *        'The specified workstation type is invalid.'
              WRITE(6,*) 'Error status:', ERROR_STATUS
              STOP
       ENDIF

       CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT,
     * GKS$K_WSTYPE_DEFAULT )
       CALL GKS$ACTIVATE_WS( WS_ID )
           .
           .
           .
       CALL GKS$DEACTIVATE_WS( WS_ID )
       CALL GKS$CLOSE_WS( WS_ID )
       CALL GKS$CLOSE_GKS()
       END
```

The following numbers correspond to the numbers in the previous example:

❶ This code calls the inquiry function GKS$INQ_WS_CATEGORY, which writes the integer value associated with the category of the translated workstation type (GKSK_WSCAT_OUTPUT, GKSK_WSCAT_OUTIN, and so forth) to the argument CATEGORY. If the translation of GKS$K_WSTYPE_DEFAULT is a valid DEC GKS supported workstation, the inquiry returns the value 0 (the constant value: GKS$_SUCCESS) to the argument ERROR_STATUS. If the inquiry function encounters an error, it returns the number of the appropriate error message in ERROR_STATUS. (To review the error message numbers, refer to Appendix D, DEC GKS Error Messages, in the *DEC GKS Reference Manual*.) All inquiry functions use this method to report the success or failure of an inquiry function call.

❷ This code makes sure that the user defined GKS$WSTYPE as a supported workstation type (if ERROR_STATUS = INQUIRY_OKAY), and makes sure that the workstation supports output on either a printer or a terminal (valid categories are GKS$K_WSCAT_OUTPUT and GKS$K_WSCAT_OUTIN). If the workstation is not capable of generating the output that the program requires, this program writes a message to SYS$OUTPUT, and program execution stops.

NOTE

Whether or not you use the DEC GKS constants, you should print the definition file for your supported language to see whether it contains code that can be useful to your application.

3.1.2 Checking the Deferral Mode

As mentioned in Chapter 1, Introducing DEC GKS, the graphics handlers have a default setting for the output deferral mode and the surface regeneration flag. In a device-independent program, you can check the current or default values and change them as your program requires.

By setting the deferral mode, you can *buffer* the generation of output images, if the given workstation supports such buffering, before transmission to the workstation surface. In this manner, you improve overall rate of transmission. In the application program, you can periodically release buffered output so that the display surface reflects the picture defined by the application up to that point in execution.

To understand deferral mode, review the following descriptions of the four possible deferral modes, in increasing order of deferral:

- GKS$K_ASAP—The workstation generates output as soon as possible.
- GKS$K_BNIG—The workstation generates output before the next global interaction.
- GKS$K_BNIL—The workstation generates output before the next local interaction.
- GKS$K_ASTI—The workstation generates output at some time.

An *interaction* is a request for input using the DEC GKS input functions. A global interaction happens on any open workstation, and a local interaction happens on a specified workstation (remember that a DEC GKS program can allocate device connections in order to open and activate several workstations).

Depending on its capabilities, the workstation can defer output at any level up to the level specified in the call to GKS$SET_DEFER_STATE. For example, if you specify GKS$K_ASAP in a call to GKS$SET_DEFER_STATE, the workstation can only generate output as soon as possible if the workstation supports that level. If you specify GKS$K_BNIG, the workstation can use the deferral mode GKS$K_ASAP or GKS$K_BNIG, depending on its capabilities. If you specify GKS$K_BNIL, the workstation can defer output in any of the modes GKSK_ASAP, GKSK_BNIG, or GKS$K_BNIL, depending on its capabilities. If you specify GKS$K_ASTI, the workstation can defer output at any level, as defined by the given workstation.

The implicit regeneration modes are described in Chapter 1, Introducing DEC GKS. Usually, you do not want the workstation to regenerate the picture unless you request an update. Consequently, most applications require a suppression of surface regenerations, giving the application program the power to specify when the device handler regenerates a surface. Remember that a surface regeneration deletes all output primitives not contained in segments.

To see how to check the current deferral and implicit regeneration modes, review the following code from Example 3–1:

```
           IMPLICIT NONE
           INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
           INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
               .
               .
               .
          * DUMMY_INTEGER, WS_TYPE

           CHARACTER*80 DUMMY_STRING

     C     Make sure that the deferral mode and regeneration flag are
     C     properly set.
❶          CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
          * WS_TYPE, DUMMY_INTEGER )

❷          CALL GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,
          * DEF_MODE, REGEN_FLAG )

     C     You can check the status of the inquiry function execution, as
     C     follows:
❸          IF ( ERROR_STATUS .NE. INQUIRY_OKAY ) THEN
                  WRITE(6,*)
          *       'The deferral inquiry caused an error.'
                  WRITE(6,*) 'Error status:', ERROR_STATUS
                  STOP
           ENDIF

     C     Defer output as long as possible and suppress implicit
     C     regenerations.
❹          IF (( DEF_MODE .NE. GKS$K_ASTI ) .OR.
          *   ( REGEN_FLAG .NE. GKS$K_IRG_SUPPRESSED )) THEN
                 CALL GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI,
          *                                 GKS$K_IRG_SUPPRESSED )
           ENDIF
               .
               .
               .
           CALL GKS$DEACTIVATE_WS( WS_ID )
           CALL GKS$CLOSE_WS( WS_ID )
           CALL GKS$CLOSE_GKS()
           END
```

The following numbers correspond to the numbers in the previous example:

❶ This code calls the inquiry function GKS$INQ_WS_TYPE to determine the workstation type that corresponds to WS_ID. You need the workstation type to determine the default deferral mode and regeneration flag of the workstation.

The arguments DUMMY_INTEGER and DUMMY_STRING contain information that is not useful to this particular application yet may be useful in other programs. For more information concerning these arguments, refer to Chapter 12, Inquiry Functions, in the *DEC GKS Reference Manual*.

❷ Using the variable WS_TYPE, which the previous function call initialized, you can inquire about the default deferral mode and regeneration flag. Inquiry functions requiring a workstation type for an argument obtain information from the workstation description table (default information); functions requiring a workstation identifier obtain information from the workstation state list of a particular workstation (current status information).

❸ This code checks the execution of the last inquiry function. If it is uncertain whether an inquiry function call will encounter an error, you can check the error status argument after such calls to inquiry functions. You can transfer control if an inquiry function writes anything but GKS$_SUCCESS to the error status argument.

❹ This code changes the deferral mode to GKS$K_ASTI and the regeneration flag to GKS$K_IRG_SUPPRESSED (as opposed to GKS$K_IRG_ALLOWED), if those are not the workstation's default values. By specifying these values, the device buffers output (for as long as the device's capabilities allow up to "at some device-determined time") and suppresses all required surface regenerations until the application updates the surface.

NOTE

When debugging your DEC GKS programs, you may wish to see generated output as you debug. To do this, set the deferral mode to GKS$K_ASAP. After you debug your program, you can set the deferral mode to any desired mode.

3.1.3 Setting Workstation-Dependent Output Attributes

Just after the data initialization section in Example 3–1, the application sets the character height attribute to be LARGER (the real value 0.04, specified in world coordinates). Character height is called a *geometric* attribute. Geometric attributes affect the size or positioning of output primitives, in world coordinate units. Since you express these attributes in world coordinate units, the effect is transformable to any workstation surface (although the results differ according to hardware capabilities); the geometric attributes are device-independent.

In that section of Example 3–1, the application also sets the attributes marker type, fill interior style, line type, and line width. These attributes are called *nongeometric* attributes. Nongeometric attributes affect the size or pattern of an output primitive in scale factors (real values) and nominal sizes (real values representing the default size as determined by the graphics handler). Since the device handler can obtain an unsupported value by multiplying the scale factor times the nominal size, the nongeometric attributes are device-dependent.

Use caution when setting device-dependent attributes in a device-independent program. In Example 3–1, setting the marker type, fill interior style, and line type is not difficult because the application specifies settings that are required by the standard; all DEC GKS graphics handlers support the settings used in this application.

However, depending on the capabilities of the device, this application could specify a line width that results in a value unsupported by the device. You may need to inquire about the maximum supported line width before you set this attribute.

Review the following code from Example 3–1 that checks the maximum supported line width.

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
         .
         .
         .
     * DUMMY_INT_ARRAY( 50 )
      REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
         .
         .
         .
     * NOM_WIDTH, MAX_WIDTH
         .
         .
         .
C     Make sure that you don't ask for a line wider than the
C     workstation's widest line.
❶     CALL GKS$INQ_PLINE_FAC( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, %DESCR( DUMMY_INT_ARRAY ), DUMMY_INTEGER,
     * NOM_WIDTH, DUMMY_REAL, MAX_WIDTH, DUMMY_INTEGER,
     * DUMMY_INTEGER )

❷     DO WHILE (( WIDER * NOM_WIDTH ) .GT. MAX_WIDTH )
         WIDER = WIDER - 0.1
      ENDDO

      CALL GKS$SET_PLINE_LINEWIDTH( WIDER )
         .
         .
         .
```

```
        CALL GKS$DEACTIVATE_WS( WS_ID )
        CALL GKS$CLOSE_WS( WS_ID )
        CALL GKS$CLOSE_GKS()
        END
```

The following numbers correspond to the numbers in the previous example:

❶ This code inquires about the device's polyline facilities, including the maximum supported line width. The variables DUMMY_INT, DUMMY_INT_ARRAY, and DUMMY_REAL are arguments whose values are of no use to this application. For more information concerning GKS$INQ_PLINE_FAC, refer to Chapter 10, Inquiry Functions, in the *DEC GKS Reference Manual.*

❷ This code checks to make sure that the line width, resulting from the multiplication of the nominal width and the scale factor specified by the application, is not greater than the maximum supported width. This code equates WIDER with the scale factor that produces the largest line width allowed by the device handler.

3.1.4 Working with Color and Monochrome Devices

When Example 3–1 creates cell arrays, changes the fill area interior color index, or changes the color representation of a color index, it must check for three conditions, as follows:

1. If the device can produce color primitives.
2. If the device is monochrome, but can produce shades of a single color.
3. If the device is strictly monochrome, supporting only two colors.

To see how Example 3–1 handles the three different conditions, review the following code example:

```
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
            .
            .
            .
       * ERROR_STATUS, DUMMY_INT, COLOR_FLAG, NUM_INDEXES,
       * CATEGORY THREE, DARK, LIGHT, NEW_FRAME_FLAG

        REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
            .
            .
            .
       * BW_X_VALUES( 9 ), BW_Y_VALUES( 9 ), BW_RED_INTENS,
       * BW_GREEN_INTENS, BW_BLUE_INTENS
```

```
         DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
            .
            .
            .
        * BW_RED_INTENS / 1.0 /, BW_GREEN_INTENS / 0.0 /,
        * BW_BLUE_INTENS / 0.0 /, BW_NUM_PTS / 9 /,
        * THREE / 3 /, DARK / 3 /, LIGHT / 2 /

         DATA BW_X_VALUES / 0.0, 0.0, 0.2, 0.2, 0.25, 0.25,
        * 1.0, 1.0, 0.0 /
         DATA BW_Y_VALUES / 0.0, 0.15, 0.15, 0.3, 0.3, 0.15,
        * 0.15, 0.0, 0.0 /
            .
            .
            .
   C     Check to see whether you are working with a color workstation.
❶        CALL GKS$INQ_COLOR_FAC( WS_TYPE, ERROR_STATUS,
        * DUMMY_INT, COLOR_FLAG, NUM_INDEXES )

   C     For all workstations with only 2 color indexes, use
   C     GKS$FILL_AREA instead of GKS$CELL_ARRAY for the sidewalk and road.
❷        IF ( NUM_INDEXES .LT. THREE ) THEN
              CALL GKS$SET_FILL_INT_STYLE(
        *              GKS$K_INTSTYLE_HATCH )
              CALL GKS$FILL_AREA( BW_NUM_PTS, BW_X_VALUES,
        *              BW_Y_VALUES )
              CALL GKS$SET_FILL_INT_STYLE(
        *              GKS$K_INTSTYLE_SOLID )
         ELSE
              CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
        *              SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL,
        *              SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
        *              %DESCR( SIDE_COLORS ) )
              CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
        *              ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL,
        *              ROAD_OFF_ROW, ROAD_NUM_COL, ROAD_NUM_ROW,
        *              %DESCR( ROAD_COLORS ) )
         ENDIF

         CALL GKS$CREATE_SEG( LAND_HOUSE )
         CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )

   C     Only change the color index if working with a workstation
   C     with more than two color indexes.
❸        IF ( NUM_INDEXES .GE. THREE ) THEN
              CALL GKS$SET_FILL_COLOR_INDEX( DARK )
         ENDIF

         CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
         CALL GKS$CLOSE_SEG()

   C     If using a two-color-index workstation, do not change the color
   C     representation.
❹        IF ( NUM_INDEXES .LT. THREE ) THEN
              GO TO 100
         ENDIF
```

```
      C     Set the color representation for color workstations with limited
      C     representation capabilities.
❺           IF (( NUM_COLORS .LE. 8 ) .AND.
          *    ( COLOR_FLAG .EQ. GKS$K_COLOR )) THEN
                 CALL GKS$SET_COLOR_REP( WS_ID, LIGHT,
          *               1.0, 0.0, 1.0 )
                 CALL GKS$SET_COLOR_REP( WS_ID, DARK,
          *               0.0, 1.0, 1.0 )
            ELSE
      C     The color representation change will alter the shading on these
      C     monochrome workstations.
                 IF ( COLOR_FLAG .EQ. GKS$K_MONOCHROME ) THEN
                      CALL GKS$SET_COLOR_REP( WS_ID, DARK,
          *                BW_RED_INTENS, BW_GREEN_INTENS,
          *                BW_BLUE_INTENS )
                 ELSE
      C     Change the color representation for the rest of the color
      C     workstations.
                      CALL GKS$SET_COLOR_REP( WS_ID, DARK,
          *                RED_INTENS_1, GREEN_INTENS_1,
          *                BLUE_INTENS_1 )
                 ENDIF
                 CALL GKS$SET_COLOR_REP( WS_ID, LIGHT,
          *               RED_INTENS_2, GREEN_INTENS_2,
          *               BLUE_INTENS_2 )
            ENDIF

      100   CONTINUE

      C     Check to see whether the picture on the screen is out of date (if
      C     there is a suppressed implicit regeneration).
❻              CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID,
          *            ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
          *            DUMMY_INTEGER, NEW_FRAME_FLAG )

      C     Release deferred output. Regenerate if necessary.  Press
      C     RETURN when you are finished viewing the picture.
❼           IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
                 CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
                 READ(5,*)
            ELSE
                 CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
                 READ(5,*)
            ENDIF
                 .
                 .
                 .
            CALL GKS$DEACTIVATE_WS( WS_ID )
            CALL GKS$CLOSE_WS( WS_ID )
            CALL GKS$CLOSE_GKS()
            END
```

The following numbers correspond to the numbers in the previous example:

❶ The call to GKS$INQ_COLOR_FAC checks the color facilities of a given device. In this call, several arguments are dummy arguments, the argument COLOR_FLAG determines whether the device is color or monochrome, and the argument NUM_INDEXES determines the number of supported indexes.

If a device supports more than two color indexes, a change to the color representation can change the picture significantly. If a device is monochrome, and if it has more than two color indexes, a change to the color representation of an index value can change portions of the picture to a different shade of the supported foreground color.

❷ This code determines whether the program draws the sidewalk and road as cell arrays or as one filled area.

If a device does not support a particular color index, it uses the default foreground color to represent the index. So, if you attempt to draw a cell array that requires unsupported color indexes, you generate two solid rectangles. This program chooses to draw a hatched pattern whenever running on a device supporting only two color indexes.

Notice that the program resets the interior fill style so that subsequent calls to GKS$FILL_AREA once again generate a solid fill.

❸ If the device on which the program is running supports more than two color indexes, this code changes the fill color index to DARK. This ensures that the house is a different color or shade than the tree.

The use of the arguments DARK and LIGHT replace the previous arguments RED and BLUE. On a VT241, the color indexes 2 and 3 represent the primary colors red and blue, but VAXstation II/GPX users will notice that those indexes represent the primary colors red and green. Other devices may represent the indexes as two different colors. Since this program changes the color representation later in the program, it serves a purpose to name the color indexes DARK and LIGHT so that they reflect the changes to be made to the index values.

❹ This code determines whether the device supports only two color indexes. If the device has only two indexes, the program chooses not to make a change to the color representations. Otherwise, the program changes the color representations according to the capabilities of the device.

❺ This code conditionally alters the color representation of the index values DARK (3) and LIGHT (2).

If the device has limited color ability, this code makes DARK and LIGHT represent alternative colors. To do this, you pass the appropriate red, green, and blue color intensities, along with the color index whose representation you are changing, to the function GKS$SET_COLOR_REP. The graphics handler maps the color representation values to the closest values supported by the device.

If the device is monochrome, this code changes the color representations so that they represent two different shades of the supported color.

Otherwise, this code changes the color representation to values that produce the color amber for index value LIGHT (2), and the color brown for index value DARK (3). If you run this program on a device that does not support these colors, the device produces the closest supported color.

❻ This code inquires about the current state of the workstation surface. The function GKS$INQ_WS_DEFER_AND_UPDATE passes a value to the argument NEW_FRAME_FLAG that tells you whether the picture on the surface is out of date. If a picture is out of date, the device needs to regenerate the picture on the workstation surface before implementing all changes requested by the program thus far.

❼ If the request for a change to a color representation placed the workstation surface out of date, then this code requests an implicit regeneration (by passing GKS$K_PERFORM_FLAG to GKS$UPDATE_WS). Keep in mind that an implicit regeneration deletes all output primitives not contained in segments. However, most devices of the category GKS$K_WSCAT_OUTIN are able to change a color representation without placing the screen out of date. Devices of the category GKS$K_WSCAT_OUTPUT behave differently according to their capabilities.

If the request for a change to a color representation does not place the workstation surface out of date, then this code releases all deferred output (by passing GKS$K_POSTPONE_FLAG to GKS$UPDATE_WS). This code does *not* perform an implicit regeneration. It simply places all output primitives on the surface so that you can view an up-to-date picture. Many of the programs in the *DEC GKS Reference Manual* use this function call to update a picture for viewing.

The use of the FORTRAN READ statement causes the program to pause so that the user can view the current picture. The user presses RETURN when ready to continue.

NOTE

Although most workstations support the interior fill styles GKS$K_INTSTYLE_HOLLOW, GKS$K_INTSTYLE_SOLID, GKS$K_INTSTYLE_PATTERN, and GKS$K_INTSTYLE_HATCHED, some workstations may not. If an interior style is not supported, DEC GKS uses a device-determined style.

If you make more than one change to a color representation in a device-dependent program, you may want to use the function GKS$INQ_DYN_MOD_WS. By using this function, you can determine whether the device requires an implicit regeneration for color representation changes. If you find that the device does not require an implicit regeneration, you do not have to keep checking the current state of NEW_FRAME_FLAG every time you make a change to the color representation, as in Example 3–1.

3.1.5 Requiring an Implicit Regeneration

When reviewing the code in Example 3–1, you see that a change to the color representation can be implemented either dynamically or by implicit regeneration. Other changes are similarly implemented, such as all representation changes (Chapter 5, Generating Output, describes the representation changes), changes to segment attributes (such as highlighting or visibility), and changes to the workstation transformation (for instance, changing the portion of the workstation surface on which to place the picture).

Example 3–1 requests a change in the workstation transformation, as shown in the following example:

```
         IMPLICIT NONE
         INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
         INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
             .
             .
             .
        * DUMMY_INTEGER, NEW_FRAME_FLAG

         REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
             .
             .
             .
        * MAX_X, MAX_Y
             .
             .
             .
   C     Inquire about the maximum device coordinate values.
❶        CALL GKS$INQ_MAX_DS_SIZE( WS_TYPE, ERROR_STATUS,
        * DUMMY_INTEGER, MAX_X, MAX_Y, DUMMY_INTEGER, DUMMY_INTEGER )

   C     Use the lower left quarter of the display surface.
❷        CALL GKS$SET_WS_VIEWPORT( WS_ID, 0.0, MAX_X/2.0, 0.0,
        * MAX_Y/2.0 )

   C     Check to see whether the picture on the screen is out of date.
❸        CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
        * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
        * NEW_FRAME_FLAG )

   C     Release deferred output. Regenerate if necessary.  Press
   C     RETURN when you are finished viewing the picture.
         IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
              READ(5,*)
         ELSE
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
              READ(5,*)
         ENDIF

         CALL GKS$DEACTIVATE_WS( WS_ID )
         CALL GKS$CLOSE_WS( WS_ID )
         CALL GKS$CLOSE_GKS()
         END
```

The following numbers correspond to the numbers in the previous example:

❶ This code inquires about the size of the workstation surface. The function GKS$INQ_MAX_DS_SIZE writes the maximum X and Y values of the device coordinate plane in the arguments MAX_X and MAX_Y.

❷ This code changes the portion of the workstation surface on which to output the picture (in this example, to the lower left quarter of the workstation surface). To do this, you calculate half of the X and Y maximum values, and then specify 0.0 to half for both the X and Y maximum values as the portion of the workstation surface.

❸ Since a change to the portion of the workstation on which to output pictures can be made dynamically or by implicit regeneration, you need to check the NEW_FRAME_FLAG argument passed to GKS$INQ_WS_DEFER_AND_UPDATE. Again, if you change the workstation viewport often, it is more efficient to call GKS$INQ_DYN_MOD_WS to determine whether the device handler makes the change dynamically or by implicit regeneration.

Also, you do not need to update a workstation just before the end of a program (as in Example 3–1). DEC GKS releases deferred output and if needed, regenerates the surface, before closing a workstation (which occurs when you call GKS$CLOSE_WS).

3.1.6 The Device-Independent Program

Example 3–1 lists the complete device-independent program described in the previous sections.

Example 3–1: A Device-Independent Program

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, NUM_STARS, NUM_TREE_PTS,
     * NUM_HOUSE_PTS, NUM_LAND_PTS, SIDE_OFF_COL,
     * SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
     * SIDE_COLORS( 1, 2 ), ROAD_OFF_COL, ROAD_OFF_ROW,
     * ROAD_NUM_COL, ROAD_NUM_ROW, ROAD_COLORS( 10, 1 ),
     * LAND_HOUSE, LIGHT, DARK, ERROR_STATUS, INQUIRY_OKAY,
     * CATEGORY, DUMMY_INTEGER, WS_TYPE, DEF_MODE,
     * REGEN_FLAG, DUMMY_INT_ARRAY( 50 ), BW_NUM_PTS,
     * COLOR_FLAG, NEW_FRAME_FLAG, THREE, NUM_INDEXES,
     * NUM_COLORS

      REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
     * STARS_Y_VALUES( 6 ), TREE_X( 29 ), TREE_Y( 29 ),
     * HOUSE_X( 12 ), HOUSE_Y( 12 ), LAND_X( 15 ),
     * LAND_Y( 15 ), SIDE_START_X, SIDE_START_Y, SIDE_DIAG_X,
     * SIDE_DIAG_Y, ROAD_START_X, ROAD_START_Y, ROAD_DIAG_X,
     * ROAD_DIAG_Y, RED_INTENS_1, RED_INTENS_2,
     * GREEN_INTENS_1, GREEN_INTENS_2, BLUE_INTENS_1,
     * BLUE_INTENS_2, LARGER, WIDER, MAX_WIDTH, DUMMY_REAL,
     * BW_RED_INTENS, BW_GREEN_INTENS, BW_BLUE_INTENS,
     * BW_X_VALUES( 9 ), BW_Y_VALUES( 9 ), NOM_WIDTH, MAX_X,
     * MAX_Y

      CHARACTER*80 DUMMY_STRING

      DATA WS_ID / 1 /, TEXT_START_X / 0.05 /,
     * TEXT_START_Y / 0.9 /, NUM_STARS / 6 /,
     * NUM_TREE_PTS / 29 /, NUM_HOUSE_PTS / 12 /,
     * NUM_LAND_PTS / 15 /, SIDE_START_X / 0.2 /,
     * SIDE_START_Y / 0.3 /, SIDE_DIAG_X / 0.25 /,
     * SIDE_DIAG_Y / 0.15 /, SIDE_OFF_COL / 1 /,
     * SIDE_OFF_ROW / 1 /, SIDE_NUM_COL / 1 /,
     * SIDE_NUM_ROW / 2 /, ROAD_START_X/ 0.0 /,
     * ROAD_START_Y / 0.15 /, ROAD_DIAG_X / 1.0 /,
     * ROAD_DIAG_Y / 0.0 /, ROAD_OFF_COL / 1 /,
     * ROAD_OFF_ROW / 1 /, ROAD_NUM_COL / 10 /,
     * ROAD_NUM_ROW / 1 /, LAND_HOUSE / 1 /,
     * RED_INTENS_1 / 0.56 /, GREEN_INTENS_1 / 0.0 /
     * BLUE_INTENS_1 / 0.0 /, RED_INTENS_2 / 0.8538 /,
     * GREEN_INTENS_2 / 0.6646 /, BLUE_INTENS_2 / 0.2862 /,
     * LIGHT / 2 /, DARK / 3 /, LARGER / 0.04 /,
     * WIDER / 3.0 /, INQUIRY_OKAY / 0 /, BW_RED_INTENS / 1.0 /,
     * BW_GREEN_INTENS / 0.0 /, BW_BLUE_INTENS / 0.0 /,
     * BW_NUM_PTS / 9 /, THREE / 3 /
```

(continued on next page)

Example 3–1 (Cont.): A Device-Independent Program

```
      DATA BW_X_VALUES / 0.0, 0.0, 0.2, 0.2, 0.25, 0.25,
     * 1.0, 1.0, 0.0 /
      DATA BW_Y_VALUES / 0.0, 0.15, 0.15, 0.3, 0.3, 0.15,
     * 0.15, 0.0, 0.0 /

      DATA SIDE_COLORS / 3, 2 /
      DATA ROAD_COLORS / 2, 3, 2, 3, 2, 3, 2, 3, 2, 3 /

      DATA STARS_X_VALUES / 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 /
      DATA STARS_Y_VALUES / 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 /

      DATA TREE_X / 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
     * 0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
     * 0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
     * 0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
     * 0.515, 0.51, 0.495, 0.475, 0.425 /
      DATA TREE_Y / 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
     * 0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
     * 0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
     * 0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
     * 0.5, 0.425, 0.38, 0.33, 0.28 /

      DATA HOUSE_X / 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
     * 0.25, 0.25, 0.2, 0.075, 0.1, 0.1 /
      DATA HOUSE_Y / 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
     * 0.75, 0.7, 0.75, 0.6, 0.6, 0.3 /

      DATA LAND_X / 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
     * 0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 /
      DATA LAND_Y / 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
     * 0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
     * 0.385 /

      CALL GKS$OPEN_GKS( 'ERROR_FILE.TXT' )

      CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
     * ERROR_STATUS, CATEGORY )


C     Make sure that the workstation type is valid.
      IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
     *   (( CATEGORY .NE. GKS$K_WSCAT_OUTPUT) .AND.
     *   ( CATEGORY .NE. GKS$K_WSCAT_OUTIN))) THEN
              WRITE(6,*)
     *        'The specified workstation type is invalid.'
              WRITE(6,*) 'Error status:', ERROR_STATUS
              STOP
      ENDIF

      CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT,
     * GKS$K_WSTYPE_DEFAULT )
      CALL GKS$ACTIVATE_WS( WS_ID )
```

(continued on next page)

Example 3–1 (Cont.): A Device-Independent Program

```
C     Make sure that the deferral mode and regeneration flag are
C     properly set.
      CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
     * WS_TYPE, DUMMY_INTEGER )

      CALL GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,
     * DEF_MODE, REGEN_FLAG )

C     You can check the status of the inquiry function execution, as
C     follows:
      IF ( ERROR_STATUS .NE. INQUIRY_OKAY ) THEN
             WRITE(6,*)
     *       'The deferral inquiry caused an error.'
             WRITE(6,*) 'Error status:', ERROR_STATUS
             STOP
      ENDIF

C     Defer output as long as possible and suppress implicit
C     regenerations.
      IF (( DEF_MODE .NE. GKS$K_ASTI ) .OR.
     *   ( REGEN_FLAG .NE. GKS$K_IRG_SUPPRESSED )) THEN
            CALL GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI,
     *                              GKS$K_IRG_SUPPRESSED )
      ENDIF

      CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS )
      CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
      CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED )


C     Make sure that you don't ask for a line wider than the
C     workstation's widest line.
      CALL GKS$INQ_PLINE_FAC( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, %DESCR( DUMMY_INT_ARRAY ), DUMMY_INTEGER,
     * NOM_WIDTH, DUMMY_REAL, MAX_WIDTH, DUMMY_INTEGER,
     * DUMMY_INTEGER )

      DO WHILE (( WIDER * NOM_WIDTH ) .GT. MAX_WIDTH )
         WIDER = WIDER - 0.1
      ENDDO

      CALL GKS$SET_PLINE_LINEWIDTH( WIDER )

      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
     * 'Starry Night' )
      CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
     * STARS_Y_VALUES )
      CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )

C     Check to see whether you are working with a color workstation.
      CALL GKS$INQ_COLOR_FAC( WS_TYPE, ERROR_STATUS,
     * NUM_COLORS, COLOR_FLAG, NUM_INDEXES )
```

(continued on next page)

Example 3–1 (Cont.): A Device-Independent Program

```
C     For all workstations with only 2 color indexes, use
C     GKS$FILL_AREA instead of GKS$CELL_ARRAY for the sidewalk and road.
      IF ( NUM_INDEXES .LT. THREE ) THEN
          CALL GKS$SET_FILL_INT_STYLE(
     *            GKS$K_INTSTYLE_HATCH )
          CALL GKS$FILL_AREA( BW_NUM_PTS, BW_X_VALUES,
     *            BW_Y_VALUES )
          CALL GKS$SET_FILL_INT_STYLE(
     *            GKS$K_INTSTYLE_SOLID )
      ELSE
          CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
     *            SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL,
     *            SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
     *            %DESCR( SIDE_COLORS ) )
          CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
     *            ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL,
     *            ROAD_OFF_ROW, ROAD_NUM_COL, ROAD_NUM_ROW,
     *            %DESCR( ROAD_COLORS ) )
      ENDIF


      CALL GKS$CREATE_SEG( LAND_HOUSE )
      CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )

C     Only change the color index if working with a workstation
C     with more than two color indexes.
      IF ( NUM_INDEXES .GE. THREE ) THEN
          CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()

C     If using a two-color-index workstation, do not change the color
C     representation.
      IF ( NUM_INDEXES .LT. THREE ) THEN
          GO TO 100
      ENDIF

C     Set the color representation for color workstations with limited
C     representation capabilities.
      IF (( NUM_COLORS .LE. 8 ) .AND.
     *   ( COLOR_FLAG .EQ. GKS$K_COLOR )) THEN
          CALL GKS$SET_COLOR_REP( WS_ID, LIGHT,
     *            1.0, 0.0, 1.0 )
          CALL GKS$SET_COLOR_REP( WS_ID, DARK,
     *            0.0, 1.0, 1.0 )
      ELSE
```

(continued on next page)

Example 3–1 (Cont.): A Device-Independent Program

```
C     The color representation change will alter the shading on these
C     monochrome workstations.
          IF ( COLOR_FLAG .EQ. GKS$K_MONOCHROME ) THEN
               CALL GKS$SET_COLOR_REP( WS_ID, DARK,
     *              BW_RED_INTENS, BW_GREEN_INTENS,
     *              BW_BLUE_INTENS )
          ELSE
C     Change the color representation for the rest of the color
C     workstations.
               CALL GKS$SET_COLOR_REP( WS_ID, DARK,
     *              RED_INTENS_1, GREEN_INTENS_1,
     *              BLUE_INTENS_1 )
          ENDIF
          CALL GKS$SET_COLOR_REP( WS_ID, LIGHT,
     *             RED_INTENS_2, GREEN_INTENS_2,
     *             BLUE_INTENS_2 )
      ENDIF

100   CONTINUE

C     Check to see whether the picture on the screen is out of date (if
C     there is a suppressed implicit regeneration).
        CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID,
     *          ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
     *          DUMMY_INTEGER, NEW_FRAME_FLAG )

C     Release deferred output. Regenerate if necessary.  Press
C     RETURN when you are finished viewing the picture.
      IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF

C     Inquire about the maximum device coordinate values.
      CALL GKS$INQ_MAX_DS_SIZE( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, MAX_X, MAX_Y, DUMMY_INTEGER, DUMMY_INTEGER )

C     Use the lower left quarter of the display surface.
      CALL GKS$SET_WS_VIEWPORT( WS_ID, 0.0, MAX_X/2.0, 0.0,
     * MAX_Y/2.0 )

C     Check to see whether the picture on the screen is out of date.
      CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NEW_FRAME_FLAG )
```

(continued on next page)

Example 3–1 (Cont.): A Device-Independent Program

```
C     Release deferred output. Regenerate if necessary.  Press
C     RETURN when you are finished viewing the picture.
      IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF

      CALL GKS$DEACTIVATE_WS( WS_ID )
      CALL GKS$CLOSE_WS( WS_ID )
      CALL GKS$CLOSE_GKS()
      END
```

3.2 Presentation of Program Examples in this Manual

From this point forward, all program examples in this manual are written in a device-independent manner. Therefore, you should be able to execute the examples no matter which type of device you use.

Also, in an attempt to present DEC GKS programming techniques without confusing you with the specifics of many different applications, almost all examples in this manual use the same Starry Night picture of the house, tree, horizon, and so forth. Therefore, you can concentrate on the new concepts presented in a given chapter.

Example 3–2 makes a few changes to the program thus far presented, as follows:

- This program splits the task of setting up, drawing the picture, and cleaning up into three distinct subroutines.
- This program places each primitive in a separate segment.
- Some of the inquiry function calls and INCLUDE statements need to be used in several subroutines.

Example 3–2 presents the base example for all other examples in this manual (Starry Night). The remaining examples in the manual slightly alter these subroutines and add new subroutines to this program.

Example 3–2: The User Manual Program Example Template

```
      IMPLICIT NONE
      INTEGER WS_ID, HOUSE, TREE, HORIZON, STARS, TITLE,
     * SIDE, ROAD

      DATA WS_ID / 1 /, TITLE / 1 /, STARS / 2 /, TREE / 3 /,
     * SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /, HOUSE / 7 /

      CALL SET_UP( WS_ID )
      CALL DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
      CALL CLEAN_UP( WS_ID )

      END

C     ************************************************************
C     Set up the DEC GKS and the workstation environments...
      SUBROUTINE SET_UP( WS_ID )
```

(continued on next page)

Example 3–2 (Cont.): The User Manual Program Example Template

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, ERROR_STATUS, CATEGORY, INQUIRY_OKAY,
     * DUMMY_INTEGER, DEF_MODE, REGEN_FLAG, WS_TYPE

      CHARACTER*80  DUMMY_STRING

      DATA INQUIRY_OKAY / 0 /

      CALL GKS$OPEN_GKS( 'Error_file.txt' )

      CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
     * ERROR_STATUS, CATEGORY )

C     Make sure that the workstation type is valid.
      IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
     *   (( CATEGORY .NE. GKS$K_WSCAT_OUTPUT ) .AND.
     *   ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ))) THEN
             WRITE(6,*)
     *       'The specified workstation type is invalid.'
             WRITE(6,*) 'Error status:', ERROR_STATUS
             STOP
      ENDIF

      CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT,
     * GKS$K_WSTYPE_DEFAULT )
      CALL GKS$ACTIVATE_WS( WS_ID )

C     Make sure that the deferral mode and regeneration flag are
C     properly set.
      CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
     * WS_TYPE, DUMMY_INTEGER )

      CALL GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,
     * DEF_MODE, REGEN_FLAG )

C     You can check the status of the inquiry function execution, as
C     follows:
      IF ( ERROR_STATUS .NE. INQUIRY_OKAY ) THEN
             WRITE(6,*)
     *       'The deferral inquiry caused an error.'
             WRITE(6,*) 'Error status:', ERROR_STATUS
             STOP
      ENDIF

C     Defer output as long as possible and suppress implicit
C     regenerations.
      IF (( DEF_MODE .NE. GKS$K_ASTI ) .OR.
     *   ( REGEN_FLAG .NE. GKS$K_IRG_SUPPRESSED )) THEN
            CALL GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI,
     *                                GKS$K_IRG_SUPPRESSED )
      ENDIF

      RETURN
      END
```

(continued on next page)

```
C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, NUM_STARS, NUM_TREE_PTS,
     * NUM_HOUSE_PTS, NUM_LAND_PTS, SIDE_OFF_COL,
     * SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
     * SIDE_COLORS( 1, 2 ), ROAD_OFF_COL, ROAD_OFF_ROW,
     * ROAD_NUM_COL, ROAD_NUM_ROW, ROAD_COLORS( 10, 1 ),
     * LIGHT, DARK, ERROR_STATUS, INQUIRY_OKAY,
     * DUMMY_INTEGER, WS_TYPE, DUMMY_INT_ARRAY( 50 ),
     * COLOR_FLAG, NUM_INDEXES, THREE, BW_NUM_PTS

      CHARACTER*80 DUMMY_STRING

      REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
     * STARS_Y_VALUES( 6 ), TREE_X( 29 ), TREE_Y( 29 ),
     * HOUSE_X( 12 ), HOUSE_Y( 12 ), LAND_X( 15 ),
     * LAND_Y( 15 ), SIDE_START_X, SIDE_START_Y, SIDE_DIAG_X,
     * SIDE_DIAG_Y, ROAD_START_X, ROAD_START_Y, ROAD_DIAG_X,
     * ROAD_DIAG_Y, LARGER, WIDER, MAX_WIDTH, DUMMY_REAL,
     * NOM_WIDTH, BW_X_VALUES( 9 ), BW_Y_VALUES( 9 )

      DATA TEXT_START_X / 0.05 /,
     * TEXT_START_Y / 0.9 /, NUM_STARS / 6 /,
     * NUM_TREE_PTS / 29 /, NUM_HOUSE_PTS / 12 /,
     * NUM_LAND_PTS / 15 /, SIDE_START_X / 0.2 /,
     * SIDE_START_Y / 0.15 /, SIDE_DIAG_X / 0.25 /,
     * SIDE_DIAG_Y / 0.3 /, SIDE_OFF_COL / 1 /,
     * SIDE_OFF_ROW / 1 /, SIDE_NUM_COL / 1 /,
     * SIDE_NUM_ROW / 2 /, ROAD_START_X/ 0.0 /,
     * ROAD_START_Y / 0.15 /, ROAD_DIAG_X / 1.0 /,
     * ROAD_DIAG_Y / 0.0 /, ROAD_OFF_COL / 1 /,
     * ROAD_OFF_ROW / 1 /, ROAD_NUM_COL / 10 /,
     * ROAD_NUM_ROW / 1 /, LIGHT / 2 /, DARK / 3 /,
     * LARGER / 0.04 /, WIDER / 3.0 /, INQUIRY_OKAY / 0 /,
     * THREE / 3 /, BW_NUM_PTS / 9 /

      DATA BW_X_VALUES / 0.0, 0.0, 0.2, 0.2, 0.25, 0.25,
     * 1.0, 1.0, 0.0 /
      DATA BW_Y_VALUES / 0.0, 0.15, 0.15, 0.3, 0.3, 0.15,
     * 0.15, 0.0, 0.0 /

      DATA SIDE_COLORS / 3, 2 /
      DATA ROAD_COLORS / 2, 3, 2, 3, 2, 3, 2, 3, 2, 3 /

      DATA STARS_X_VALUES / 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 /
      DATA STARS_Y_VALUES / 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 /
```

(continued on next page)

```
      DATA TREE_X / 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
     * 0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
     * 0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
     * 0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
     * 0.515, 0.51, 0.495, 0.475, 0.425 /
      DATA TREE_Y / 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
     * 0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
     * 0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
     * 0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
     * 0.5, 0.425, 0.38, 0.33, 0.28 /

      DATA HOUSE_X / 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
     * 0.25, 0.25, 0.2, 0.075, 0.1, 0.1 /
      DATA HOUSE_Y / 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
     * 0.75, 0.7, 0.75, 0.6, 0.6, 0.3 /

      DATA LAND_X / 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
     * 0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 /
      DATA LAND_Y / 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
     * 0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
     * 0.385 /

      CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS )
      CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
      CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED )

C     Obtain the workstation type.
      CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
     * WS_TYPE, DUMMY_INTEGER )

C     Make sure that you don't ask for a line wider than the
C     workstation's widest line.
      CALL GKS$INQ_PLINE_FAC( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, %DESCR( DUMMY_INT_ARRAY ), DUMMY_INTEGER,
     * NOM_WIDTH, DUMMY_REAL, MAX_WIDTH, DUMMY_INTEGER,
     * DUMMY_INTEGER )

      DO WHILE (( WIDER * NOM_WIDTH ) .GT. MAX_WIDTH )
         WIDER = MAX_WIDTH/NOM_WIDTH
      ENDDO

      CALL GKS$SET_PLINE_LINEWIDTH( WIDER )

      CALL GKS$CREATE_SEG( TITLE )
      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
     * 'Starry Night' )
      CALL GKS$CLOSE_SEG()

      CALL GKS$CREATE_SEG( STARS )
      CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
     * STARS_Y_VALUES )
      CALL GKS$CLOSE_SEG()
```

(continued on next page)

```
      CALL GKS$CREATE_SEG( TREE )
      CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )
      CALL GKS$CLOSE_SEG()

C     Check to see whether you are working with a color workstation.
      CALL GKS$INQ_COLOR_FAC( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, COLOR_FLAG, NUM_INDEXES )

C     For all workstations with less than three color indexes,
C     use GKS$FILL_AREA instead of GKS$CELL_ARRAY for the sidewalk
C     and road.
      IF ( NUM_INDEXES .LT. THREE ) THEN
           CALL GKS$CREATE_SEG( SIDE )
           CALL GKS$SET_FILL_INT_STYLE(
     *              GKS$K_INTSTYLE_HATCH )
           CALL GKS$FILL_AREA( BW_NUM_PTS, BW_X_VALUES,
     *              BW_Y_VALUES )
           CALL GKS$SET_FILL_INT_STYLE(
     *              GKS$K_INTSTYLE_SOLID )
           CALL GKS$CLOSE_SEG()
      ELSE
C           CALL GKS$CREATE_SEG( SIDE )
           CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
     *              SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL,
     *              SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
     *              %DESCR( SIDE_COLORS ) )
C           CALL GKS$CLOSE_SEG()
C           CALL GKS$CREATE_SEG( ROAD )
           CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
     *              ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL,
     *              ROAD_OFF_ROW, ROAD_NUM_COL, ROAD_NUM_ROW,
     *              %DESCR( ROAD_COLORS ) )
C           CALL GKS$CLOSE_SEG()
      ENDIF

      CALL GKS$CREATE_SEG( HORIZON )
      CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y )
      CALL GKS$CLOSE_SEG()

      CALL GKS$CREATE_SEG( HOUSE )
C     Only change the color index if working with a workstation
C     with more than two color indexes.
      IF ( NUM_INDEXES .GE. THREE ) THEN
           CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()

      RETURN
      END
```

(continued on next page)

Example 3–2 (Cont.): The User Manual Program Example Template

```
C       ************************************************************
C       Clean up the DEC GKS and the workstation environments...
        SUBROUTINE CLEAN_UP( WS_ID )

        IMPLICIT NONE
        INTEGER WS_ID

        CALL GKS$DEACTIVATE_WS( WS_ID )
        CALL GKS$CLOSE_WS( WS_ID )
        CALL GKS$CLOSE_GKS()

        RETURN
        END
```

Figures 3–1 through 3–5 illustrate the output from Example 3–2 on five different supported workstations. Notice that the picture occupies a larger or smaller portion of the workstation surface depending on the width-to-height ratio of the device surface. By default, DEC GKS generates output on the largest square that the workstation can produce, with the lower left corner of the default world coordinate space corresponding with the lower left corner of the workstation surface.

Figure 3–1: Starry Night on a VAXstation Workstation

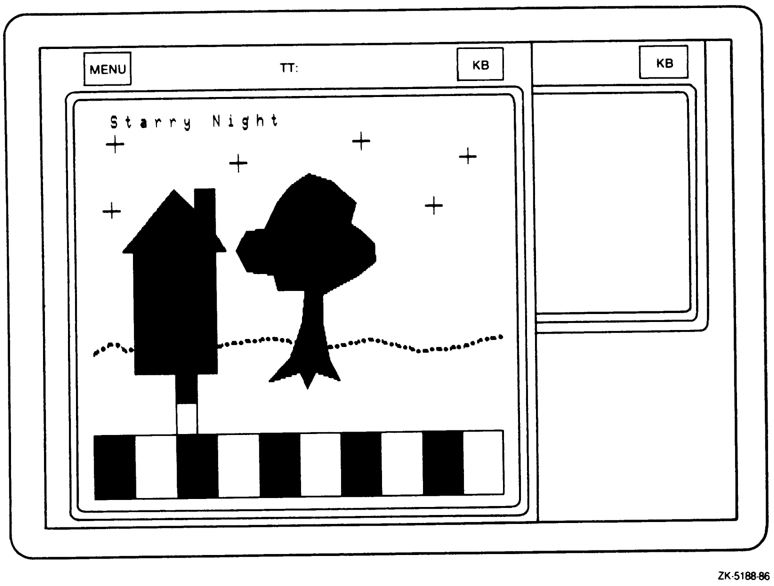


Figure 3–2: Starry Night on a VT240 Terminal

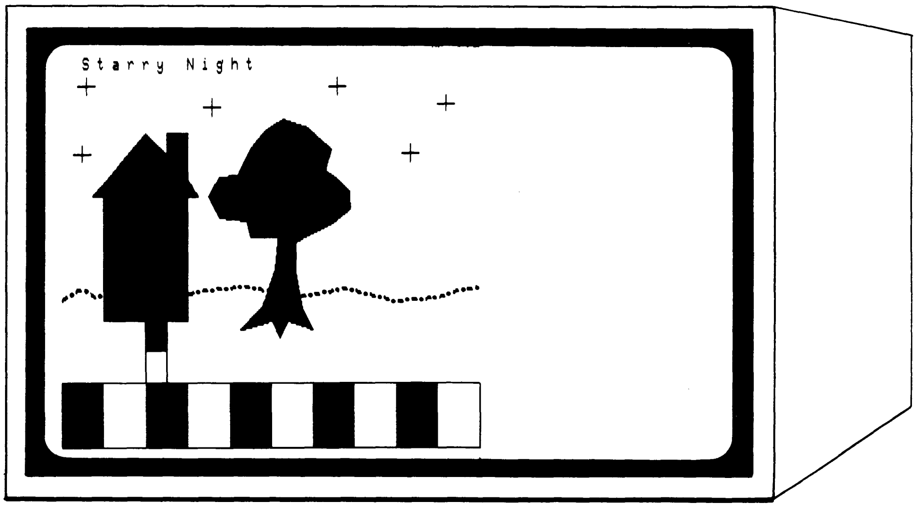


Figure 3–3: Starry Night on a TEKTRONIX—4014 Terminal

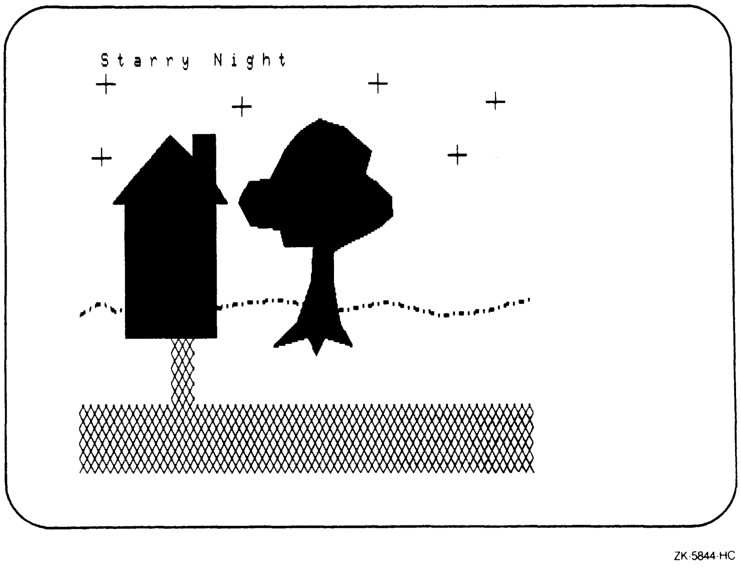
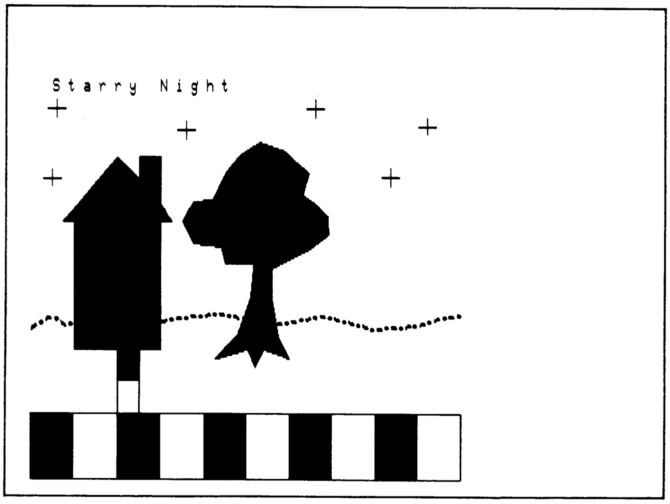
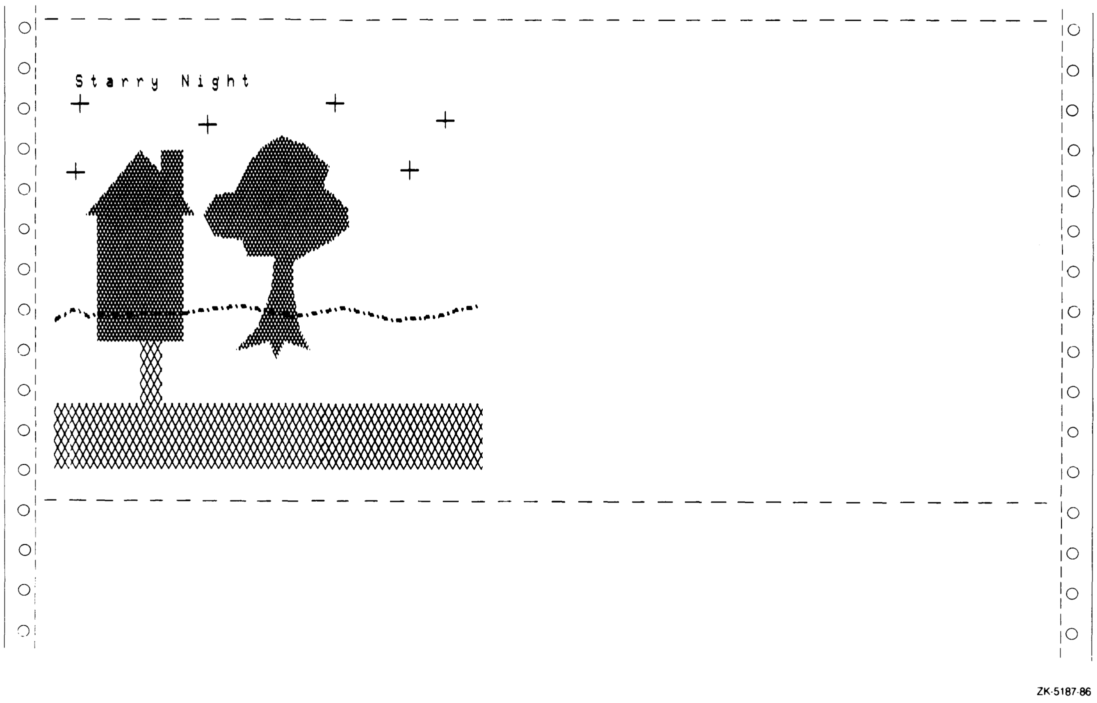


Figure 3–4: Starry Night on an LCG01 Printer



ZK-5189-86

Figure 3–5: Starry Night on an LA100 Printer



Chapter 4

Composing and Transforming Pictures

This chapter provides an overview of picture transformation that you need as a basis for learning additional details of output generation. This chapter discusses the following concepts in detail:

- Normalization transformations (picture composition)
- Clipping of primitives
- Workstation windows (zooming in and out of a picture)
- Workstation viewports (using portions of the device surface)

Using DEC GKS, you can transform segments as well as primitives and pictures. Chapter 5, Generating Output, discusses segment transformations in detail.

NOTE

Section 4.4 contains the code that you must add to the Starry Night program in Example 3–2 to produce the program example contained in this chapter. You may wish to add this code to the base program so that you can execute the program while reading this chapter. The lines of blue code in the example signify the new code that you need to add to Example 3–2.

4.1 DEC GKS Coordinate Systems

In the previous chapters in this manual, you plotted your picture on the default world coordinate range ([0,1] x [0,1]), and DEC GKS drew the picture on the largest square that your workstation could produce, with the lower left corner of the world coordinate range corresponding to the lower left corner of the workstation surface.

The DEC GKS coordinate system offers greater flexibility than that provided by using default transformations. The DEC GKS coordinate systems address the following needs of graphical programming:

- Ability to plot portions of a picture on separate world coordinate ranges (ranges that contain coordinate point values that are relevant to the data)
- Ability to construct a picture on a device-independent coordinate plane
- Ability to show any portion of the composed picture on any portion of any workstation surface.

To meet the needs of graphical programming, DEC GKS uses the following three distinct coordinate systems when producing any picture.

- World coordinate system
- Normalized device coordinate (NDC) system
- Device coordinate system

You use portions of the world coordinate system to plot your output primitives, a portion of the device-independent NDC coordinate plane to compose a complete picture, and a portion of the device coordinate plane to present all or part of your picture on all or part of the surface of the workstation.

The following sections describe the three systems.

4.1.1 The World Coordinate System

The world coordinate plane is an imaginary coordinate plane used to plot a graphical primitive or picture. This imaginary plane consists of an X and a Y axis that extend infinitely in all four directions, and whose intersecting origin is the point (0.0, 0.0).

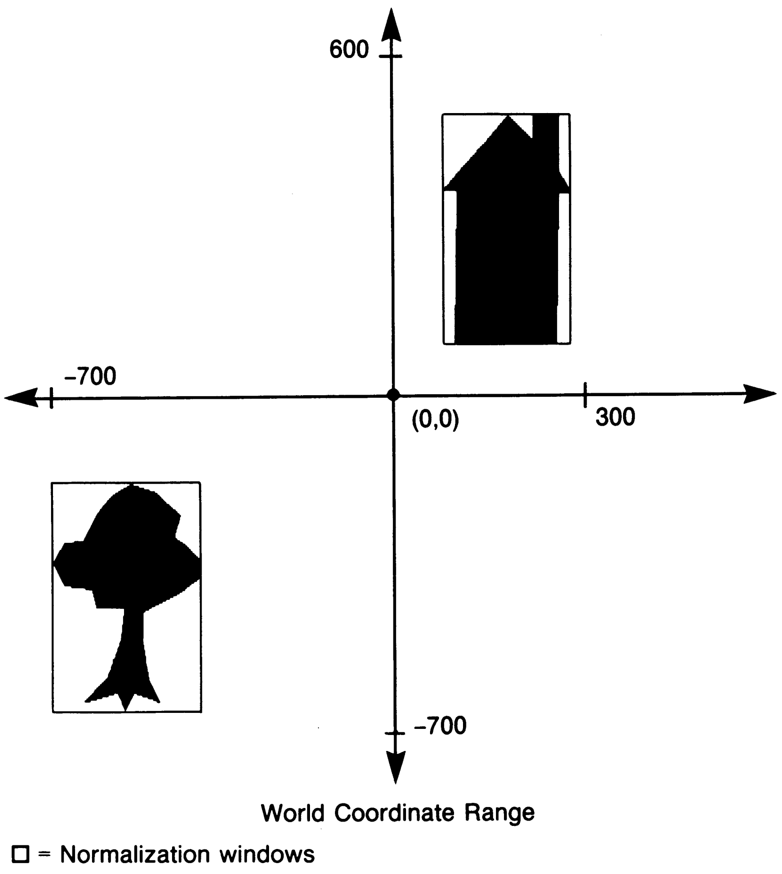
The infinite world coordinate system gives you the flexibility to plot any output primitive according to whatever data is relevant. If your data contains negative numbers, you can use a portion of the world coordinate plane that contains negative X and Y values. Or, if your primitive requires coordinate points from

0.0 to 500.0, you can map the primitive to the rectangular world coordinate range ([0,500] x [0,500]).

You pass the world coordinate points of your plotted image to the DEC GKS output functions. The rectangular portion of the world coordinate system in which you plot a graphical image is called the *normalization window*. By telling DEC GKS which portion of the world coordinate plane you are using for your normalization window, you can *map* different windows to a rectangular portion of the normalized device coordinate (NDC) plane called the *normalization viewport*. The process of mapping from the world coordinate range to the NDC range is called the *normalization transformation*.

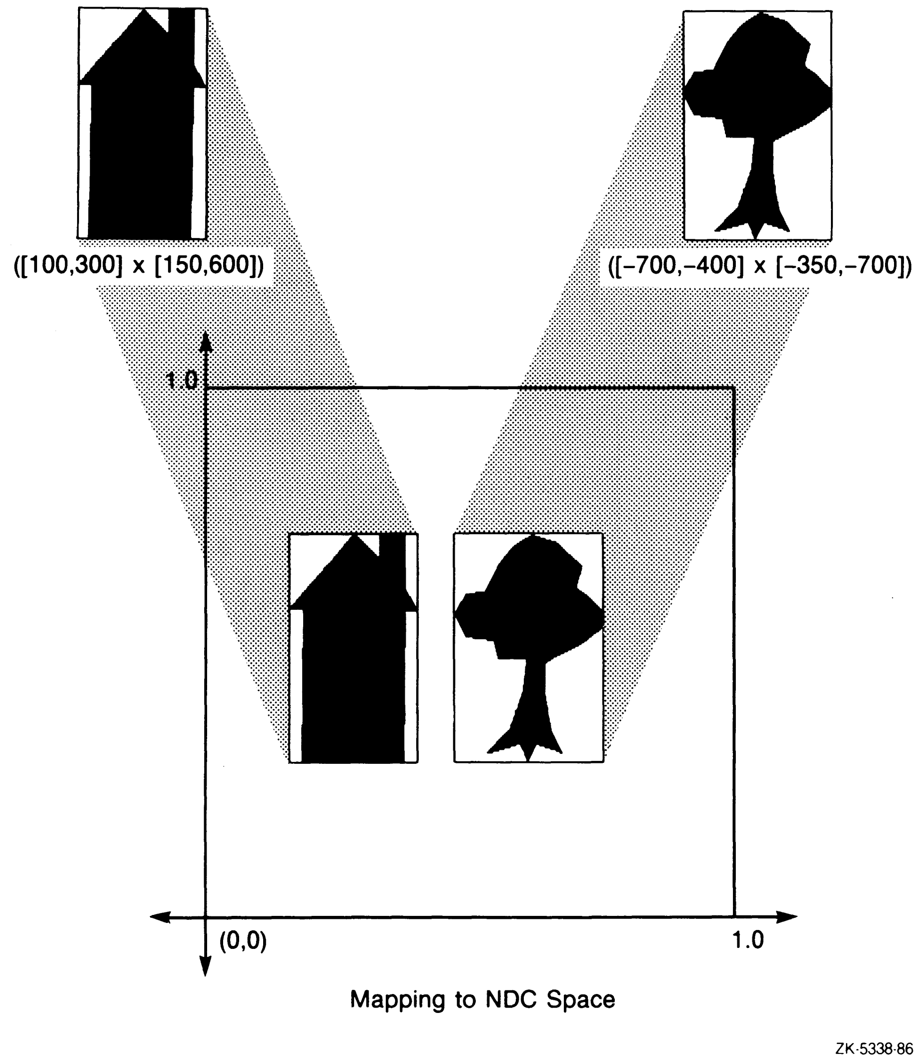
You can envision this process as cutting portions of the world coordinate plane and pasting them on the NDC plane. The world coordinate range is a scratch pad and the NDC range is your pasteboard. Figures 4–1 and 4–2 illustrate the mapping process from normalization windows to corresponding normalization viewports.

Figure 4–1: **Plotting Portions of a Picture in World Coordinates**



ZK-5341-86

Figure 4–2: Composing a Picture on the NDC Plane



Previous chapters in this manual used the DEC GKS default transformations. By default, DEC GKS defines the world coordinate range ([0,1] x [0,1]) to be the normalization window and the NDC range ([0,1] x [0,1]) to be the normalization viewport. After mapping the window to the viewport, DEC GKS maps the NDC range ([0,1] x [0,1]) to the largest portion of the workstation surface that maintains the picture's shape. Since the default portion of the NDC plane is square, DEC GKS maps the picture to the largest square portion of the workstation surface, with the lower left corner of the default NDC space corresponding to the lower left corner of the workstation surface.

4.1.2 The NDC and Device Coordinate Systems

The normalized device coordinate (NDC) system is a device-independent system used to contain an entire picture. You construct the picture using applicable normalization transformations that map primitives from the world coordinate range to NDC space.

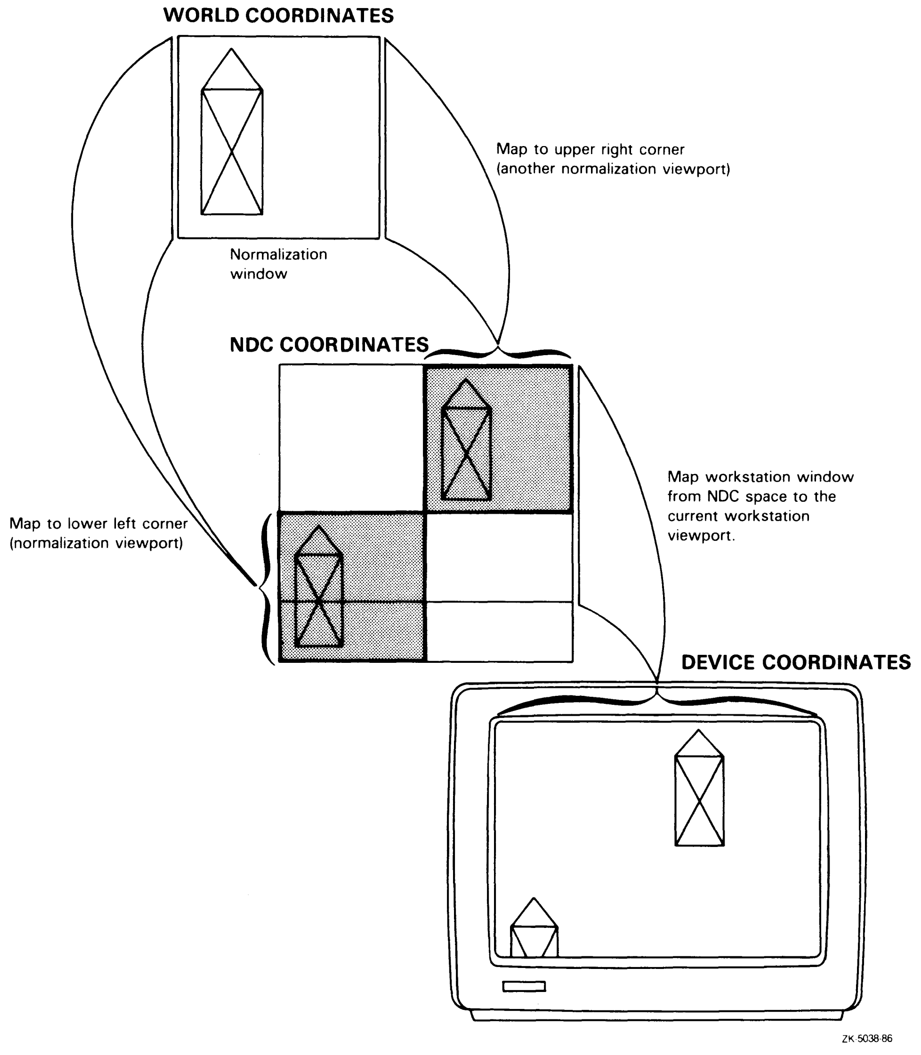
When you generate output, DEC GKS performs a second transformation called the *workstation transformation.* This transformation maps a portion of the NDC space (the *workstation window*) onto a portion of the device coordinate space (*workstation viewport*).

Theoretically, the NDC space is an infinite coordinate plane. You can map any normalization window from the world coordinate plane to any rectangle on the NDC space. However, when DEC GKS maps images from the NDC space to the surface of the physical device, the largest possible portion of the NDC space that can be mapped is ([0,1] x [0,1]). Consequently, only images mapped within this square portion of the NDC space can subsequently be mapped onto the physical device surface. (Remember that the maximum X and Y values in the device coordinate system can be different values on different devices.)

The NDC plane is an intermediate coordinate system that is independent of both the needs of the application (plotting primitives) and of the device coordinate requirements (displaying the picture on the workstation surface). You use the workstation transformation to map all or part of the NDC range ([0,1] x [0,1]) onto all or part of the device coordinate range.

Figure 4–3 illustrates both the normalization and workstation tranformation process.

Figure 4–3: The DEC GKS Transformations


WORLD COORDINATES
Map to upper right corner (another normalization viewport)
Normalization window
NDC COORDINATES
Map workstation window from NDC space to the current workstation viewport.
Map to lower left corner (normalization viewport)
DEVICE COORDINATES
ZK-5038-86

4.2 Composing a Picture

In Example 3–2, all of the primitives in the picture have coordinate points that fall within the default normalization window on the world coordinate plane. DEC GKS maps that range to the normalized device coordinate (NDC) space. In many applications, this is an inconvenient way of plotting pictures. A portion of the world coordinate system that allows easy plotting of the sidewalk and road may not provide an easy system for plotting the house (notice all of the real numbers of different sizes needed to plot the house). Perhaps you want to plot the house without having to use the decimal portion of the coordinate points. Perhaps your application contains data expressed in negative numbers.

To make it easier to plot your pictures, DEC GKS allows you to plot a single primitive (the house, the tree, and so forth) on any rectangular portion of the world coordinate plane, and to map all of these portions onto the NDC space, thereby composing a complete picture.

4.2.1 Changing the Default Normalization Transformation

In subroutine DRAW_PICTURE in Example 3–2, the code plots all output primitives within the default normalization window ([0,1] x [0,1]) and DEC GKS maps them to the default normalization viewport ([0,1] x [0,1]). Since the picture contains several different output primitives, it is unlikely that an application plots all images within the same normalization window.

To illustrate the composition of pictures using various normalization windows and viewports, this section shows how to alter the plotting of the house and how to map the house onto different normalization viewports on the NDC space.

Let's assume that it is desirable to plot all of the house's points without using the decimal portion of the real number world coordinate value. To do this, review the house's current world coordinate values, as follows:

```
C       ************************************************************
C       Draw the picture, and place each primitive in a segment...
        SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
       * ROAD, HOUSE, HORIZON )

        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
            .
            .
            .
        DATA HOUSE_X / 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
       * 0.25, 0.25, 0.2, 0.075, 0.1, 0.1 /
        DATA HOUSE_Y / 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
       * 0.75, 0.7, 0.75, 0.6, 0.6, 0.3 /
            .
            .
            .
```

The range of X values is from 0.075 to 0.325, and the range of Y values is from 0.3 to 0.75. So, to work with numbers that are easier to understand, you can define a normalization window with an X range from 75.0 to 325.0, and a Y range from 300.0 to 750.0 ([75,325] x [300,750]).

To see how to define a new normalization window, review the following code:

```
C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER H_NORM_LEFT,
          .
          .
          .
      DATA H_NORM_LEFT / 1 /
          .
          .
          .
❶     DATA HOUSE_X / 100.0, 300.0, 300.0, 325.00, 300.0, 300.0,
     * 250.0, 250.0, 200.0, 75.0, 100.0, 100.0 /
      DATA HOUSE_Y / 300.0, 300.0, 600.0, 600.0, 640.0, 750.0,
     * 750.0, 700.0, 750.0, 600.0, 600.0, 3.00 /
          .
          .
          .
      CALL GKS$CREATE_SEG( HOUSE )
C     Only change the color index if working with a workstation
C     that has more than two color indexes.
      IF ( NUM_INDEXES .GE. THREE ) THEN
          CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF
❷     CALL GKS$SET_WINDOW( H_NORM_LEFT, 75.0, 325.0, 300.0, 750.0 )
          .
          .
          .
❸     CALL GKS$SELECT_XFORM( H_NORM_LEFT )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

      CALL GKS$CLOSE_SEG()
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

❶ When using a different world window for the plotting of the house, you specify points that fall within the dimensions of the window ([75,325] x [300,750]).

2 This call to GKS$SET_WINDOW associates the normalization window ([75,325] x [300,750]) with the normalization transformation number 1 (H_NORM_LEFT). This program uses the variable H_NORM_LEFT because this normalization transformation eventually places the house in the left portion of the picture.

   Keep in mind that this call does not *change* the current normalization window; it only associates a window with a normalization transformation number.

3 This call to GKS$SELECT_XFORM actually changes the current normalization transformation from the default (whose window has the range ([0,1] x [0,1])) to H_NORM_LEFT (whose window has the range ([75,325] x [300,750])). From this point on (unless you change the normalization transformation again), mapping of all output primitives takes place using the window and viewport associated with H_NORM_LEFT.

If you are plotting a picture in a portion of the world coordinate plane that is not the default range, you need to associate the window dimensions with a normalization number by calling GKS$SET_WINDOW, and you need to tell DEC GKS to use the normalization window and viewport associated with the new normalization number by calling GKS$SELECT_XFORM.

DEC GKS supports a range of normalization transformation numbers from the value 0 to 255. You can either choose a new normalization number each time you wish to use a new window or viewport, or you can redefine the window or viewport associated with a single number, depending on the needs of your application.

The value 0 is a special normalization number. This is the DEC GKS default normalization transformation that maps primitives from the world coordinate range ([0,1] x [0,1]) to the NDC range ([0,1] x [0,1]). All of the program examples in the previous chapters in this manual use the default transformation.

You cannot pass the value 0 to GKS$SET_WINDOW. You cannot change the normalization window and viewport associated with the default normalization transformation. The default normalization is also called the *unity transformation*.

If you need to use a window and viewport with other than the default ranges, you need to associate the new ranges with some other valid normalization transformation number.

If you want to use a normalization viewport other than the default, you need to associate a normalization viewport range with the appropriate normalization number. In the last example, the house is mapped to the default normalization viewport, since the application failed to specify a different viewport range. To see how to map the house into its proper place in the picture, review the following code:

```
C     **********************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER H_NORM_LEFT, UNITY,
          .
          .
          .
      DATA H_NORM_LEFT / 1 /, UNITY / 0 /
          .
          .
          .
      DATA HOUSE_X / 100.0, 300.0, 300.0, 325.00, 300.0, 300.0,
     * 250.0, 250.0, 200.0, 75.0, 100.0, 100.0 /
      DATA HOUSE_Y / 300.0, 300.0, 600.0, 600.0, 640.0, 750.0,
     * 750.0, 700.0, 750.0, 600.0, 600.0, 3.00 /
          .
          .
          .
      CALL GKS$CREATE_SEG( HOUSE )
C     Only change the color index if working with a workstation
C     that has more than two color indexes.
      IF ( NUM_INDEXES .GE. THREE ) THEN
          CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF

      CALL GKS$SET_WINDOW( H_NORM_LEFT, 75.0, 325.0, 300.0, 750.0 )
❶     CALL GKS$SET_VIEWPORT( H_NORM_LEFT, 0.075, 0.325, 0.3, 0.75 )
      CALL GKS$SELECT_XFORM( H_NORM_LEFT )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

      CALL GKS$CLOSE_SEG()

❷     CALL GKS$SELECT_XFORM( UNITY )
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

❶ This code associates a normalization viewport rectangle on the left side of the NDC space with the normalization transformation H_NORM_LEFT. DEC GKS maps the house from the normalization window already associated with H_NORM_LEFT to this viewport. Figure 4–4 illustrates the space occupied by this normalization viewport. Figure 4–5 illustrates the NDC space after generating the house using the normalization transformation H_NORM_LEFT.

❷ This code resets the current normalization number to the value 0 (the number of the unity transformation) so that subsequent calls to output functions use the default normalization window and viewport.

Figure 4–4: Changing Normalization Viewport

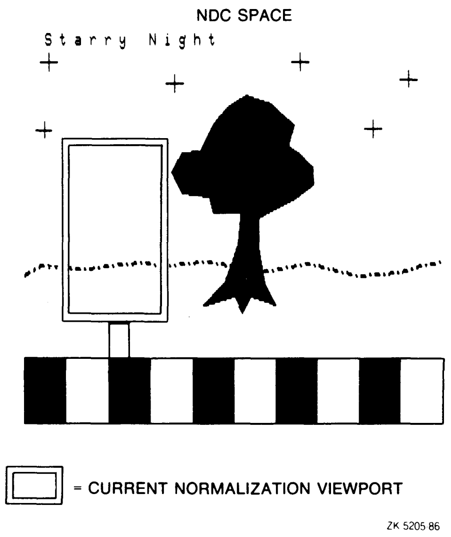
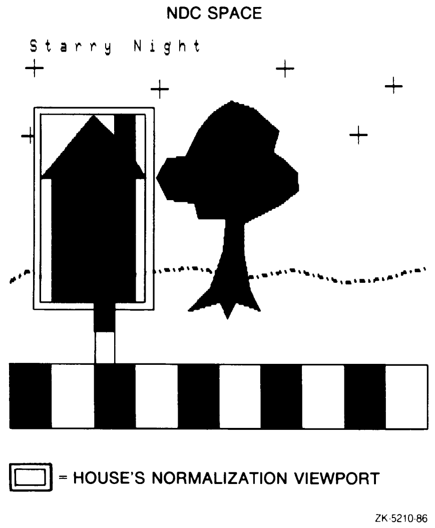


Figure 4–5: Mapping to the New Normalization Viewport



When you execute the code after making these changes to Example 3–2, notice that the example produces the same picture on the device surface as the original example.

4.2.2 Altering the Aspect Ratio

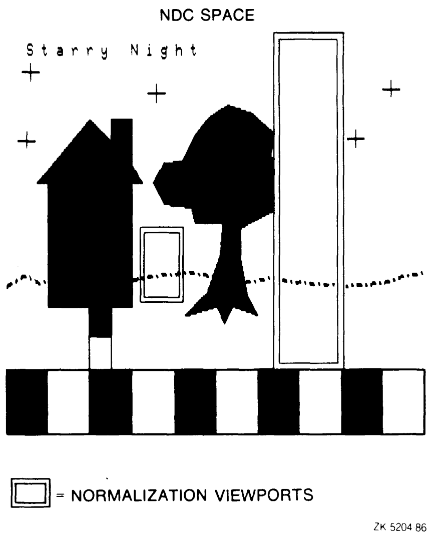
To this point, this chapter has described how to plot a primitive in a different normalization window. This section expands on this idea by describing how to alter the shape of a primitive in a normalization window by altering the dimensions of the normalization viewport.

For example, once you plot the house, you can map that house onto any number of normalization viewports on the NDC space. Not only can you map the house onto different viewports, you can alter the size and proportion of those viewports to alter the shape of the house. The proportionate shape of primitives contained in a window is called the *aspect ratio*. To calculate the aspect ratio of primitives in a given window, divide MAX_WINDOW_UNITS_Y by MAX_WINDOW_UNITS_X.

You can envision the process of altering aspect ratio as stretching or shrinking a primitive to fit inside of the boundary of the normalization viewport. By altering the aspect ratio, you can produce a vanishing effect or you can simulate scaling (shrinking and expanding).

Figure 4–6 outlines two normalization viewports that you can use to map the house to NDC space. When mapping the house from the normalization window to the two designated viewports, DEC GKS maintains the relative position of each of the coordinate points within the viewport. However, in maintaining the relative position of the points, mapping to a normalization viewport whose proportion is different than the corresponding window causes an alteration in the primitives' aspect ratio. For instance, the smaller viewport produces a house shorter and wider than the plotted primitive, and the larger viewport produces a house taller and more narrow than the plotted primitive.

Figure 4–6: Choosing Various Normalization Viewports



The following code shows you how to map the house to several normalization viewports. Figure 4–7 illustrates an approximation of the picture generated by this code. Keep in mind that the picture may appear differently on each device.

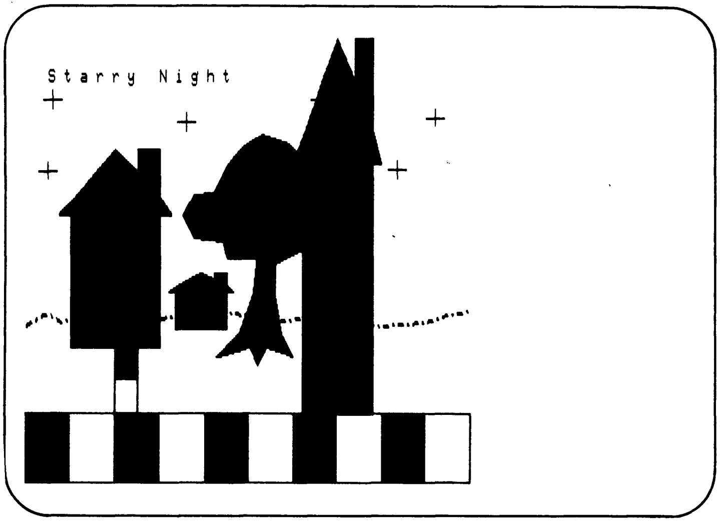
```
C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER H_NORM_LEFT, UNITY, H_NORM_FRONT, H_NORM_BACK,
         .
         .
         .
      DATA UNITY / 0 /, H_NORM_LEFT / 1 /, H_NORM_BACK / 2 /,
     * H_NORM_FRONT / 3 /
         .
         .
         .
      CALL GKS$CREATE_SEG( HOUSE )
C     Only change the color index if working with a color workstation
C     (or a VT125/240 or a VAXstation).
      IF ( NUM_INDEXES .GE. THREE ) THEN
           CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF

      CALL GKS$SET_WINDOW( H_NORM_LEFT, 75.0, 325.0, 300.0, 750.0 )
      CALL GKS$SET_VIEWPORT( H_NORM_LEFT, 0.075, 0.325, 0.3, 0.75 )
      CALL GKS$SELECT_XFORM( H_NORM_LEFT )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

❶     CALL GKS$SET_WINDOW( H_NORM_BACK, 75.0, 325.0, 300.0, 750.0 )
      CALL GKS$SET_VIEWPORT( H_NORM_BACK, 0.32, 0.465, 0.345, 0.47 )
      CALL GKS$SELECT_XFORM( H_NORM_BACK )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

❷     CALL GKS$SET_WINDOW( H_NORM_FRONT, 75.0, 325.0, 300.0, 750.0 )
      CALL GKS$SET_VIEWPORT( H_NORM_FRONT, 0.6, 0.8, 0.15, 1.0 )
      CALL GKS$SELECT_XFORM( H_NORM_FRONT )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

      CALL GKS$CLOSE_SEG()

      CALL GKS$SELECT_XFORM( UNITY )
         .
         .
         .
```

The following numbers correspond to the numbers in the previous example:

❶ This code specifies a normalization window and viewport that DEC GKS associates with the normalization transformation number 2 (H_NORM_BACK). After calling GKS$SELECT_XFORM and passing H_NORM_BACK, DEC GKS maps all output (the house) from the associated window to the associated viewport. The new viewport space is a small rectangular area between the leftmost house and the tree. Mapping to this viewport gives the effect of mapping a small house in the distance. (Notice that the specified window is always the same world coordinate space containing the plotted house.)

❷ This code specifies the same normalization window containing the plotted house, but specifies a tall, thin viewport at the rightmost portion of the picture. After selecting H_NORM_FRONT, DEC GKS maps all generated output from the associated window to the associated viewport. Mapping to this viewport gives the effect of mapping a tall house to the forefront of the picture. Notice how this newly generated house covers, or *occludes*, the tree.

Figure 4–7: Viewing the Aspect Ratio of Transformed Primitives—VT241



When you compose a picture, you need to be aware of the proportion of your normalization window as compared to your normalization viewport. If you map a primitive from a square window to a rectangular viewport whose X dimension is twice its Y dimension, the primitive will appear shorter and fatter than the primitive plotted in the world coordinate space. To maintain the aspect ratio of the plotted picture (as was done with the leftmost house), your normalization window and viewport must be proportionally equivalent.

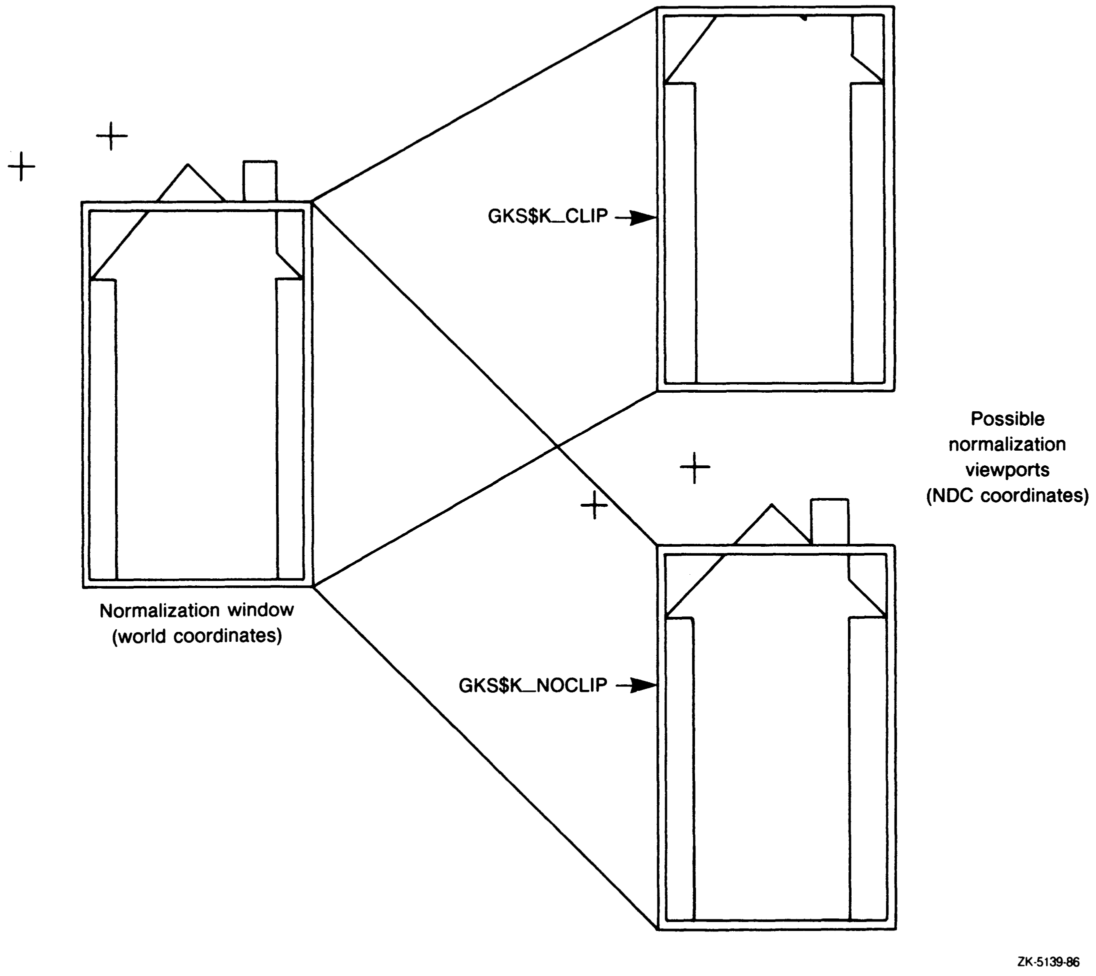
4.2.3 Clipping a Primitive

In previous chapters, when you mapped primitives from the default normalization window to the default viewport, GKS mapped the entire primitive to the NDC plane. There was no portion of any primitive that was omitted from the picture generated on the surface of the workstation.

During normalization transformations involving viewports whose ranges are smaller than the default range ([0,1] x [0,1]), portions of primitives extending outside of the normalization window may or may not be mapped to the NDC space, depending on the current *clipping* flag. The normalization viewport is also called the *clipping rectangle.*

By default, all primitives are clipped at the normalization viewport. To change the current clipping flag, you pass either GKS$K_CLIP or GKS$K_NOCLIP to the function GKS$SET_CLIPPING. Figure 4–8 illustrates the difference in mapping depending on the current status of the clipping flag. Notice in the figure that the stars, which are located completely outside of the normalization window, are mapped to the NDC space when the clipping flag is set to GKS$K_NOCLIP.

Figure 4–8: Clipping Output Primitives


GKS$K_CLIP
Possible normalization viewports (NDC coordinates)
Normalization window (world coordinates)
GKS$K_NOCLIP
ZK-5139-86

4.3 Viewing the Composed Picture

Once you compose a picture using the DEC GKS normalization transformation functions, you must decide how you wish to present the picture on the workstation surface. For instance, you need to decide whether you wish to show the user the entire picture, whether you want to zoom in on particular objects, whether you want to pan across a picture, and how much of the device surface you need in order to display the portion of the picture on a given workstation.

To make decisions concerning picture presentation, you must have a knowledge of the DEC GKS *workstation transformation* process. The workstation transformations take place from NDC space to the device coordinate plane (the surface of the physical device). As with normalization transformations, DEC GKS uses a window and viewport to perform this mapping. DEC GKS maps the picture from the *workstation window* (located in NDC space) to the *workstation viewport* (located on the device coordinate plane). The range of the device coordinate plane is completely device dependent.

By default, DEC GKS maps the composed pictures from the workstation window range ([0,1] x [0,1]) to the workstation viewport range that maintains the picture's aspect ratio. Since the default workstation window is a square, DEC GKS maps the picture to the largest square area on the device plane (which will maintain the aspect ratio of the picture), whose lower left corner is the origin of the device coordinate system (0.0, 0.0).

Unlike the normalization transformations, you have limited control over the workstation windows and viewports since you must work with limitations placed on both ranges. To make the distinction between the two types of transformations, you can think of the normalization transformations as being device-independent picture *composition*, and the workstation transformations as being device-dependent picture *presentation*. (You pass a normalization transformation number when calling the function GKS$SET_WINDOW, but you pass a workstation identifier when calling GKS$SET_WS_WINDOW.) There can be many normalization transformations used to create a single picture, but there is only one current workstation transformation, with one window and viewport used to present a picture.

Even though DEC GKS stores primitives outside the default NDC range ([0,1] x [0,1]), you cannot define a workstation window larger than that default range. If you attempt to do so, DEC GKS generates an error. Simply, there is no way to map a primitive located outside of that default NDC range to the physical device surface. DEC GKS clips all primitives at the workstation window, regardless of the current clipping flag. The clipping flag only controls clipping at the normalization viewport. You can control clipping of *primitives* during output generation, but you cannot control the required clipping of the *picture* at the workstation window.

When working with the workstation viewport, you cannot define a viewport that is larger than the display surface. If you attempt to do so, DEC GKS generates an error. When redefining the workstation viewport, you should use the inquiry function GKS$INQ_MAX_DS_SIZE to determine the limits of the device's coordinate plane.

You need to keep in mind that DEC GKS always maintains the picture's aspect ratio when mapping to the workstation viewport. This means that DEC GKS may not use the entire defined viewport; DEC GKS uses the largest rectangle *within the current workstation viewport* that is proportionately equivalent to current workstation window. So, if the current workstation window is square, DEC GKS maps the contents of the square workstation window to the largest square space within the current workstation viewport beginning at the lower left corner.

Another consideration when working with the workstation transformations is whether or not the screen is out of date. Using most of the DEC GKS supported devices, if you make a change to the workstation window or viewport after you generate output, you need to regenerate the surface of the workstation (if the workstation does not perform this action by default) to implement the changes. In making such a change, you cause all primitives not contained in segments to be cleared from the workstation surface.

Even though your control over the workstation transformation is limited, the effects on the representation of your picture can be quite impressive. As mentioned at the beginning of this section, you can pan across a picture, and zoom in and out of a picture. The remaining sections in this chapter illustrate workstation transformations according to specific tasks.

The following subsections describe methods you use when mapping to the entire workstation window, and describe methods of viewing the composed picture.

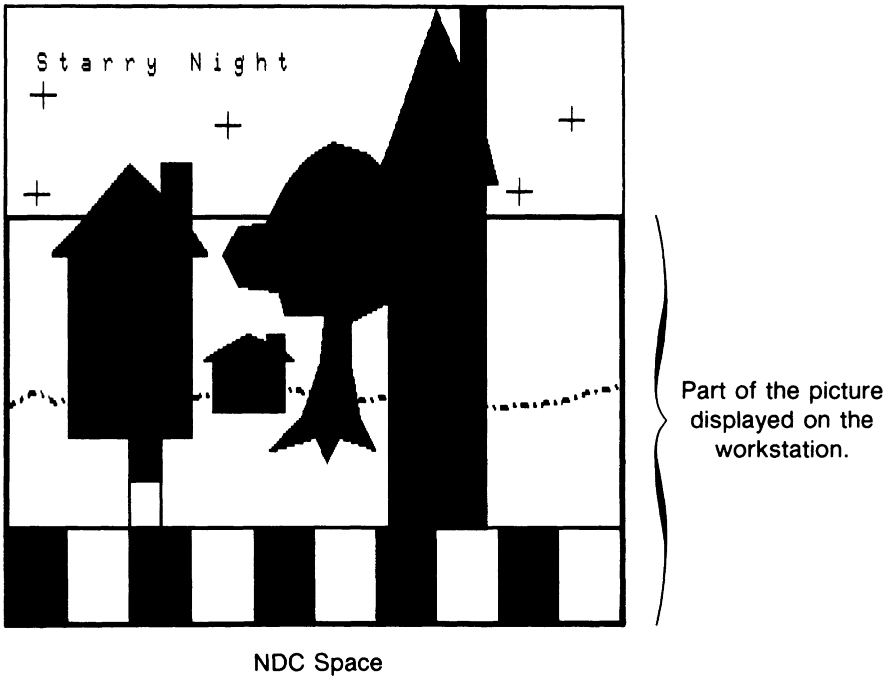
4.3.1 Choosing Between Aspect Ratio and Drawing the Entire Picture

Since DEC GKS always maps the workstation window to the largest rectangle within the current workstation viewport that is proportionately equivalent to current workstation window, the device handler may or may not use the entire surface of the workstation to present the picture. Logically, you need to define a workstation window within the ([0,1] x [0,1]) NDC boundary that has the same proportions as the entire device coordinate plane.

Before you can write a program that uses the entire workstation surface to display the picture, you need to make a decision. Do you want to maintain the shape of your composed picture, or do you want to draw the entire picture on the workstation's surface? When using the entire workstation surface to present a picture, you can either maintain the picture's aspect ratio or show the whole picture, but you cannot do both.

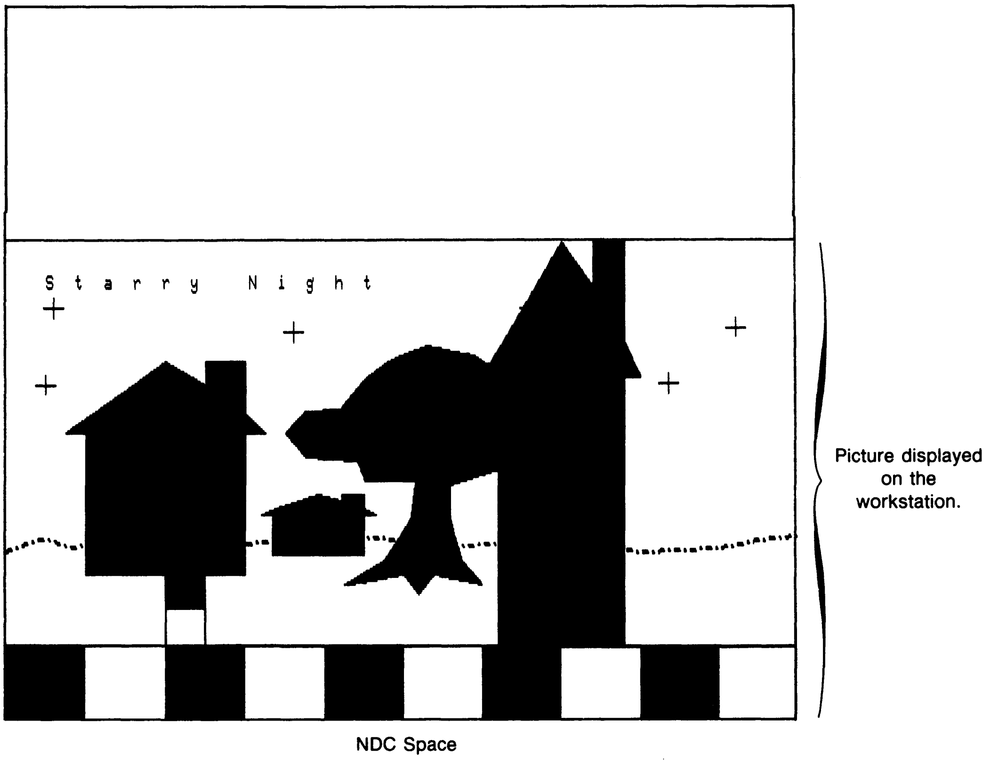
To maintain the shape of the Starry Night picture as composed on the NDC plane, you need to define a workstation window within the range ([0,1] x [0,1]) that is proportionate to the dimensions of the device coordinate system. However, if you do this, DEC GKS clips the portion of the picture exceeding the newly proportionate workstation window boundary. To maintain shape while using the entire display surface, you probably will not be able to show the entire picture. Figure 4–9 illustrates the portion of the NDC space mapped to the workstation surface.

Figure 4–9: Maintaining Shape Over the Portion of Visible Picture



If you decide that you want to show all of the primitives in the composed picture, you must map all of your primitives into the portion of the NDC range ([0,1] x [0,1]) that is proportionate to the device coordinate plane. However, if you adjust the mapping of all of your primitives in this way, you alter the aspect ratio of your primitives. Simply, your picture appears flattened or stretched. The program examples in this chapter alter the aspect ratio in order to present the entire composed picture. Figure 4–10 illustrates the effects of altering the normalization viewports so as to display all generated primitives.

Figure 4–10: Maintaining Visible Primitives Over Picture Shape



ZK-5339-86

If both aspect ratio and entire picture presentation is crucial to your application, you need to use a smaller portion of your workstation's surface that is proportional to your picture. Section 4.3.5 describes how to alter the portion of the workstation surface used to present a picture.

4.3.2 Mapping a Picture to the Entire Workstation Surface

As mentioned in the previous section, the examples in this chapter define a workstation window that is proportionate to the device coordinate plane, and then map all primitives within the proportionate area of the NDC space. This process alters the shape of the Starry Night program. The most difficult aspect of this process is to write code that is device independent, when you do not know the exact dimensions of any given display surface.

The following code examples show how to alter the code in the subroutine DRAW_PICTURE in Example 3–2, so that DEC GKS maps from a workstation window with the same proportion as the appropriate device coordinate plane.

```
    C     ***********************************************************
    C     Draw the picture, and place each primitive in a segment...
          SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
         * ROAD, HOUSE, HORIZON )

          IMPLICIT NONE
          INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
          INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
              .
              .
              .
         * WS_TYPE, ERROR_STATUS, CATEGORY, LARGEST_VIEWPORT

          REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
              .
              .
              .
         * DISPLAY_X, DISPLAY_Y, MAX_COORD, RATIO_X, RATIO_Y
              .
              .
              .
          CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
         * ERROR_STATUS, CATEGORY )

❶         IF ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ) THEN
               WRITE(6,*)
         *        'The specified workstation type is not OUTIN.'
               STOP
          ENDIF

❷         CALL GKS$INQ_MAX_DS_SIZE( WS_TYPE, ERROR_STATUS,
         * DUMMY_INTEGER, DISPLAY_X, DISPLAY_Y, DUMMY_INTEGER,
         * DUMMY_INTEGER )

❸         MAX_COORD = MAX( DISPLAY_X, DISPLAY_Y )

❹         IF (( DISPLAY_X / MAX_COORD ) .EQ. 1.0 ) THEN
                RATIO_X = 1.0
                RATIO_Y = DISPLAY_Y / MAX_COORD
          ELSE
                RATIO_X = DISPLAY_X / MAX_COORD
                RATIO_Y = 1.0
          ENDIF
```

```
❺      CALL GKS$SET_VIEWPORT( LARGEST_VIEWPORT, 0.0, RATIO_X, 0.0,
     * RATIO_Y )
       CALL GKS$SELECT_XFORM( LARGEST_VIEWPORT )

❻      CALL GKS$SET_WS_WINDOW( WS_ID, 0.0, RATIO_X, 0.0, RATIO_Y )
       CALL GKS$SET_WS_VIEWPORT( WS_ID, 0.0, DISPLAY_X, 0.0,
     * DISPLAY_Y )
           .
           .
           .
       CALL GKS$CREATE_SEG( HOUSE )
C      Only change the color index if working with a color workstation
C      (or a VT125/240 or a VAXstation).
       IF ( NUM_INDEXES .GE. THREE ) THEN
            CALL GKS$SET_FILL_COLOR_INDEX( DARK )
       ENDIF

❼      CALL GKS$SET_WINDOW( H_NORM_LEFT, 75.0, 325.0, 300.0, 750.0 )
       CALL GKS$SET_VIEWPORT( H_NORM_LEFT, 0.075*RATIO_X, 0.325*RATIO_X,
     * 0.3*RATIO_Y, 0.75*RATIO_Y )
       CALL GKS$SELECT_XFORM( H_NORM_LEFT )

       CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

       CALL GKS$SET_WINDOW( H_NORM_BACK, 75.0, 325.0, 300.0, 750.0 )
       CALL GKS$SET_VIEWPORT( H_NORM_BACK, 0.32*RATIO_X, 0.465*RATIO_X,
     * 0.345*RATIO_Y, 0.47*RATIO_Y )
       CALL GKS$SELECT_XFORM( H_NORM_BACK )

       CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

       CALL GKS$SET_WINDOW( H_NORM_FRONT, 75.0, 325.0, 300.0, 750.0 )
       CALL GKS$SET_VIEWPORT( H_NORM_FRONT, 0.6*RATIO_X, 0.8*RATIO_X,
     * 0.15*RATIO_Y, 1.0*RATIO_Y )
       CALL GKS$SELECT_XFORM( H_NORM_FRONT )

       CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

       CALL GKS$CLOSE_SEG()

❽      CALL GKS$SELECT_XFORM( LARGEST_VIEWPORT )

       RETURN
       END
           .
           .
           .
```

The following numbers correspond to the numbers in the previous example:

❶ This code ends program execution if the workstation is not of the type GKS$K_WSCAT_OUTIN. The DEC GKS supported GKS$K_WSCAT_OUTIN workstations illustrate panning and zooming better than the GKS$K_WSCAT_OUTPUT workstations. You can alter this line of code if you want to see the effects on a particular GKS$K_WSCAT_OUTPUT workstation.

❷ The call to GKS$INQ_MAX_DS_SIZE returns the maximum X and Y values of the device coordinate system.

❸ Using the FORTRAN built-in function MAX, this code determines which maximum coordinate value is larger.

❹ This code determines the ratio of the maximum X and Y device coordinate values to the larger of the two. In this manner, you establish one ratio equivalent to 1.0 (the largest NDC value that you can use as a dimension of the workstation window), and you establish another ratio that is less than 1.0. Having established these values, you can define a portion of the default NDC space ([0,1] x [0,1]) that is proportionately equivalent to whichever device coordinate plane you use.

❺ This code establishes LARGEST_VIEWPORT (1) to be the current normalization transformation. This normalization transformation maps the default normalization window to the space on the NDC plane that is proportional to the display surface. Most of the output primitives in the picture use this normalization transformation.

❻ This code sets the workstation transformation so that mapping takes place from the proportionately scaled workstation window to the entire display surface.

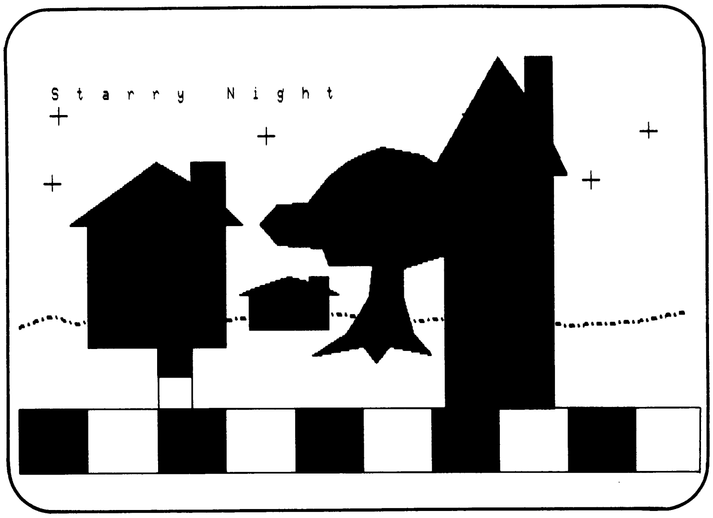
❼ If you define or redefine normalization transformations, you need to make sure that you multiply all NDC coordinate values by the appropriate X or Y ratio. In this manner, you assure that all primitives ultimately appear within the workstation window (which is proportional to the display surface size).

In this application, the normalization windows are not made proportional to the current display surface size. Consequently, the picture's aspect ratio on the display surface may be different than the aspect ratio of the plotted image (you plot on a square normalization window, and then map to a potentially rectangular normalization viewport). The normalization windows used to plot primitives should use proportions required by the needs of your application, not by the needs of any one device coordinate system.

❽ This code reestablishes the normalization transformation LARGEST_VIEWPORT so that all subsequently generated output uses this normalization transformation.

Figure 4–11 illustrates the effect of the previous code example on the VT241. Remember that the picture fills the device coordinate range of any device and may appear differently on various workstations.

Figure 4–11: Using the Entire Display Surface—VT241



ZK-5199-86

4.3.3 Zooming In and Out of a Picture

One of the most useful graphical effects that you can achieve using the workstation transformations is the *zooming* effect. By zooming in and out of a picture, you can give the effect of movement and distance. For instance, you can alter the code in subroutine DRAW_PICTURE in Example 3–2 to give the effect of the user moving towards the house in the distance (closest to the horizon).

To see how to zoom in and out of a picture, review the following code.

```
C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
          .
          .
          .
C     Find out if workstation transformations require implicit
C     regenerations, or if the change is made immediately...
❶     CALL GKS$INQ_DYN_MOD_WS_ATTB( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * WS_XFORMS )
```

```
❷          CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
           CALL ZOOM_PICTURE( WS_ID, WS_XFORMS, RATIO_X, RATIO_Y )

           RETURN
           END

     C     ************************************************************
     C     Zoom in on the picture...
❸          SUBROUTINE ZOOM_PICTURE( WS_ID, WS_XFORMS, RATIO_X, RATIO_Y )

           IMPLICIT NONE
           INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
           INTEGER WS_ID, WS_XFORMS, INCR, ERROR_STATUS, DUMMY_INTEGER,
          * NEW_FRAME_FLAG

           REAL RATIO_X, RATIO_Y, START_X, START_Y, MAX_X, MAX_Y

           DATA START_X / 0.0 /, START_Y / 0.0 /

     C     Use local variables MAX_X and MAX_Y...
           MAX_X = RATIO_X
           MAX_Y = RATIO_Y

❹          DO 200 INCR = 1, 3, 1

           MAX_Y = MAX_Y - (MAX_Y * 0.12 )
           MAX_X = MAX_X - ( MAX_X * 0.12 )
           START_X = START_X + ( MAX_X * 0.12 )
           START_Y = START_Y + ( MAX_Y * 0.12 )

❺          CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
          * START_Y, MAX_Y )

           IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
                CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           ENDIF

❻    200   CONTINUE

❼          DO 300 INCR = 1, 3, 1

           MAX_X = MAX_X + ( MAX_X * 0.12 )
           MAX_Y = MAX_Y + ( MAX_Y * 0.12 )
           START_X = START_X - ( MAX_X * 0.12 )
           START_Y = START_Y - ( MAX_Y * 0.12 )

❽          IF ( INCR .EQ. 3 ) THEN
                MAX_X = RATIO_X
                MAX_Y = RATIO_Y
                START_X = 0.0
                START_Y = 0.0
           ENDIF

           CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
          * START_Y, MAX_Y )

           IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
                CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           ENDIF

     300   CONTINUE
```

```
        RETURN
        END
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

❶ This code inquires as to whether or not workstation transformations require implicit regenerations or if the changes are made immediately. You can pass the flag WS_XFORMS to all subroutines that make workstation transformation changes. By comparing the flag to GKS$K_IRG (Implicit Regeneration), you know whether you need to force a regeneration to update the workstation surface.

❷ This code, located within subroutine DRAW_PICTURE, updates the surface of the workstation by releasing all deferred output. The call to GKS$UPDATE_WS does not cause an implicit regeneration.

This code calls the subroutine ZOOM_PICTURE, which zooms into the middle of the picture composed on the NDC plane.

❸ The program passes WS_ID, RATIO_X, and RATIO_Y to ZOOM_PICTURE. RATIO_X and RATIO_Y are the maximum X and Y NDC values of the current workstation window. These values are of the same proportion as the maximum X and Y device coordinate values. To zoom in on a picture, you need to reduce the dimension of the workstation window, zooming in on a particular point (in this code, the middle of the picture), while keeping the workstation viewport the same size.

This code also assigns the values RATIO_X and RATIO_Y to the local variables MAX_X and MAX_Y.

❹ This loop zooms into the middle of the picture in three steps. The code reduces the workstation window maximum X and Y values by twelve percent and increases the starting points by the same percentage.

Keep in mind that in order to map the workstation window onto the entire workstation surface, you must specify a window that has the same proportion as the maximum device coordinate plane. If you do not, DEC GKS does not use the entire workstation viewport to display the picture; DEC GKS uses the largest rectangle *within the current workstation viewport* that maintains the aspect ratio of the workstation window. Consequently, if you reduce the maximum X and Y values by a certain percentage, you need to *increase* the starting points by the same percentage in order to maintain the proportion of the window.

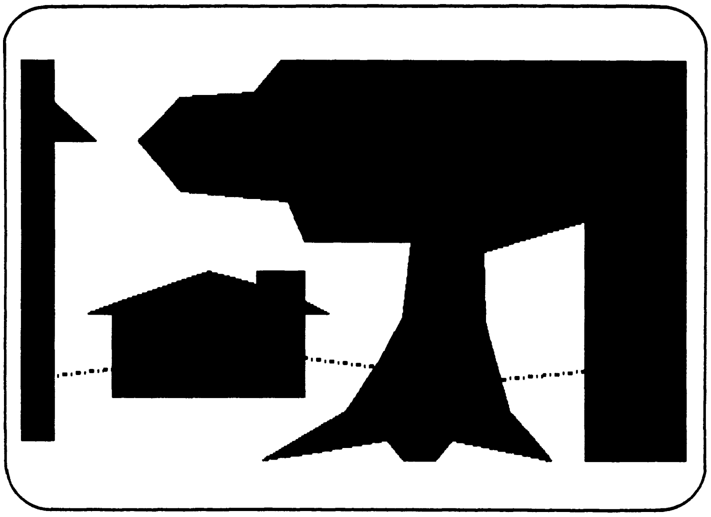
❺ This code sets the workstation window, and if the device handler requires an implicit regeneration to implement the change, generates the implicit regeneration. Any primitives not contained in segments are cleared from the workstation surface.

❻ Figure 4–12 illustrates the surface of the VT241 at this point in the program execution. Keep in mind that the picture may appear differently on other devices.

❼ This code zooms out of the picture in three steps, updating the surface as needed.

❽ This code ensures that the last picture on the workstation surface reflect the workstation window as it was when ZOOM_PICTURE was called.

Figure 4–12: Zooming In on a Picture—VT241



ZK-5148-86

4.3.4 Panning Across a Picture

As described in Section 4.3.3, the most important aspect to zooming in on a picture is maintaining the proportion of the workstation window as you decrease its size. Panning across the picture is easier than zooming since you do not have to alter the size of the window. You only need to move the window from position to position on the NDC space. Panning across a picture is useful when you do not want to show the user the entire picture at one time.

To see how to pan across a picture, review the code added to subroutine ZOOM_PICTURE from Example 3–2.

```
   C     ************************************************************
   C     Zoom in on the picture...
         SUBROUTINE ZOOM_PICTURE( WS_ID, WS_XFORMS, RATIO_X, RATIO_Y )
             .
             .
             .
         READ(5,*)
         CALL PAN_PICTURE( WS_ID, WS_XFORMS, START_X, MAX_X,
        * START_Y, MAX_Y )
         READ(5,*)
             .
             .
             .
   C     ************************************************************
   C     Pan across the picture, first left, then right...
❶        SUBROUTINE PAN_PICTURE( WS_ID, WS_XFORMS, START_X, MAX_X,
        * START_Y, MAX_Y )

         IMPLICIT NONE
         INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
         INTEGER WS_ID, WS_XFORMS, INCR, ERROR_STATUS, DUMMY_INTEGER,
        * NEW_FRAME_FLAG

         REAL MAX_X, MAX_Y, START_X, START_Y

         DO 400 INCR = 1, 3, 1

❷        MAX_X = MAX_X - 0.075
         START_X = START_X - 0.075

         CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
        * START_Y, MAX_Y )

         IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
         ENDIF

❸ 400    CONTINUE

         DO 500 INCR = 1, 3, 1

❹        MAX_X = MAX_X + 0.075
         START_X = START_X + 0.075

         CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
        * START_Y, MAX_Y )

         IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
         ENDIF

  500    CONTINUE

         RETURN
         END
             .
             .
             .
```

The following numbers correspond to the numbers in the previous example:

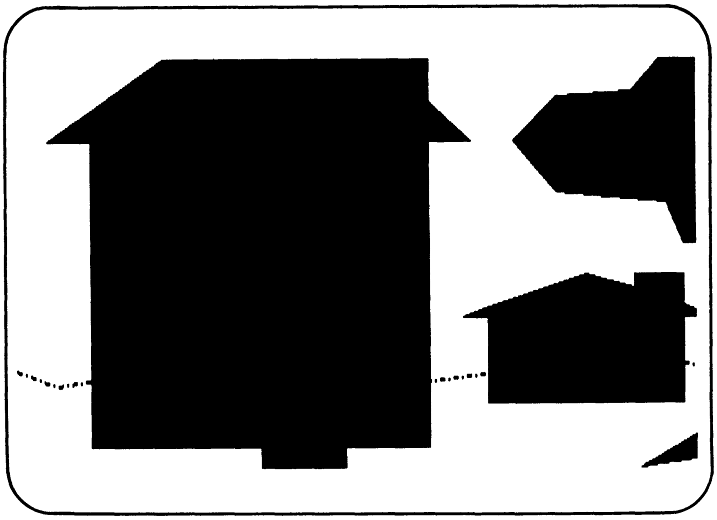
❶ In subroutine ZOOM_PICTURE, you reduce the size of the current workstation window. You pass the reduced window dimensions (START_X, START_Y, MAX_X, and MAX_Y) to PAN_PICTURE.

❷ The loops used in PAN_PICTURE are identical to the loops used in ZOOM_PICTURE except that this code subtracts three quarters of an NDC point from both X workstation window values. In effect, this shifts the workstation window further left within the NDC space.

❸ Figure 4–13 illustrates the surface of the VT241 at this point in the program execution. Keep in mind that the picture may appear differently on other devices.

❹ This code shifts the workstation window back to the right within the NDC space. Consequently, the position of the workstation window within NDC space is the same as it was when you called PAN_PICTURE.

Figure 4–13: Panning Across a Picture—VT241


ZK-5207-86

4.3.5 Using a Smaller Portion of the Workstation Surface

The theory behind using a smaller portion of the workstation surface is the same as the theory behind using the entire device coordinate plane. As long as the workstation window and viewport are proportionately equivalent, DEC GKS maps the entire window to the entire viewport space.

To see how to "shrink" the picture on the surface of the workstation, review the code added to the subroutine DRAW_PICTURE in Example 3–2.

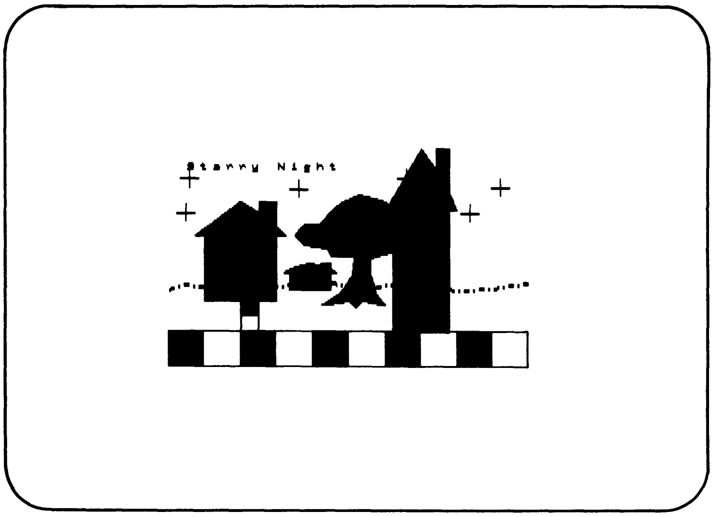
```
C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
          .
          .
          .
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)
      CALL ZOOM_PICTURE( WS_ID, WS_XFORMS, RATIO_X, RATIO_Y )
      READ(5,*)
      CALL SHRINK_PICTURE( WS_ID, WS_XFORMS, DISPLAY_X,
     * DISPLAY_Y )
          .
          .
          .
C     ************************************************************
C     Shrink and then expand the portion of the display surface used...
      SUBROUTINE SHRINK_PICTURE( WS_ID, WS_XFORMS, DISPLAY_X,
     * DISPLAY_Y )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_XFORMS, INCR, ERROR_STATUS, DUMMY_INTEGER,
     * NEW_FRAME_FLAG

      REAL DISPLAY_X, DISPLAY_Y, START_X, START_Y, MAX_X, MAX_Y

      DATA START_X / 0.0 /, START_Y / 0.0 /

      MAX_X = DISPLAY_X
      MAX_Y = DISPLAY_Y

❶     MAX_Y = MAX_Y - (MAX_Y * 0.3 )
      MAX_X = MAX_X - ( MAX_X * 0.3 )
      START_X = START_X + ( MAX_X * 0.3 )
      START_Y = START_Y + ( MAX_Y * 0.3 )

      CALL GKS$SET_WS_VIEWPORT( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

❷     MAX_Y = DISPLAY_Y
      MAX_X = DISPLAY_X
      START_X = 0.0
      START_Y = 0.0

      CALL GKS$SET_WS_VIEWPORT( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

      RETURN
      END
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

1. This code proportionately reduces the size of the workstation viewport in the same way that you reduce the workstation window when zooming in on a picture. In this manner, you can map the entire picture to the entire range of the reduced workstation viewport.

   Figure 4–14 illustrates the surface of the VT241 at this point in the program execution. Keep in mind that the picture may appear differently on other devices.
2. This code restores the workstation viewport to the entire device coordinate range.

Figure 4–14: Reducing the Workstation Surface Area—VT241



4.4 Program Example Used in this Chapter

Example 4–1 presents all of the changes that you need to make to Example 3–2 in order to follow the code examples in this chapter.

Example 4-1: Using the DEC GKS Transformations

```
           .
           .
           .
C      ************************************************************
C      Draw the picture, and place each primitive in a segment...
       SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

       IMPLICIT NONE
       INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
       INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
           .
           .
           .
     * COLOR_FLAG, NUM_INDEXES, THREE, BW_NUM_PTS, H_NORM_LEFT,
     * LARGEST_VIEWPORT, H_NORM_FRONT, H_NORM_BACK, CATEGORY,
     * WS_XFORMS

       REAL TEXT_START_X, TEXT_START_Y, STARS_X_VALUES( 6 ),
           .
           .
           .
     * DISPLAY_X, DISPLAY_Y, MAX_COORD, RATIO_X, RATIO_Y

       DATA TEXT_START_X / 0.05 /,
           .
           .
           .
     * THREE / 3 /, BW_NUM_PTS / 9 /, H_NORM_LEFT / 1 /,
     * LARGEST_VIEWPORT / 4 /, H_NORM_BACK / 2 /,
     * H_NORM_FRONT / 3 /

       DATA HOUSE_X / 100.0, 300.0, 300.0, 325.00, 300.0, 300.0,
     * 250.0, 250.0, 200.0, 75.0, 100.0, 100.0 /
       DATA HOUSE_Y / 300.0, 300.0, 600.0, 600.0, 640.0, 750.0,
     * 750.0, 700.0, 750.0, 600.0, 600.0, 3.00 /
           .
           .
           .
       CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
     * ERROR_STATUS, CATEGORY )

C      Only allow execution for terminal screens...
       IF ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ) THEN
            WRITE(6,*)
     *        'The specified workstation type is not OUTIN.'
            STOP
       ENDIF

C      Obtain the maximum X and Y device coordinate values...
       CALL GKS$INQ_MAX_DS_SIZE( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DISPLAY_X, DISPLAY_Y, DUMMY_INTEGER,
     * DUMMY_INTEGER )
```

(continued on next page)

```
C     Find out which maximum value is largest...
      MAX_COORD = MAX( DISPLAY_X, DISPLAY_Y )

C     Depending on which is larger, establish the X to Y (or Y to X)
C     ratio.  To establish a portion of the NDC space that is the same
C     proportion as the display coordinate system, use the values 1.0 and
C     the Y/X---X/Y ratio as the maximum normalization viewport values.
      IF (( DISPLAY_X / MAX_COORD ) .EQ. 1.0 ) THEN
            RATIO_X = 1.0
            RATIO_Y = DISPLAY_Y / MAX_COORD
      ELSE
            RATIO_X = DISPLAY_X / MAX_COORD
            RATIO_Y = 1.0
      ENDIF

C     Establish a normalization viewport that is proportionate to the
C     device coordinate plane.
      CALL GKS$SET_VIEWPORT( LARGEST_VIEWPORT, 0.0, RATIO_X, 0.0,
     * RATIO_Y )
      CALL GKS$SELECT_XFORM( LARGEST_VIEWPORT )

C     Establish the same portion of the NDC space to be the workstation
C     window, and establish the entire device coordinate plane as the
C     workstation viewport.
      CALL GKS$SET_WS_WINDOW( WS_ID, 0.0, RATIO_X, 0.0, RATIO_Y )
      CALL GKS$SET_WS_VIEWPORT( WS_ID, 0.0, DISPLAY_X, 0.0,
     * DISPLAY_Y )
          .
          .
          .

      CALL GKS$CREATE_SEG( HOUSE )
C     Only change the color index if working with a color workstation
C     (or a VT125/240 or a VAXstation).
      IF ( NUM_INDEXES .GE. THREE ) THEN
           CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF

      CALL GKS$SET_WINDOW( H_NORM_LEFT, 75.0, 325.0, 300.0, 750.0 )

C     When working with NDC points, you need to translate the point by
C     multiplying by the appropriate ratio.  In this way, the whole
C     picture is mapped to the proportionate viewport.
      CALL GKS$SET_VIEWPORT( H_NORM_LEFT, 0.075*RATIO_X, 0.325*RATIO_X,
     * 0.3*RATIO_Y, 0.75*RATIO_Y )
      CALL GKS$SELECT_XFORM( H_NORM_LEFT )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

      CALL GKS$SET_WINDOW( H_NORM_BACK, 75.0, 325.0, 300.0, 750.0 )
      CALL GKS$SET_VIEWPORT( H_NORM_BACK, 0.32*RATIO_X, 0.465*RATIO_X,
     * 0.345*RATIO_Y, 0.47*RATIO_Y )
      CALL GKS$SELECT_XFORM( H_NORM_BACK )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
```

(continued on next page)

Example 4–1 (Cont.): Using the DEC GKS Transformations

```
      CALL GKS$SET_WINDOW( H_NORM_FRONT, 75.0, 325.0, 300.0, 750.0 )
      CALL GKS$SET_VIEWPORT( H_NORM_FRONT, 0.6*RATIO_X, 0.8*RATIO_X,
     * 0.15*RATIO_Y, 1.0*RATIO_Y )
      CALL GKS$SELECT_XFORM( H_NORM_FRONT )

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )

      CALL GKS$CLOSE_SEG()

      CALL GKS$SELECT_XFORM( LARGEST_VIEWPORT )

C     Find out if workstation transformations require implicit
C     regenerations, or if the change is made immediately...
      CALL GKS$INQ_DYN_MOD_WS_ATTB( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * WS_XFORMS )


C     Flush deferred output. Type RETURN when you are finished viewing
C     the picture...
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

C     Zoom in and out, and then reduce the portion of the surface used
C     for picture generation...
      CALL ZOOM_PICTURE( WS_ID, WS_XFORMS, RATIO_X, RATIO_Y )
      READ(5,*)
      CALL SHRINK_PICTURE( WS_ID, WS_XFORMS, DISPLAY_X,
     * DISPLAY_Y )

      RETURN
      END

C     ************************************************************
C     From this point forward, all code is additional code that you
C     need to add to the "Starry Night" program.
C     ************************************************************

C     ************************************************************
C     Zoom in on the picture...
      SUBROUTINE ZOOM_PICTURE( WS_ID, WS_XFORMS, RATIO_X, RATIO_Y )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_XFORMS, INCR, ERROR_STATUS, DUMMY_INTEGER,
     * NEW_FRAME_FLAG

      REAL RATIO_X, RATIO_Y, START_X, START_Y, MAX_X, MAX_Y

      DATA START_X / 0.0 /, START_Y / 0.0 /

C     Use local variables MAX_X and MAX_Y...
      MAX_X = RATIO_X
      MAX_Y = RATIO_Y
```

(continued on next page)

```
C     Zoom in on 3 increments...
      DO 200 INCR = 1, 3, 1

C     Reduce the workstation window by 12%...
      MAX_Y = MAX_Y - (MAX_Y * 0.12 )
      MAX_X = MAX_X - ( MAX_X * 0.12 )
      START_X = START_X + ( MAX_X * 0.12 )
      START_Y = START_Y + ( MAX_Y * 0.12 )

C     Establish the new workstation window...
      CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

C     If regeneration is needed, do it...
      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

200   CONTINUE

C     Now that the current workstation window is a small portion of
C     the entire picture, pan across the NDC plane...
      READ(5,*)
      CALL PAN_PICTURE( WS_ID, WS_XFORMS, START_X, MAX_X,
     * START_Y, MAX_Y )
      READ(5,*)

C     In 3 increments, zoom out...
      DO 300 INCR = 1, 3, 1

      MAX_X = MAX_X + ( MAX_X * 0.12 )
      MAX_Y = MAX_Y + ( MAX_Y * 0.12 )
      START_X = START_X - ( MAX_X * 0.12 )
      START_Y = START_Y - ( MAX_Y * 0.12 )

C     If it is the last increment, do not rely on the calculations.
C     Just reset the workstation window to be the proportionate window
C     that was in place when you called ZOOM_PICTURE...
      IF ( INCR .EQ. 3 ) THEN
           MAX_X = RATIO_X
           MAX_Y = RATIO_Y
           START_X = 0.0
           START_Y = 0.0
      ENDIF

      CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

C     If regeneration is needed, do it...
      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

300   CONTINUE
```

(continued on next page)

```
      RETURN
      END

C     ************************************************************
C     Pan across the picture, first left, then right...
      SUBROUTINE PAN_PICTURE( WS_ID, WS_XFORMS, START_X, MAX_X,
     * START_Y, MAX_Y )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_XFORMS, INCR, ERROR_STATUS, DUMMY_INTEGER,
     * NEW_FRAME_FLAG

      REAL MAX_X, MAX_Y, START_X, START_Y

C     In 3 increments, pan to the left...
      DO 400 INCR = 1, 3, 1

      MAX_X = MAX_X - 0.075
      START_X = START_X - 0.075

      CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

C     If regeneration is needed, do it...
      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
          CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

400   CONTINUE

C     In 3 increments, pan to the right...
      DO 500 INCR = 1, 3, 1

      MAX_X = MAX_X + 0.075
      START_X = START_X + 0.075

      CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

C     If regeneration is needed, do it...
      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
          CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

500   CONTINUE

      RETURN
      END

C     ************************************************************
C     Shrink and then expand the portion of the display surface used...
      SUBROUTINE SHRINK_PICTURE( WS_ID, WS_XFORMS, DISPLAY_X,
     * DISPLAY_Y )
```

(continued on next page)

Example 4–1 (Cont.): Using the DEC GKS Transformations

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_XFORMS, INCR, ERROR_STATUS, DUMMY_INTEGER,
     * NEW_FRAME_FLAG

      REAL DISPLAY_X, DISPLAY_Y, START_X, START_Y, MAX_X, MAX_Y

      DATA START_X / 0.0 /, START_Y / 0.0 /

C     Use local variables...
      MAX_X = DISPLAY_X
      MAX_Y = DISPLAY_Y


      MAX_Y = MAX_Y - (MAX_Y * 0.3 )
      MAX_X = MAX_X - ( MAX_X * 0.3 )
      START_X = START_X + ( MAX_X * 0.3 )
      START_Y = START_Y + ( MAX_Y * 0.3 )

      CALL GKS$SET_WS_VIEWPORT( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

C     If regeneration is needed, do it...
      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

C     Set the workstation viewport to the entire device coordinate space.
      MAX_Y = DISPLAY_Y
      MAX_X = DISPLAY_X
      START_X = 0.0
      START_Y = 0.0

      CALL GKS$SET_WS_VIEWPORT( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

C     If regeneration is needed, do it...
      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

      RETURN
      END
          .
          .
          .
```

Chapter 5

Generating Output

This chapter provides an overview of issues related to output generation. For instance, you can generate the same polyline in the same position several times during program execution, but depending on certain state list entries, that line can have a different appearance for each generation. Also, whether or not the line is part of a segment may affect its appearance on the workstation surface.

This chapter discusses the following concepts in detail:

- Geometric and nongeometric output attributes
- Individual and bundled output attributes
- Aspect Source Flags
- Text attributes
- Segment transformations and clipping
- Segment attributes
- Surface update and regeneration

NOTE

Section 5.4 contains the code that you must add to the Starry Night program in Example 3–2 to produce the program example contained in this chapter. You may wish to add this code to the base program so that you can execute the program while reading this chapter. The lines of blue code in the example signify the new code that you need to add to Example 3–2.

5.1 Output Attributes

An output *attribute* is an aspect of an output primitive that determines how the primitive appears on the surface of the workstation. For instance, when calling the function GKS$POLYLINE, DEC GKS must know whether to represent the line as a solid line, a dashed line, a dotted line, or a dashed and dotted line. The current attribute settings determine how DEC GKS represents an output primitive.

When generating lines on the workstation surface, you can also alter the line width and the line color. Most of the other output primitives have alterable attributes that affect the appearance of the corresponding primitive (generalized drawing primitives and cell arrays do not). For instance, when generating text, you can alter the character spacing or the text color.

DEC GKS stores the default output attribute values for a given workstation in the workstation description table. DEC GKS stores the current values in the GKS and workstation state lists. You can use the inquiry functions to obtain information about the default or current attribute settings.

All primitives have a special type of attribute called the *pick identifier*. This attribute does not affect how the primitive appears on the surface of the workstation upon primitive generation. You can use the pick identifier as an aid during pick input. Chapter 6, Requesting Input, discusses the use of the pick identifier during pick input.

5.1.1 Geometric and Nongeometric Attributes

Of the attributes that affect the appearance of the primitive at the time of generation, there are *geometric* attributes and *nongeometric* attributes.

Nongeometric attributes affect the style and the pattern of the output primitives (such as polyline color, text spacing, and fill area internal style). Since many of the nongeometric attributes involve scale factors and *nominal* sizes, the effects of these attributes are device dependent. Most output primitives have nongeometric attributes (cell arrays and GDPs do not).

Nominal sizes are the default sizes of markers and line widths as defined by a graphics handler. In most cases the nominal size is also the smallest size that a workstation can produce, but not always. To reset a marker size or polyline width, DEC GKS multiplies the scale factor values by the nominal size. The default value for a scale factor is 1.0 (the nominal size multiplied by the value 1.0, producing no change in the default size).

When you alter the nongeometric attributes, you could possibly specify a scale factor creating a size that is larger than the workstation's largest size. If this happens, the graphics handler ignores the scale specification and uses the largest size defined by the handler. You can obtain the smallest, largest, and nominal sizes by calling one of the functions GKS$INQ_PREDEF_primitive_FAC.

Geometric attributes affect the size or positioning of text and fill area primitives (such as character height, character path, and pattern size). Fill area and text are the only two output primitives that have changeable geometric attributes. The geometric attributes are specified in world coordinate units. Since the world coordinates are device independent, the geometric attributes are device independent.

For a list of the nongeometric and geometric attributes, refer to Chapter 5, Output Attribute Functions, in the *DEC GKS Reference Manual.*

5.1.2 Individual and Bundled Attributes

In the previous chapters in this manual, the code examples contain code that changes attribute values as follows:

```
          .
          .
          .
      CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS )
      CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
      CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED )

C     Obtain the workstation type.
      CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
     * WS_TYPE, DUMMY_INTEGER )

C     Make sure that you don't ask for a line wider than the
C     workstation's widest line.
      CALL GKS$INQ_PLINE_FAC( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, %DESCR( DUMMY_INT_ARRAY ), DUMMY_INTEGER,
     * NOM_WIDTH, DUMMY_REAL, MAX_WIDTH, DUMMY_INTEGER,
     * DUMMY_INTEGER )

      DO WHILE (( WIDER * NOM_WIDTH ) .GT. MAX_WIDTH )
         WIDER = WIDER - 0.1
      ENDDO

      CALL GKS$SET_PLINE_LINEWIDTH( WIDER )
          .
          .
          .
```

In this way, the program alters the following output attributes:

- Character height (geometric attribute)
- Marker type (nongeometric attribute)
- Fill area interior style (nongeometric attribute)
- Line type (nongeometric attribute)
- Line width (nongeometric attribute)

The only nongeometric attribute change that involves scaling is the change to the line width (the call to GKS$SET_PLINE_LINEWIDTH). If you request a size that is too large, DEC GKS uses the largest size supported by the workstation without generating an error. However, if you specify the value 0.0 or a negative scale value, DEC GKS generates an error.

In all of the attribute changes presented so far in this manual, the programs changed *individual* attribute settings. When you need to change one attribute value, you called a single output attribute function that alters that setting. The DEC GKS state list stores the current individual attribute setting for each attribute (the entry for current *line type* can be GKS$K_LINETYPE_SOLID (1), the entry for current *line width scale factor* can be 2.0, and so forth).

By default, DEC GKS checks the current individual attributes settings before generating the requested output primitive. The individual settings are device independent; changing the individual line type to GKS$K_LINETYPE_SOLID causes the next generated line to be solid no matter on which device DEC GKS generates the primitive.

When altering the geometric output attributes of a primitive, you can only alter them individually. When altering the nongeometric attributes of a primitive, you have the option of changing the values individually, or changing them in a *bundle*.

Bundles are groupings of nongeometric attribute settings. Each workstation predefines a bundle table for each primitive that has nongeometric attributes. Each table contains all the defined bundle groups for its primitive. To access a given bundle, you must specify an integer index value that refers to one group of bundled settings within the table.

For example, a graphics handler can predefine the index value 1 to represent a solid green line with a width scale factor of 1.0 (the nominal width). A portion of a default polyline bundle table can be as follows:

Index	Line Type	Line Width	Color Index	Description
1	1	1.0	1	Solid green
2	1	1.0	2	Solid red
3	1	1.0	3	Solid blue
	.			
	.			
	.			
26	4	1.0	5	Dashed-dotted magenta
27	4	1.0	6	Dashed-dotted yellow
28	4	1.0	7	Dashed-dotted black

In previous programs, you specify numeric values to represent colors. Those numeric values are color index values that point into a color table. For instance, the default LCG01 color index table is as follows:

Index	Color	Red Intensity	Green Intensity	Blue Intensity
0	White	1.0	1.0	1.0
1	Green	0.0	1.0	0.0
2	Red	1.0	0.0	0.0
3	Blue	0.0	0.0	1.0
4	Cyan	0.0	1.0	1.0
5	Magenta	1.0	0.0	1.0
6	Yellow	1.0	1.0	0.0
7	Black	0.0	0.0	0.0

Appendix H, DEC GKS Color Chart, in the *DEC GKS Reference Manual,* provides a set of red, green, and blue intensities to use as a guide when defining color index values.

Each color index points to a set of red, green, and blue intensity values that actually determine the color represented by the index value. As with all bundle tables, you have the option of using the predefined representations of index values, you may be able to define representations for additional index values, or you can redefine an existing representation of index values.

If you define or change a bundle index value used by a primitive already generated on the workstation surface, you may or may not cause an implicit regeneration of the surface. If an implicit regeneration occurs, you lose all output not contained in segments. Section 5.1.4 discusses representation changes in detail.

DEC GKS stores the current bundle table representations in the workstation state list. Since each graphics handler can predefine different bundle tables with a different number of index values, the bundled attributes are device dependent.

If you use bundled attributes, you save time. You do not have to set the nongeometric attributes individually (with separate function calls). With a single function call, you can set a group of attributes.

DEC GKS binds either the GKS$K_ASF_INDIVIDUAL or the GKS$K_ASF_BUNDLED attributes to a primitive at the time of output generation. If you specified a primitive's attributes individually, then you cannot alter its appearance in subsequent portions of the program. If you specified a primitive's attributes using a bundle index, and if the primitive is in a segment (or if your workstation supports dynamic attribute changes), then you can alter the primitive by redefining representation of its bundle index. You change the representation of a bundle index by calling one of the GKS$SET_primitive_REP functions.

Before output generation, DEC GKS must determine whether to use individual or bundled attributes for a specified primitive. To determine which type of attribute to use, DEC GKS checks the attribute's *aspect source flag*. Section 5.1.3 describes aspect source flags in detail.

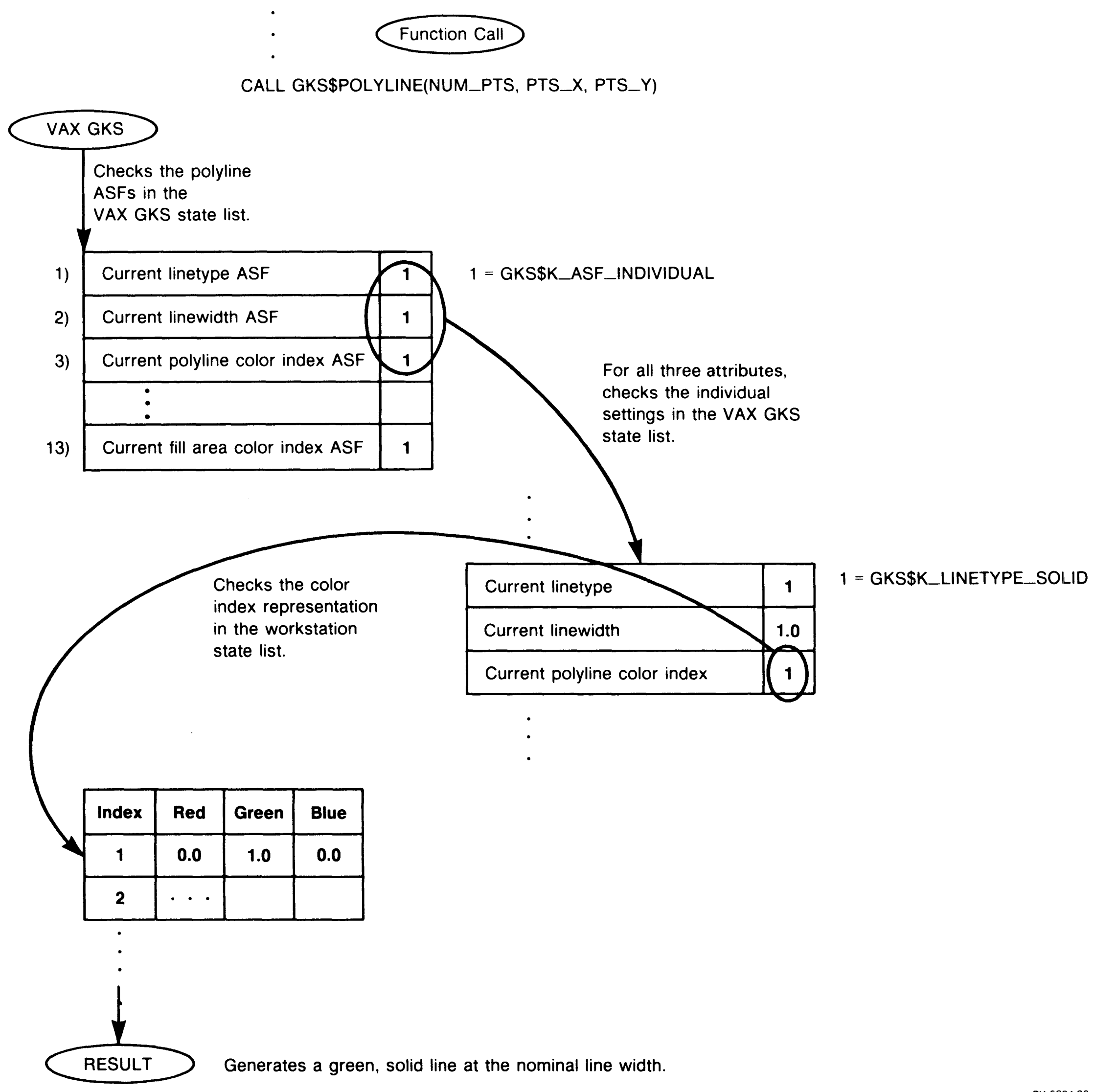
5.1.3 Aspect Source Flags

When you call an output function, DEC GKS must determine whether to use the current individual attributes associated with a primitive, whether to use all of the attribute values associated with the current bundle index, or whether to use some individual settings and some bundled settings.

To determine which setting to use for which nongeometric attribute, DEC GKS checks the current value of the aspect source flag (ASF). The aspect source flags are elements of the DEC GKS state list that contain either the value GKS$K_ASF_BUNDLED (0) or the value GKS$K_ASF_INDIVIDUAL (1). If the attribute ASF contains GKS$K_ASF_INDIVIDUAL (the default situation), DEC GKS uses the individual setting for that particular attribute. If the ASF contains GKS$K_ASF_BUNDLED, DEC GKS determines the current bundle index, enters the appropriate bundle table, obtains the value for the particular setting, and uses that setting during output generation.

Figure 5–1 illustrates action taken by DEC GKS for the default polyline aspect source flag settings.

Figure 5–1: Default Aspect Source Flag Settings


Function Call
CALL GKS$POLYLINE(NUM_PTS, PTS_X, PTS_Y)
VAX GKS
Checks the polyline ASFs in the VAX GKS state list.
1) Current linetype ASF 1
2) Current linewidth ASF 1
3) Current polyline color index ASF 1
13) Current fill area color index ASF 1
1 = GKS$K_ASF_INDIVIDUAL
For all three attributes, checks the individual settings in the VAX GKS state list.
Current linetype 1
Current linewidth 1.0
Current polyline color index 1
1 = GKS$K_LINETYPE_SOLID
Checks the color index representation in the workstation state list.
Index Red Green Blue
1 0.0 1.0 0.0
2 . . .
RESULT
Generates a green, solid line at the nominal line width.
ZK-5224-86

Since there are a total of thirteen nongeometric attributes, DEC GKS requires that you define a thirteen-element integer array. Each element of the array corresponds to a single aspect source flag. Depending on the contents of the array elements, DEC GKS uses individual or bundled attribute values during the next call to generate output. You must structure the thirteen-element integer array as follows:

Element	Nongeometric Attribute
1	line type
2	line width scale factor
3	polyline color index
4	marker type
5	marker size scale factor
6	polymarker color index
7	text font and precision
8	character expansion factor
9	character spacing
10	text color index
11	fill area interior style
12	fill area style index
13	fill area color index

After you define your array with each element containing either GKS$K_ASF_BUNDLED (0) or GKS$K_ASF_INDIVIDUAL (1), you pass the array to the functions GKS$SET_ASF as follows:

```
          .
          .
          .
CALL GKS$SET_ASF( FLAG_ARRAY )
          .
          .
          .
```

When working with bundled attribute values, you can either allow the device handler to use the default index value or you can specify a new index value. For instance, you specify a new polyline bundle index value to the device handlers as follows.

```
        .
        .
        .
C  Setting a polyline index value...
   CALL GKS$SET_PLINE_INDEX( 4 )
        .
        .
        .
```

Figure 5–2 illustrates what happens if you pass GKS$K_ASF_BUNDLED in the first three elements of the integer array (which correspond to line type, line width, and line color).

DEC GKS treats each nongeometric attribute value separately according to its current ASF. Depending on the ASF value, DEC GKS can use individual settings or bundled settings for the generation of an output primitive. Figure 5–3 illustrates what happens if you define GKS$K_ASF_BUNDLED for some polyline ASF flags and GKS$K_ASF_INDIVIDUAL for other polyline flags. If an attribute's ASF is set to GKS$K_ASF_BUNDLED and you have not set a bundle index, DEC GKS uses bundle index 1 by default.

Figure 5–2: Specifying Bundled Aspect Source Flag Settings

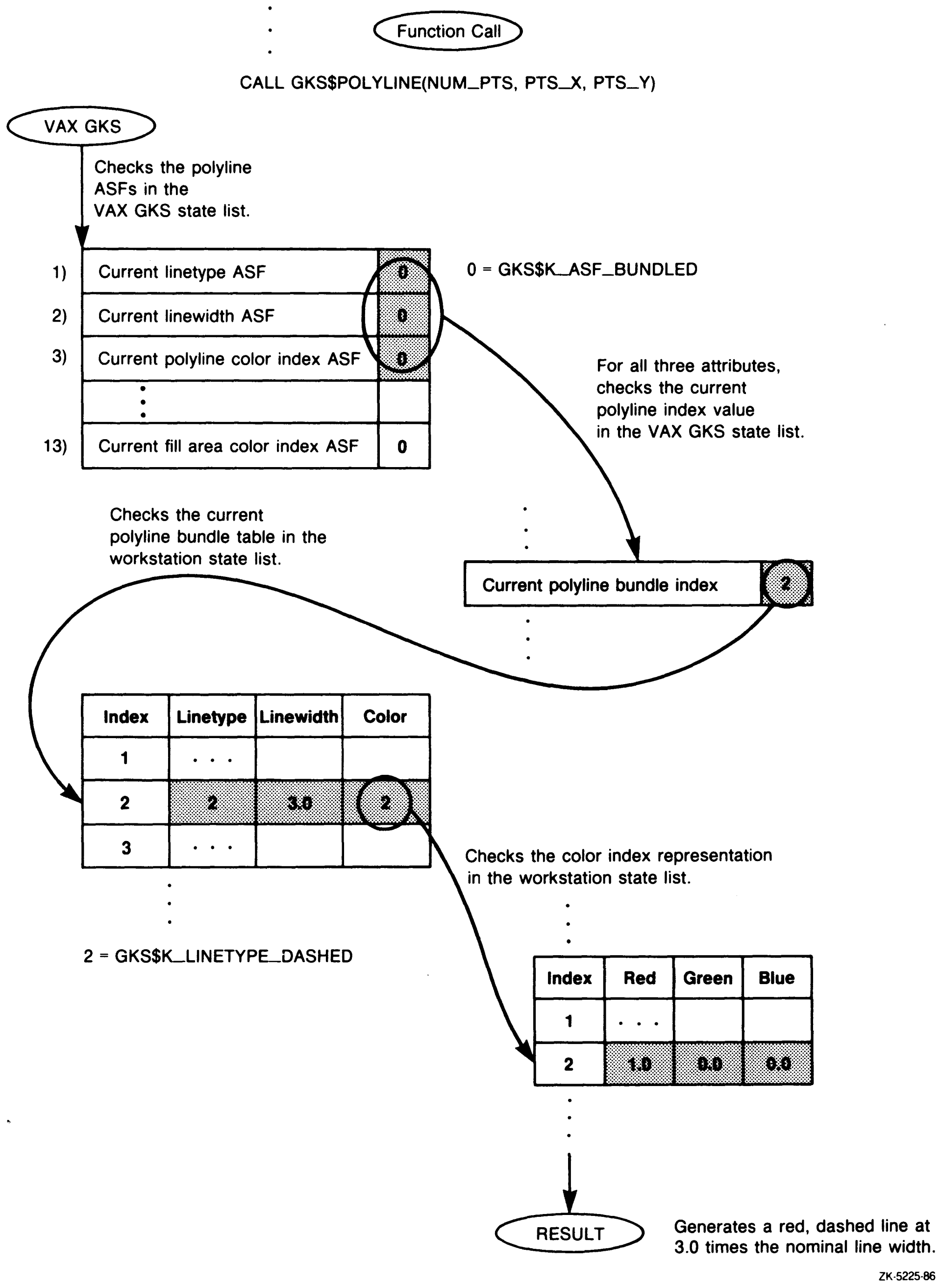
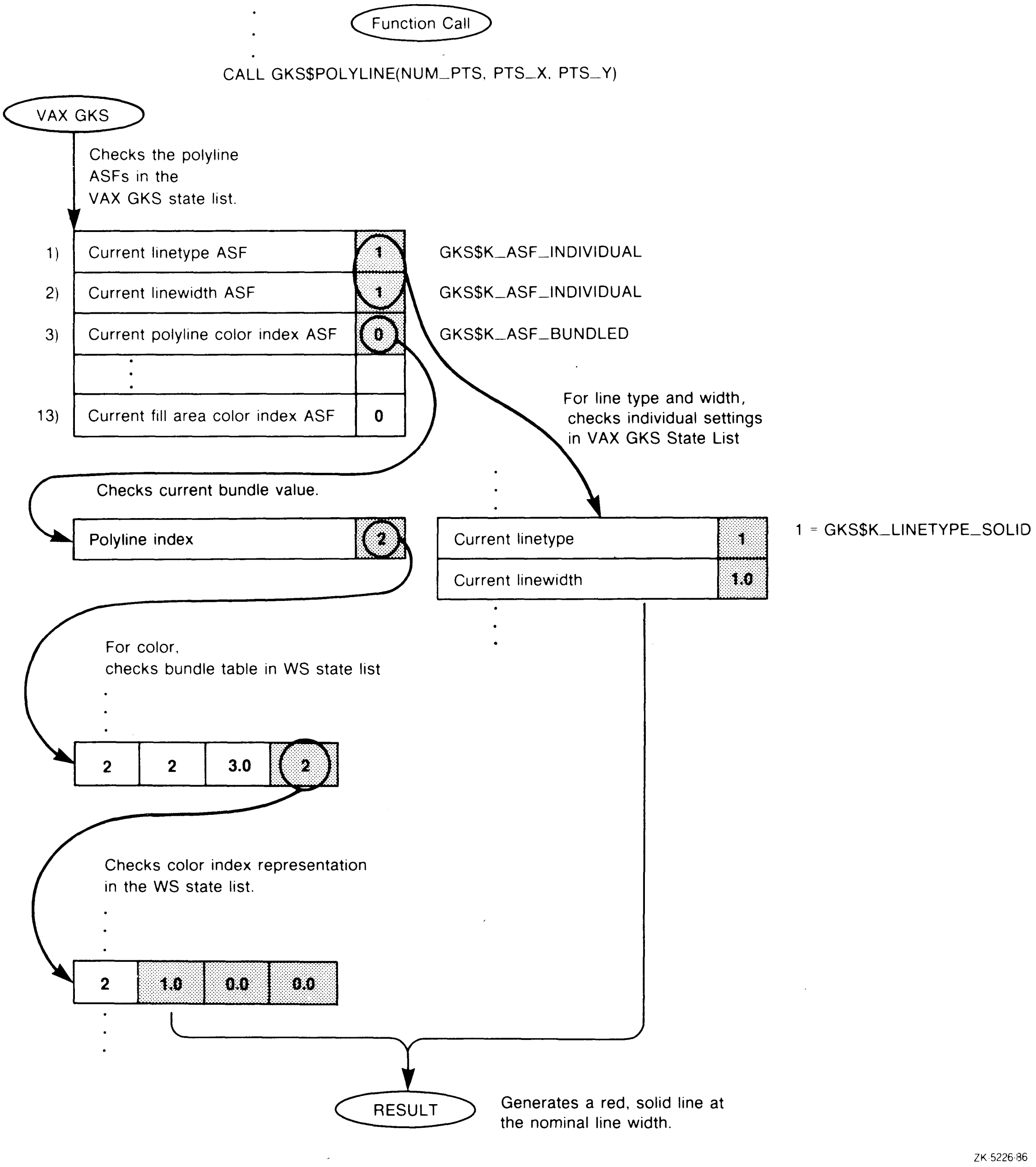

Function Call
CALL GKS$POLYLINE(NUM_PTS, PTS_X, PTS_Y)
VAX GKS
Checks the polyline ASFs in the VAX GKS state list.
1) Current linetype ASF 0
2) Current linewidth ASF 0
3) Current polyline color index ASF 0
13) Current fill area color index ASF 0
0 = GKS$K_ASF_BUNDLED
For all three attributes, checks the current polyline index value in the VAX GKS state list.
Current polyline bundle index 2
Checks the current polyline bundle table in the workstation state list.
Index
Linetype
Linewidth
Color
1 . . .
2 2 3.0 2
3 . . .
2 = GKS$K_LINETYPE_DASHED
Checks the color index representation in the workstation state list.
Index
Red
Green
Blue
1 . . .
2 1.0 0.0 0.0
RESULT
Generates a red, dashed line at 3.0 times the nominal line width.
ZK-5225-86

Figure 5–3: Specifying Bundled and Individual ASFs


Function Call
CALL GKS$POLYLINE(NUM_PTS, PTS_X, PTS_Y)
VAX GKS
Checks the polyline ASFs in the VAX GKS state list.
1) Current linetype ASF 1 GKS$K_ASF_INDIVIDUAL
2) Current linewidth ASF 1 GKS$K_ASF_INDIVIDUAL
3) Current polyline color index ASF 0 GKS$K_ASF_BUNDLED
13) Current fill area color index ASF 0
For line type and width, checks individual settings in VAX GKS State List
Checks current bundle value.
Polyline index 2
Current linetype 1
Current linewidth 1.0
1 = GKS$K_LINETYPE_SOLID
For color, checks bundle table in WS state list
2 2 3.0 2
Checks color index representation in the WS state list.
2 1.0 0.0 0.0
RESULT
Generates a red, solid line at the nominal line width.
ZK-5226-86

5.1.4 Bundle Index Representations

The separate nongeometric attribute settings that comprise an attribute bundle are the *representation* of the bundle index. For instance, the representation of polyline bundle index 3 for the LCG01 includes a solid line type, a nominal line width, and a color index of 3. By default, the color representation for index number 3 is the color blue. The following table illustrates the representation of bundle index 3:

Index	Line Type	Line Width	Color Index	Description
	.			
	.			
	.			
3	1	1.0	3	Solid blue
	.			
	.			
	.			

A graphics handler supports a given number of bundle representations. Of that maximum number of bundle indexes, the graphics handler can predefine any number of them.

Bundle representations are not static. You can change the attribute settings associated with a predefined bundle index, or you can establish a new representation for a supported index value that was not predefined. To establish or change a representation associated with a given index value, you use the SET REPRESENTATION functions (GKSSET_PLINE_REP, GKSSET_PMARK_REP, and so forth).

For instance, the first three index values in the VT125/240 fill area bundle table are as follows:

Index	Interior Style	Style Index	Color Index	Description
1	1	NA	1	Green solid fill
2	1	NA	2	Red solid fill
3	1	NA	3	Blue solid fill
	.			
	.			
	.			

NOTE

DEC GKS only uses the interior fill style index for interior styles GKS$K_INTSTYLE_PATTERN and GKS$K_INTSTYLE_HATCH. For solid fill interior styles (GKS$K_INTSTYLE_SOLID), the interior fill style is not applicable.

In the previous code examples in this chapter, the examples represent the tree and house as solid fill areas of two distinct colors. The following example uses individual and bundled nongeometric fill area attributes:

```
            .
            .
            .
C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, WISS, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
            .
            .
            .
      CALL GKS$CREATE_SEG( STARS )
      CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES,
     * STARS_Y_VALUES )
      CALL GKS$CLOSE_SEG()

C     Use a bundle index for some of the fill area attributes.
      CALL FILL_ATTS( WS_ID, WS_TYPE )

      CALL GKS$CREATE_SEG( TREE )
      CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )
      CALL GKS$CLOSE_SEG()
            .
            .
            .
C     ************************************************************
C     Use bundled attribute values for most of the fill attributes...
      SUBROUTINE FILL_ATTS( WS_ID, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, FILL_INDEX, STYLE_INDEX, ASP_SOURCE_FLAGS( 13 ),
     * RED, WS_TYPE, DUMMY_INT_ARRAY( 150 ), DUMMY_INTEGER,
     * ERROR_STATUS, INQUIRY_OKAY, NUM_HATCH, NUM_FILL_INDEXES

      DATA FILL_INDEX / 2 /, STYLE_INDEX / -5 /, RED / 2 /,
     * INQUIRY_OKAY / 0 /

C     Set all attributes to be individual (the default)...
❶     DATA ASP_SOURCE_FLAGS / 1,1,1,1,1,1,1,1,1,1,1,1,1 /

C     Make sure that the fill index number 2 and the hatch
C     style -5 are valid.
❷     CALL GKS$INQ_FILL_FAC( WS_TYPE, ERROR_STATUS, DUMMY_INTEGER,
     * %DESCR( DUMMY_INT_ARRAY ), NUM_HATCH,
     * %DESCR( DUMMY_INT_ARRAY ), NUM_FILL_INDEXES, DUMMY_INTEGER )
```

```
C     If the workstation does not have enough indexes or hatch styles,
C     signal the errors.
      IF (( NUM_HATCH .LT. 5 ) .OR.
     *   (( NUM_FILL_INDEXES .LT. FILL_INDEX ) .OR.
     *   ( ERROR_STATUS .NE. INQUIRY_OKAY ))) THEN
          WRITE(6,*)
     * 'Fill area facilities not adequate for this program.'
          WRITE(6,*) ERROR_STATUS, NUM_FILL_INDEXES, NUM_HATCH
          GO TO 300
      ENDIF

❸     ASP_SOURCE_FLAGS( 11 ) = GKS$K_ASF_BUNDLED
      ASP_SOURCE_FLAGS( 12 ) = GKS$K_ASF_BUNDLED
      ASP_SOURCE_FLAGS( 13 ) = GKS$K_ASF_INDIVIDUAL
      CALL GKS$SET_ASF( ASP_SOURCE_FLAGS )

C     Set the representation for bundle index 2.
❹     CALL GKS$SET_FILL_REP( WS_ID, FILL_INDEX,
     * GKS$K_INTSTYLE_HATCH, STYLE_INDEX, RED )

C     Set the current fill bundle index.
❺     CALL GKS$SET_FILL_INDEX( FILL_INDEX )

      RETURN
300   STOP
      END
```

The following numbers correspond to the numbers in the previous example:

❶ This code sets all elements of ASP_SOURCE_FLAGS to the value 1 (GKS$K_ASF_INDIVIDUAL), which is the default setting.

❷ This code checks the graphic handlers fill area attribute facilities to see how many fill and hatch indexes are supported. Since this subroutine uses 2 (FILL_INDEX) as the fill area index, and -5 (HATCH_INDEX) as the style index, you need to be sure that the graphics handler supports at least two fill indexes and five hatch styles.

The hatch style index points into the style bundle table. Since all hatch styles are device dependent (the GKS standard does not define standard hatch styles), their index values are negative.

If the device does not support enough fill area or index styles, the program stops execution.

❸ This code changes the elements of ASP_SOURCE_FLAGS that correspond with the nongeometric fill area attributes. Notice that the flags specify GKS$K_ASF_BUNDLED for fill area interior style and style only. When the graphics handler generates the fill area, it uses the *individual* setting for color.

To tell DEC GKS which ASF values to use, you must pass ASP_SOURCE_FLAGS to GKS$SET_ASF.

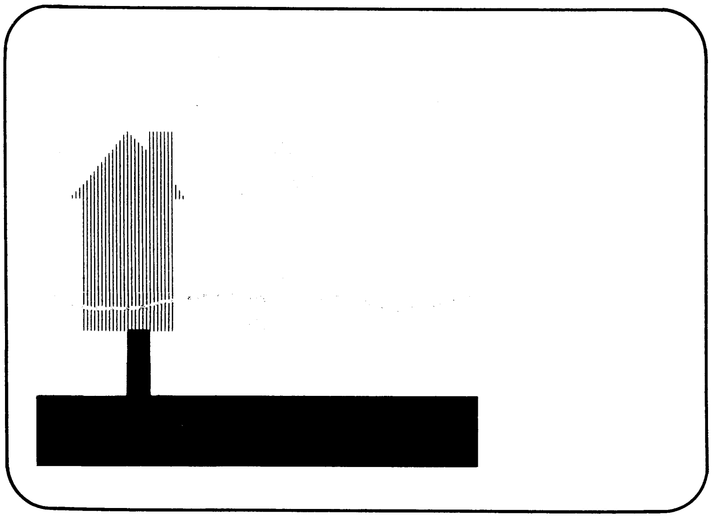
❹ This call to GKS$SET_FILL_REP changes the fill area representation associated with fill index number 2 (FILL_INDEX). The new representation is as follows:

Index	Interior Style	Style Index	Color Index	Description
	.			
	.			
	.			
2	3	–5	2	Red hatched fill
	.			
	.			
	.			

❺ To tell the graphics handler which index value to use for the fill area bundled values, call GKS$SET_FILL_INDEX and pass the index value 2 (FILL_INDEX).

Figure 5–4 illustrates the effects of this subroutine on the workstation surface. Keep in mind that the picture may look different depending on the device you are using. Notice that the colors of the house and tree did not change to red (as specified in bundle representation number 2), since the graphics handler continued to use the individual color attribute setting.

Figure 5–4: Changing the Bundle Representation—VT241



When you change individual attribute settings, the change does not take place until subsequent generation of the appropriate output primitive. This is not true for bundle representation changes. A change to a bundle representation affects previously generated primitives whose attributes are bound to that bundle representation.

When you call the SET REPRESENTATION functions, DEC GKS can either make the change immediately or can require an implicit regeneration of the workstation surface to make the change. In Chapter 2, Programming With DEC GKS, the change to the color representation causes the VT241 to make the changes immediately, without redrawing the entire picture. However, other devices may require a surface regeneration, which would delete all primitives not contained in segments.

In the previous example, there exists no previously generated fill areas that are affected by the change to the bundle representation. Consequently, you do not have to check for a postponed surface regeneration. See Section 5.3 for more information concerning deferral and regeneration.

5.1.5 Text Attributes

When you work with normalization transformation changes, you may find that you need to alter the text attributes in order to maintain the size and proportion of the character string.

For example, if you map a normalization window onto a small portion of the default NDC space ([0,1] x [0,1]), the text maintains its relative position, its alignment, color, font, precision, path, and direction. However, the text may then appear small, crowded, and possibly skewed when mapped to the workstation viewport.

Fortunately, DEC GKS calculates the character spacing and width according to the current text height. Consequently, you can adjust all three attributes by adjusting the character height to the current normalization transformation proportions.

The following code example illustrates how to adjust text height:

```
      INTEGER WS_ID, HOUSE, TREE, HORIZON, STARS, TITLE,
     * SIDE, ROAD

      DATA WS_ID / 1 /, TITLE / 1 /, STARS / 2 /,
     * TREE / 3 /, SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /,
     * HOUSE / 7 /

      CALL SET_UP( WS_ID )

      CALL DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      CALL TITLE_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
          .
          .
          .
      CALL CLEAN_UP( WS_ID )

      END
          .
          .
          .
C     ************************************************************
C     Adjusting text according to normalization transformations.
      SUBROUTINE TITLE_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, UNITY, UPPER_QUARTER

      REAL TEXT_START_X, TEXT_START_Y, HEIGHT_RATIO,
     * NEW_MAX_X, NEW_MIN_Y, WIDTH_RATIO, LARGER, TEMP
```

```
      DATA TEXT_START_X / 0.05 /, TEXT_START_Y / 0.9 /,
     * UNITY / 0 /, UPPER_QUARTER / 1 /, NEW_MAX_X / 0.6 /,
     * NEW_MIN_Y / 0.5 /, LARGER / 0.04 /

  C   Clear the screen and delete all of the segments from WS_ID.
❶     CALL GKS$CLEAR_WS( WS_ID, GKS$K_CLEAR_ALWAYS )

  C   Set the text height as in the previous subroutine.
❷     CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )

  C   Pause.  Type RETURN when finished viewing the picture. Then
  C   clear the screen.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)
      CALL GKS$CLEAR_WS( WS_ID, GKS$K_CLEAR_ALWAYS )

  C   Map the default normalization window to the upper left quarter
  C   of the NDC space.  When text is mapped to a smaller viewport,
  C   the character height, character width, and character spacing
  C   is effected.
❸     CALL GKS$SET_WINDOW( UPPER_QUARTER, 0.0, 1.0,
     * 0.0, 1.0 )
      CALL GKS$SET_VIEWPORT( UPPER_QUARTER, 0.0, NEW_MAX_X,
     * NEW_MIN_Y, 1.0 )
      CALL GKS$SELECT_XFORM( UPPER_QUARTER )

  C   Show the effects on text height...
❹     CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )

  C   Pause.  Type RETURN when finished viewing the picture. Then
  C   clear the screen.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)
      CALL GKS$CLEAR_WS( WS_ID, GKS$K_CLEAR_ALWAYS )

  C   Determine the change to the Y values made from the normalization
  C   transformation change.
❺     HEIGHT_RATIO = ( 1.0 - NEW_MIN_Y) / 1.0
      WIDTH_RATIO = NEW_MAX_X / 1.0

  C   Turn off clipping and adjust the text height.
❻     CALL GKS$SET_CLIPPING( GKS$K_NOCLIP )
      CALL GKS$SET_TEXT_HEIGHT( LARGER + (LARGER * HEIGHT_RATIO ))

❼     CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

  C   Reset the normalization viewport.
      CALL GKS$SET_CLIPPING( GKS$K_CLIP )
      CALL GKS$SELECT_XFORM( UNITY )

      RETURN
      END
```

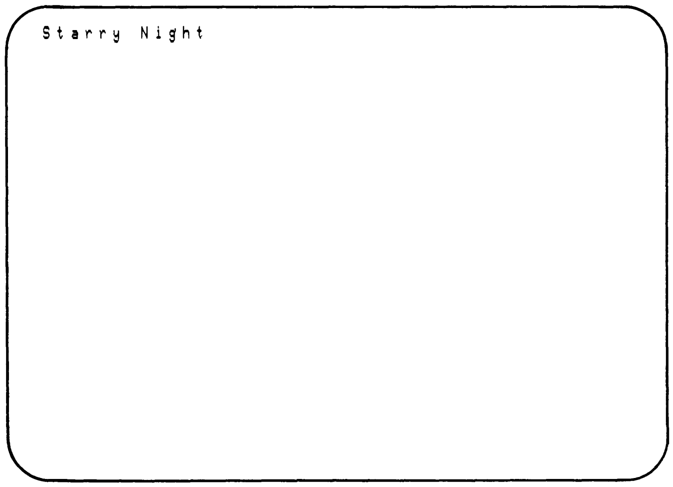
The following numbers correspond to the numbers in the previous example:

1. This code clears the workstation and also deletes all segments associated with the workstation. Once deleted, you cannot recall these segments. See Section 5.2.1 for more information concerning more efficient segment storage.
2. This code generates text in the same proportions as most previous examples in this manual. Figure 5–5 illustrates the generated text.
3. This code maps all output to the upper left quarter of the NDC space ([0,0.5] x [0.5,1.0]). Since you specify text height in world coordinate units, the new normalization transformation reduces the height proportionately with the viewport reduction. Also, since character spacing and width are dependent on character height, those attribute values are reduced.
4. This code generates the reduced text using the new normalization transformation. Figure 5–6 illustrates the generated text.
5. This code calculates the reduction ratio of the Y NDC range according to the changes in the normalization viewport. The change in the Y range affects text height whereas the change to the X range does not.
6. Since the change in text height within the world coordinate space may cause the text to exceed the defined normalization window, you should turn off clipping so that DEC GKS maps all of the text to NDC space. Figure 5–7 illustrates the need to turn off clipping.

   When you adjust the text height, you use the calculated Y axis ratio. Adjusting the text height automatically adjusts the character spacing and width.
7. This code generates the adjusted text. Figure 5–8 illustrates the effect of the adjustment.

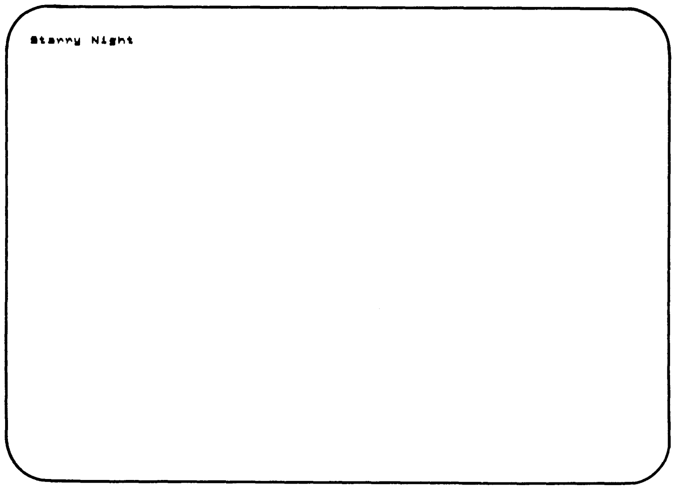
If you use normalization transformations whose viewport boundaries are disproportionate to the window boundaries (for instance, mapping a square to a wide rectangle), then you may need to adjust the character expansion factor so that the text width is adequate. For more information concerning character expansion, refer to Chapter 5, Output Attribute Functions, in the *DEC GKS Reference Manual.*

Figure 5–5: Generating Text—VT241


Starry Night

ZK 5206-86

Figure 5–6: Reducing the Normalization Viewport—VT241



ZK-5200-86

Figure 5–7: Clipping the Adjusted Text

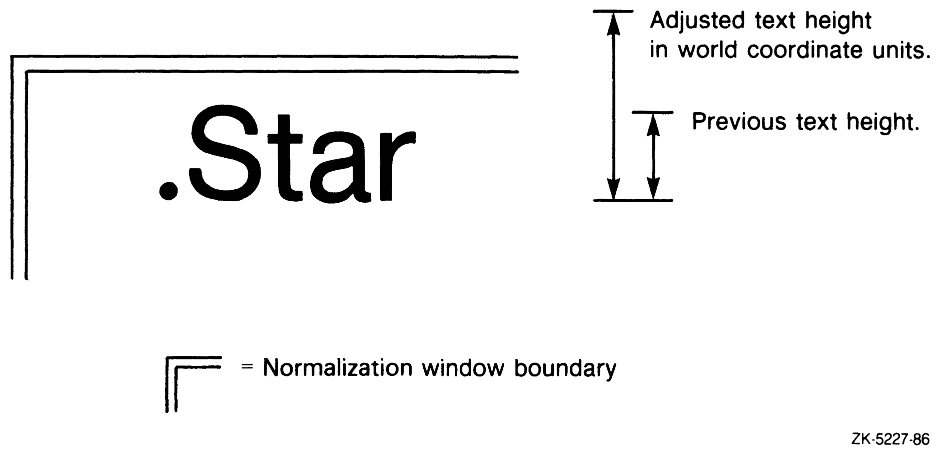
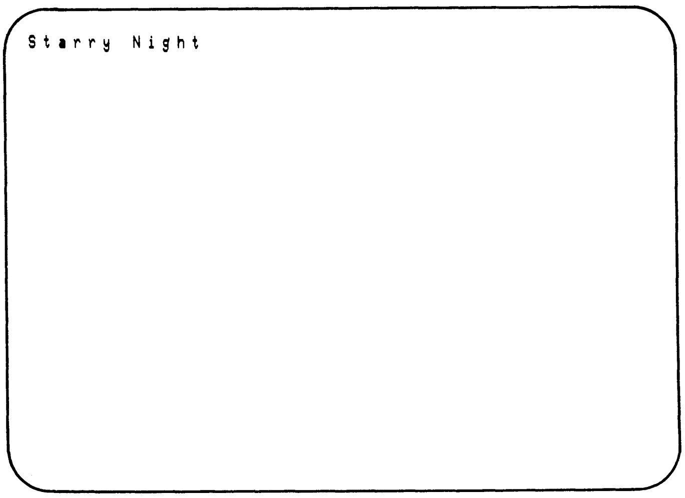


Figure 5–8: Adjusting Text to Normalization Transformation Changes—VT241



ZK-5201-86

5.2 Using Segments

In all previous code examples, the only purpose for placing the output primitives into segments was the following:

- To keep the primitives from being deleted upon surface regeneration
- To take advantage of having output attributes *bound* to the primitives at the time of generation.

As an example of binding attribute values to a primitive, consider the regeneration of the horizon. Every time DEC GKS needs to regenerate that segment, it knows to draw a dashed-dotted line, to increase the width, to represent that line according to color index number 1, and to clip that horizon line at the default normalization viewport, ([0,1] x [0,1]) in NDC space.

Consequently, when DEC GKS creates a segment, it stores the attribute and clipping information at the time of output generation. You cannot change these attributes and settings for the previously generated primitive. As an example, if you change the normalization window so that the horizon's plotted points fall outside of the current window and viewport, DEC GKS still draws and clips the horizon as generated.

In fact, DEC GKS stores the primitive's NDC coordinate values, since you may have composed portions of a segment from many different normalization windows; the NDC space has workable limitations and is device independent.

The following sections discuss how to take advantage of the full capabilities provided in DEC GKS segment support.

5.2.1 Workstation Independent Segment Storage

One advantage to using segments is that it provides a means to transport output primitives from a device-independent storage structure to various workstations. This storage structure is called workstation independent segment storage (WISS).

WISS is a data structure that stores information pertinent to the primitives contained in a segment. Since DEC GKS treats WISS as a workstation, all you have to do is activate WISS as you do any other workstation at the time of segment creation.

To insure that your level of GKS supports the WISS workstation, you can use the following code:

```
           .
           .
           .
C     Make sure that WISS is supported.
      CALL GKS$INQ_LEVEL( ERROR_STATUS, GKS_LEVEL )

      IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
     *   ( GKS_LEVEL .LT. GKS$K_LEVEL_2A )) THEN
             WRITE(6,*)
     *       'This level of GKS does not support WISS.'
             WRITE(6,*) 'Error status:', ERROR_STATUS
             STOP
      ENDIF
           .
           .
           .
```

To open and activate WISS, use the following code:

```
           .
           .
           .
      GKS$OPEN_WS( 2, 'THIS_IS_IGNORED.TXT', GKS$K_WSTYPE_WISS )
      GKS$ACTIVATE( 2 )
```

```
C     Create segments...
          .
          .
          .
```

Once you store all desired segments on WISS, you can deactivate the workstation as you do any other. However, you cannot close WISS unless you are finished using WISS for storage. In other words, you cannot copy segments from WISS to other open workstations unless WISS is open. When you are finished using WISS, you can close WISS as you do any other workstation. Once you close WISS, DEC GKS deletes all stored segments in WISS.

There are three ways to transport a segment (or its primitives) from WISS to other open workstations, as follows:

1. *Associate* the segment so that the receiving workstation stores the identical segment.
2. *Copy* the segment's primitives so that the receiving workstation generates the primitives but does not store them as a segment.
3. *Insert* the segment's primitives so that the receiving workstation generates the primitives but does not store them as a segment.

The difference between copying and inserting a segment is that you can insert a segment's primitives into an open segment, but you cannot copy a segment's primitives into an open segment. The receiving workstation does *not* treat the inserted set of segment primitives as a segment, but *does* add those transformed primitives to the segment being created. If you insert a segment at a time when there is no segment open, segment insertion transforms the segment and then copies the primitives to the workstation surface. (When a segment is open, DEC GKS is in the *operating state* GKS$K_SGOP.)

During segment insertion, DEC GKS allows you to specify an additional segment transformation matrix to apply to the inserted segment primitives. A segment transformation allows you to scale, rotate, and shift (or, *translate*) all of the primitives in the segment. To review the order in which DEC GKS applies normalization, segment, and insertion transformations, refer to Chapter 9, Segment Functions, in the *DEC GKS Reference Manual*. See Section 5.2.2 for detailed information concerning segment transformations.

In the subroutine TITLE_ATTS, the call to GKS$CLEAR_WS deletes all segments stored in workstation dependent segment storage (WDSS) on the workstation WS_ID. In the previous subroutine, there was no way to recall those segments without recreating them. The following example shows you how to use WISS to copy the segments back to the workstation after deletion (as long as you do not delete them from WISS).

```
        INTEGER WS_ID, HOUSE, TREE, HORIZON, STARS, TITLE,
       * SIDE, ROAD, WISS

        DATA WS_ID / 1 /, WISS / 2 /, TITLE / 1 /, STARS / 2 /,
       * TREE / 3 /, SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /,
       * HOUSE / 7 /

        CALL SET_UP( WS_ID, WISS )

        CALL DRAW_PICTURE( WS_ID, WISS, TITLE, STARS, TREE, SIDE,
       * ROAD, HOUSE, HORIZON )

        CALL TITLE_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
       * ROAD, HOUSE, HORIZON )

        CALL CLEAN_UP( WS_ID, WISS )

        END

  C     ***********************************************************
  C     Set up the DEC GKS and the workstation environments...
        SUBROUTINE SET_UP( WS_ID, WISS )

        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, ERROR_STATUS, CATEGORY, INQUIRY_OKAY,
       * DUMMY_INTEGER, DEF_MODE, REGEN_FLAG, WISS
            .
            .
            .
        CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT,
       * GKS$K_WSTYPE_DEFAULT )
❶       CALL GKS$OPEN_WS( WISS, GKS$K_CONID_DEFAULT,
       * GKS$K_WSTYPE_WISS )

        CALL GKS$ACTIVATE_WS( WS_ID )
            .
            .
            .
  C     ***********************************************************
  C     Draw the picture, and place each primitive in a segment...
        SUBROUTINE DRAW_PICTURE( WS_ID, WISS, TITLE, STARS, TREE, SIDE,
       * ROAD, HOUSE, HORIZON )

        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, WISS, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
            .
            .
            .
  C     Store the segments in Workstation Independent Storage.
❷       CALL GKS$ACTIVATE_WS( WISS )

        CALL GKS$CREATE_SEG( TITLE )
        CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
       * 'Starry Night' )
        CALL GKS$CLOSE_SEG()
            .
            .
            .
        CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
        CALL GKS$CLOSE_SEG()
```

```
C     Reset the normalization transformation to the default...
      CALL GKS$SELECT_XFORM( UNITY )

C     Do not store any more segments in WISS.
❸     CALL GKS$DEACTIVATE_WS( WISS )

      RETURN
      END
          .
          .
          .
C     ************************************************************
C     Adjusting text according to normalization transformations.
      SUBROUTINE TITLE_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
          .
          .
          .
C     Turn off clipping and adjust the text height.
      CALL GKS$SET_CLIPPING( GKS$K_NOCLIP )
      CALL GKS$SET_TEXT_HEIGHT( LARGER + (LARGER * HEIGHT_RATIO ))

      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

C     Reset the normalization viewport.
      CALL GKS$SET_CLIPPING( GKS$K_CLIP )
      CALL GKS$SELECT_XFORM( UNITY )

C     Redraw the segments unconditionally.  A call to GKS$UPDATE_WS
C     with the argument GKS$K_PERFORM_FLAG will do the same thing,
C     but only if the workstation surface is out of date.
❹     CALL GKS$REDRAW_SEG_ON_WS( WS_ID )
      READ(5,*)

C     Restore the picture. Associate segments on WISS with WS_ID.
C     WS_ID stores the primitives as segments.
❺     CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, TITLE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, STARS )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, TREE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, SIDE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, ROAD )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, HOUSE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, HORIZON )

C     Redraw the segments.
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

      RETURN
      END
          .
          .
          .
C     ***********************************************************
C     Clean up the DEC GKS and the workstation environments...
      SUBROUTINE CLEAN_UP( WS_ID, WISS )

      IMPLICIT NONE
      INTEGER WS_ID, WISS

      CALL GKS$DEACTIVATE_WS( WS_ID )
```

```
        CALL GKS$CLOSE_WS( WS_ID )
❻       CALL GKS$CLOSE_WS( WISS )
        CALL GKS$CLOSE_GKS()

        RETURN
        END
```

The following numbers correspond to the numbers in the previous example:

❶ This code opens WISS and assigns the workstation identifier 2 (WISS). DEC GKS ignores the connection identifier argument (GKS$K_CONID_DEFAULT). This example chooses not to activate WISS until segment creation.

❷ Just before segment creation, this code activates WISS as it does any other workstation. DEC GKS generates created segments on all active workstations (in this example, workstations WS_ID and WISS).

❸ Once segment creation is complete, this code deactivates WISS.

❹ This call to GKS$REDRAW_SEG_ON_WS only clears the screen since the previous call to GKS$CLEAR_WS deleted all segments stored on WS_ID. Note that the call to GKS$REDRAW_SEG_ON_WS deletes all primitives not contained in a segment (the text at the top of the workstation surface).

❺ This code associates all segments on WISS with WS_ID. The next call to GKS$REDRAW_SEG_ON_WS reproduces the entire Starry Night picture since all segments are once again stored on WS_ID.

❻ This code closes WISS.

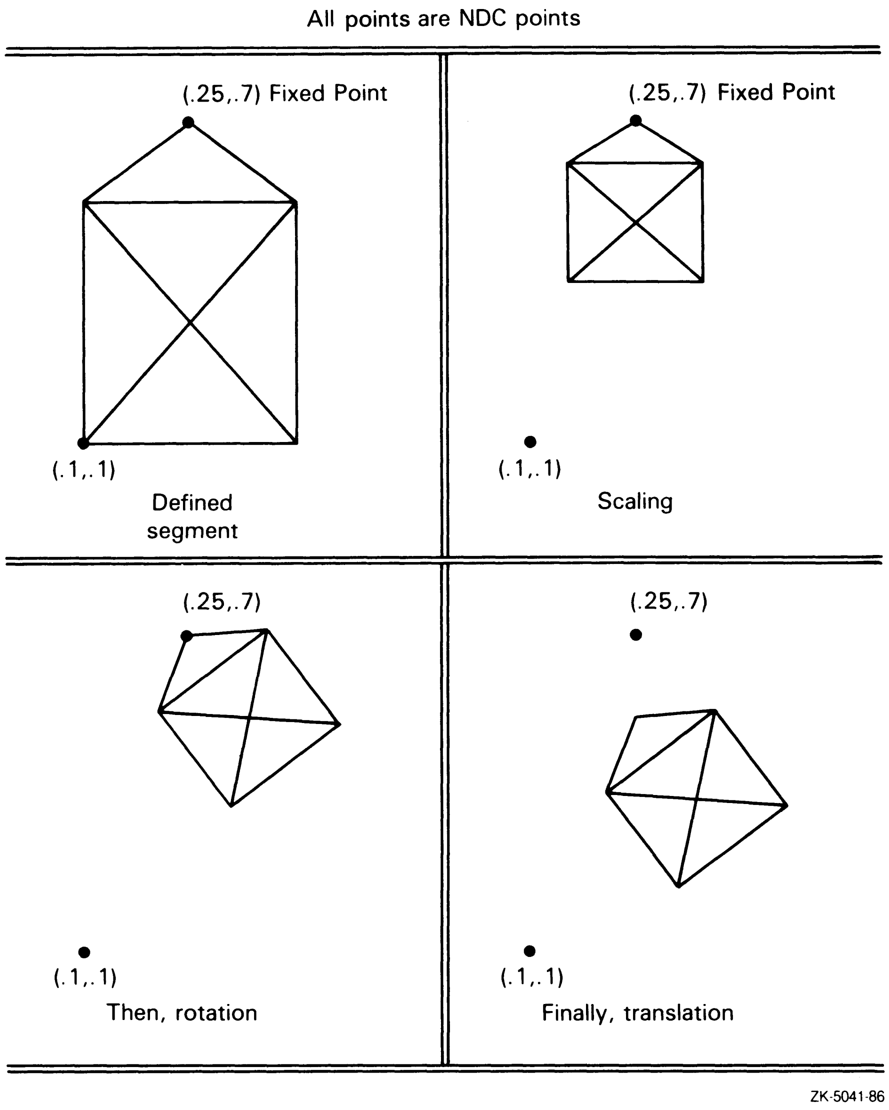
5.2.2 Segment Transformations

When generating a picture containing a segment, DEC GKS performs tasks in the following order:

1. Establishes the current normalization transformation.
2. Applies the current segment transformation to the primitives in the defined segment.
3. Clips each of the segment primitives according to the bound clipping rectangles.
4. If clipping is enabled, clips the entire picture at the current clipping rectangle (normalization viewport in NDC space).

The segment transformation is the establishment of a matrix designating values for scaling, rotation, and translation. When DEC GKS applies a segment transformation, it first applies the scaling values, then applies the rotation values, and last applies the translation values. Figure 5–9 illustrates this process.

Figure 5–9: Order of Segment Transformations



By default, DEC GKS applies the *identity* segment transformation to all segments. This transformation specifies the values 0.0 for both the X and Y fixed points, 0.0 for the X and Y translation vectors, 0.0 for the rotation value, and 1.0 for the X and Y scaling values. The identity transformation makes no changes to the segment primitives as they are stored and clipped on the NDC coordinate plane.

To take advantage of segment transformations, you need to specify component values to either GKS$EVAL_XFORM_MATRIX or GKS$ACCUM_XFORM_MATRIX, and then pass the matrix to GKS$SET_SEG_XFORM to establish that transformation as the current segment transformation. The following code example shows how to work with segment transformations:

```
      INTEGER WS_ID, HOUSE, TREE, HORIZON, STARS, TITLE,
     * SIDE, ROAD, WISS

      DATA WS_ID / 1 /, WISS / 2 /, TITLE / 1 /, STARS / 2 /,
     * TREE / 3 /, SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /,
     * HOUSE / 7 /

      CALL SET_UP( WS_ID, WISS )

      CALL DRAW_PICTURE( WS_ID, WISS, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      CALL TITLE_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      CALL SEG_ATTS( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON )

      CALL CLEAN_UP( WS_ID, WISS )

      END
         .
         .
         .
C     ***********************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, WISS, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
         .
         .
         .
❶     DATA HOUSE_X / 100.0, 300.0, 300.0, 325.00, 300.0, 300.0,
     * 250.0, 250.0, 200.0, 75.0, 100.0, 100.0 /
      DATA HOUSE_Y / 300.0, 300.0, 600.0, 600.0, 640.0, 750.0,
     * 750.0, 700.0, 750.0, 600.0, 600.0, 3.00 /
         .
         .
         .
C     Map the house onto a small portion of the NDC space...
      CALL GKS$SET_WINDOW( HOUSE_NORM, 75.0, 325.0, 300.0,
     * 750.0 )
      CALL GKS$SET_VIEWPORT( HOUSE_NORM, 0.075, 0.325,
     * 0.3, 0.75 )
❷     CALL GKS$SELECT_XFORM( HOUSE_NORM )

      CALL GKS$CREATE_SEG( HOUSE )
C     Only change the color index if working with a color workstation
C     (or a VT125/240 or a VAXstation).
      IF ( NUM_INDEXES .GE. THREE ) THEN
          CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()
      C     Reset the normalization transformation to the default...
            CALL GKS$SELECT_XFORM( UNITY )
                .
                .
                .
      C     ************************************************************
      C     Illustrate segment transformations...
            SUBROUTINE SEG_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
           * ROAD, HOUSE, HORIZON )

            IMPLICIT NONE
            INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
            INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
           * HORIZON, ERROR_STATUS, NUM_PRIORITIES, DUMMY_INTEGER,
           * WS_TYPE, SEG_XFORM, VIS_TO_INVIS, HIGH_CHANGE,
           * PRIOR_CHANGE

            CHARACTER*80 DUMMY_STRING

    ❸       REAL HOUSE_XFORM_MATRIX( 6 ), DUMMY_REAL,
           * HOUSE_ROTATION, TREE_ROTATION, HOUSE_FIXED_X,
           * TREE_XFORM_MATRIX( 6 ), HOUSE_FIXED_Y, TREE_FIXED_X,
           * TREE_FIXED_Y, VECTOR_X, VECTOR_Y, SCALE_X_1,
           * SCALE_Y_1, SCALE_X_2, SCALE_Y_2,
           * TITLE_XFORM_MATRIX( 6 ), IDENTITY( 6 ), DUMMY_SCALE

    ❹       DATA  HOUSE_FIXED_X / 0.2 /,
           * HOUSE_FIXED_Y / 0.525 /, TREE_FIXED_X / 0.52 /,
           * TREE_FIXED_Y / 0.35 /, VECTOR_X / 0.33 /,
           * VECTOR_Y / -0.1 /, SCALE_X_1 / 0.25 /,
           * SCALE_Y_1 / 0.25 /, DUMMY_REAL / 0.0 /,
           * SCALE_X_2 / 5.2 /, SCALE_Y_2 / 5.2 /,
           * DUMMY_SCALE / 1.0 /

      C     The house's rotation is 180 degrees and the tree's rotation
      C     is -20 degrees.
    ❺       HOUSE_ROTATION = 3.14
            TREE_ROTATION = -3.14/9.0

      C     Obtain the workstation type.
            CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
           * WS_TYPE, DUMMY_INTEGER )

      C     Find out if changes to the segment attributes require
      C     implicit regenerations or if changes occur immediately...
    ❻       CALL GKS$INQ_DYN_MOD_SEG_ATTB( WS_TYPE, ERROR_STATUS,
           * SEG_XFORM, VIS_TO_INVIS, DUMMY_INTEGER, HIGH_CHANGE,
           * PRIOR_CHANGE, DUMMY_INTEGER, DUMMY_INTEGER )

      C     Establish an identity segment transformation...
    ❼       CALL GKS$EVAL_XFORM_MATRIX( DUMMY_REAL, DUMMY_REAL,
           * DUMMY_REAL, DUMMY_REAL, DUMMY_REAL, DUMMY_SCALE,
           * DUMMY_SCALE, GKS$K_COORDINATES_NDC, IDENTITY )

      C     Flip the house onto its roof...
            CALL GKS$EVAL_XFORM_MATRIX( HOUSE_FIXED_X, HOUSE_FIXED_Y,
           * DUMMY_REAL, DUMMY_REAL, HOUSE_ROTATION, DUMMY_SCALE,
           * DUMMY_SCALE, GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
    ❽       CALL GKS$SET_SEG_XFORM( HOUSE, HOUSE_XFORM_MATRIX )
```

```
  C     Shrink the tree...
        CALL GKS$EVAL_XFORM_MATRIX( TREE_FIXED_X, TREE_FIXED_Y,
       * DUMMY_REAL, DUMMY_REAL, DUMMY_REAL, SCALE_X_1,
       * SCALE_Y_1, GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
        CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )

  C     Move the title...
        CALL GKS$EVAL_XFORM_MATRIX( DUMMY_REAL, DUMMY_REAL,
       * VECTOR_X, VECTOR_Y, DUMMY_REAL, DUMMY_SCALE,
       * DUMMY_SCALE, GKS$K_COORDINATES_NDC, TITLE_XFORM_MATRIX )
        CALL GKS$SET_SEG_XFORM( TITLE, TITLE_XFORM_MATRIX )

  C     Pause.  Type RETURN when finished viewing the picture.
  C     If regeneration is needed, do it...
❾       IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
             READ(5,*)
        ELSE
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
             READ(5,*)
        ENDIF

  C     By accumulating the tree's matrix, we can add to the
  C     translation increment by increment...
❿       CALL GKS$ACCUM_XFORM_MATRIX( TREE_XFORM_MATRIX,
       * TREE_FIXED_X, TREE_FIXED_Y, DUMMY_REAL, DUMMY_REAL,
       * TREE_ROTATION, SCALE_X_2, SCALE_Y_2,
       * GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
        CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )

  C     Pause.  Type RETURN when finished viewing the picture.
⓫       IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
             READ(5,*)
        ELSE
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
             READ(5,*)
        ENDIF

  C     By accumulation, shift the tree's X coordinate points by 0.15...
        CALL GKS$ACCUM_XFORM_MATRIX( TREE_XFORM_MATRIX,
       * DUMMY_REAL, DUMMY_REAL, 0.15, 0.0, DUMMY_REAL,
       * DUMMY_SCALE, DUMMY_SCALE, GKS$K_COORDINATES_NDC,
       * TREE_XFORM_MATRIX )
        CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )

  C     Pause.  Type RETURN when finished viewing the picture.
⓬       IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
             READ(5,*)
        ELSE
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
             READ(5,*)
        ENDIF
```

```
   C     Return the tree to its original size and position...
⓭         CALL GKS$SET_SEG_XFORM( TREE, IDENTITY )

   C     Pause.  Type RETURN when finished viewing the picture.
⓮         IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
              READ(5,*)
         ELSE
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
              READ(5,*)
         ENDIF

   C     Shift the house past its normalization viewport boundary to show
   C     how segments are clipped...
⓯         CALL GKS$ACCUM_XFORM_MATRIX( HOUSE_XFORM_MATRIX,
        * DUMMY_REAL, DUMMY_REAL, 0.1, 0.0, DUMMY_REAL, DUMMY_SCALE,
        * DUMMY_SCALE, GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( HOUSE, HOUSE_XFORM_MATRIX )

   C     Pause.  Type RETURN when finished viewing the picture.
⓰         IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
              READ(5,*)
         ELSE
              CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
              READ(5,*)
         ENDIF

         RETURN
         END
              .
              .
              .
```

The following numbers correspond to the numbers in the previous example:

❶ This code plots the house using different world coordinate points. In order to map the house to the NDC plane with the rest of the picture, you need to redefine the normalization window and viewport before you generate the house.

❷ This code redefines the normalization window and viewport for the generation of the house only. Figure 5–10 illustrates the mapping of the house to NDC space.

❸ This code declares the variable that will contain the segment transformation matrix, a real array with 6 elements.

❹ This code defines the fixed point values, the translation vectors, and the scale factors. The translation vectors are X and Y values that are added to the current coordinates to alter the segment's position. The scale factors are values that are multiplied by the distance between a segment's fixed point and points in the segment primitives, shrinking or expanding the segment. The fixed points are values that are needed to scale the segment, and are needed to specify an axis on which to rotate the segment.

❺ This code establishes two rotation values in radians. Specifying negative radian values turns the segment counterclockwise. You specify a full circle by passing 2*pi radians. You specify 180 degrees by passing pi radians. The value pi equals approximately 3.14.

❻ This code inquires whether or not DEC GKS needs to regenerate the workstation surface in order to implement changes to segment transformations and other segment attributes. The arguments SEG_XFORM, VIS_TO_INVIS, HIGH_CHANGE, and PRIOR_CHANGE can be compared to the constant GKS$K_IRG (Implicit ReGeneration) to see if the graphics handler requires a surface regeneration or if the change is immediate.

❼ This code creates an identity matrix using GKS$EVAL_XFORM_MATRIX. The component variables DUMMY_REAL (0.0) and DUMMY_SCALE (1.0) are used to specify no change to the original position or size of the segment as defined. This code does *not* place this transformation into effect for any segment, it only creates a matrix.

The argument GKS$K_COORDINATES_NDC specifies that the fixed point and translation components are expressed in NDC points. If you choose, you can work with world coordinate values by passing the argument GKS$K_COORDINATES_WC. If you choose world coordinates, DEC GKS transforms the fixed points to the NDC plane using the *current* normalization transformation. If the current transformation is different than the ones used during segment creation, you may obtain unexpected results.

❽ This code establishes the current segment transformation for the house. If you call GKS$SET_SEG_XFORM twice using the same transformation matrix, DEC GKS produces a segment of the same size and position. A call to GKS$SET_SEG_XFORM only replaces one current segment transformation with another one. The effects are not cumulative.

❾ Figure 5–11 illustrates the effects of the current segment transformation values.

❿ To establish a matrix that simulates a cumulative effect of several segment transformations, you can use GKS$ACCUM_XFORM_MATRIX. GKS$ACCUM_XFORM_MATRIX accepts a transformation matrix as its first argument, mathematically implements the scaling, rotation, and translation values specified as the next set of arguments, and writes a new matrix to the last argument. GKS$ACCUM_XFORM_MATRIX creates a new matrix that is functionally equivalent to establishing the first matrix and then implementing the new component values from that size and position.

⓫ Figure 5–12 shows the effect of the matrix created by the call to GKS$ACCUM_XFORM_MATRIX.

⓬ Figure 5–13 shows the effect of the matrix created by the second call to GKS$ACCUM_XFORM_MATRIX.

⓭ This code reestablishes the identity transformation for the tree.

⓮ Figure 5–14 illustrates the effect of the identity transformation on the tree.

⓯ After accumulating the house's transformation by adding an X translation vector of 0.1, the house is no longer contained in the normalization viewport that DEC GKS stores with the segment as its clipping rectangle.

Keep in mind that the house is the only primitive that uses a different normalization viewport (clipping rectangle); the program maps all other primitives to the default NDC area ([0,1] x [0,1]). DEC GKS stores clipping rectangles with segment primitives at the time of creation. Their clipping rectangles cannot be changed.

⓰ Figure 5–15 illustrates the surface of the workstation after executing this code. After translating the house, the primitive exceeds the normalization viewport that was in effect when the house was placed in the segment. The only way to prevent clipping of segments during translation or scaling (when they may cross their bound clipping rectangles) is to disable clipping when you create the segment, as follows:

```
C     Disable clipping...
      CALL GKS$SET_CLIPPING( GKS$K_NOCLIP )

      CALL GKS$CREATE_SEG( HOUSE )
          .
          .
          .
      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()

C     Re-enable clipping (if desired)...
      CALL GKS$SET_CLIPPING( GKS$K_CLIP )
```

Figure 5–10: Placement of the House on NDC Space

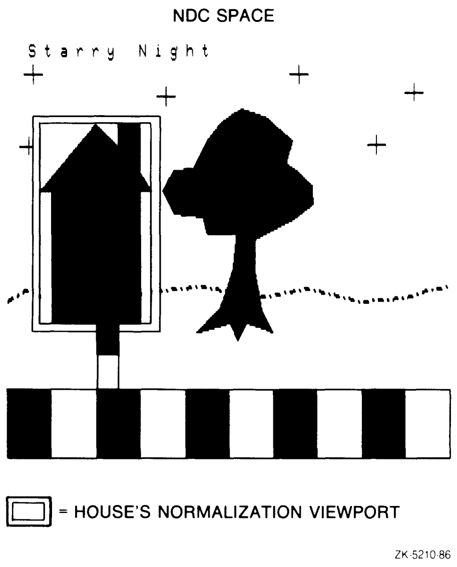


Figure 5–11: Transformed Segments—VT241

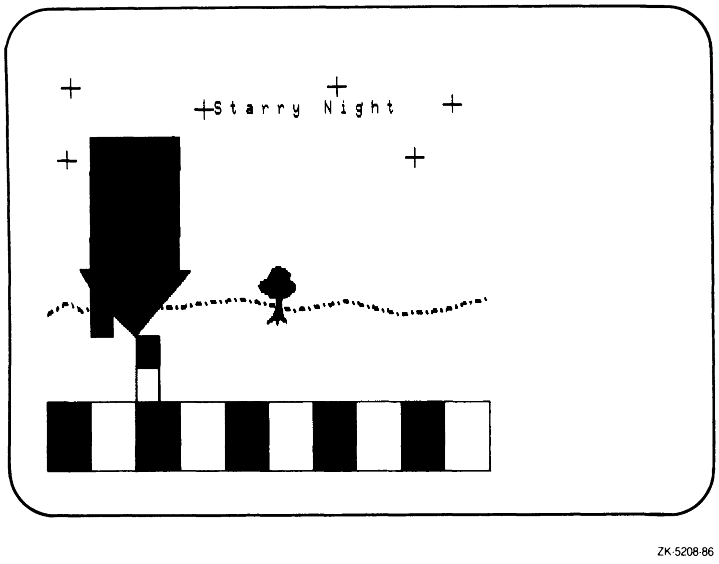
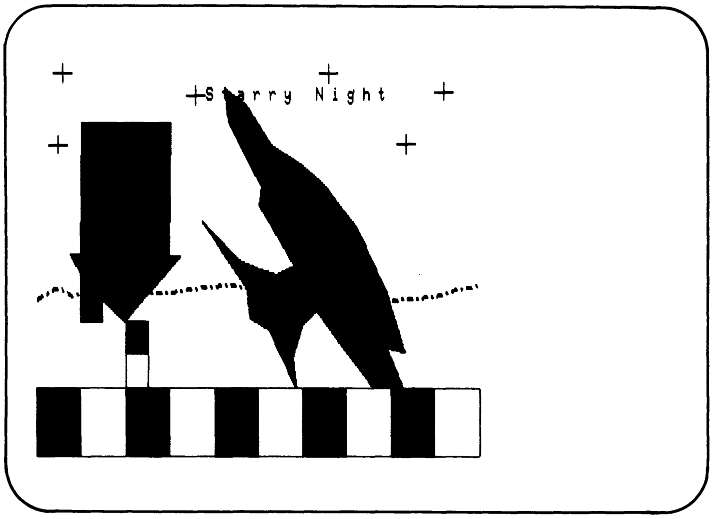
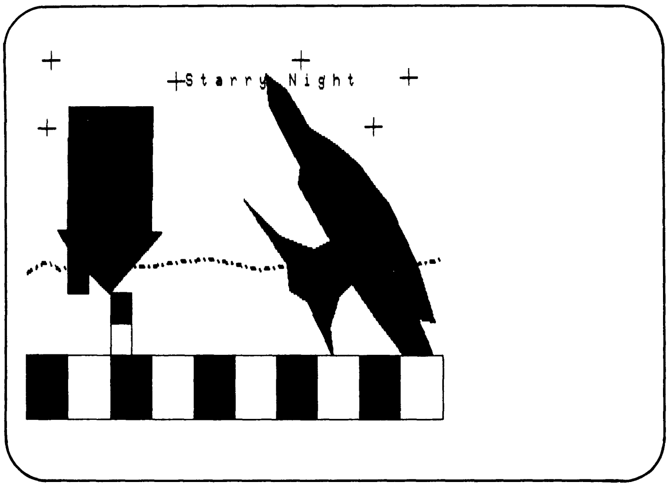


Figure 5–12: Accumulating the Tree Segment Transformation— VT241



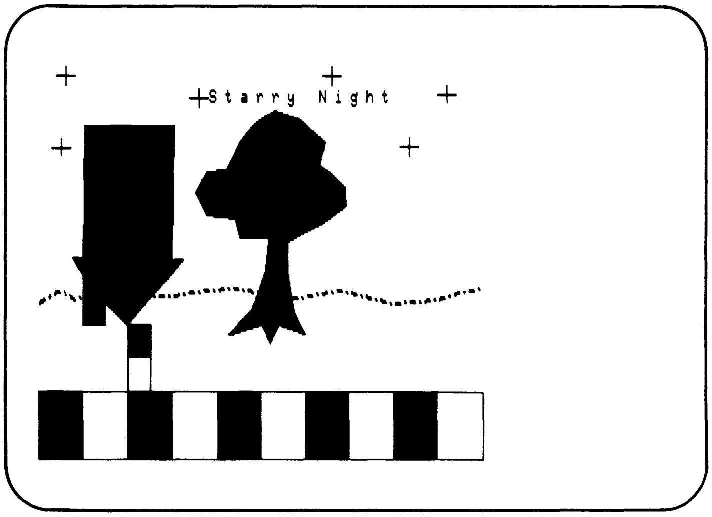
ZK-5196-86

Figure 5–13: Accumulating a Translation of the Tree—VT241



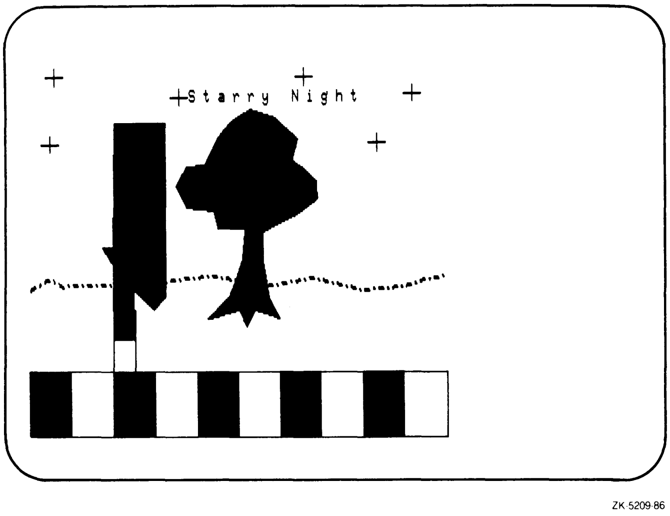
ZK-5198-86

Figure 5–14: Restoring the Identity Transformation of the Tree—VT241



ZK-5197-86

Figure 5–15: Moving Past the House's Clipping Rectangle—VT241



5.2.3 Segment Attributes

The previous subsection described segment transformations. Segment transformation is only one type of segment attribute. The following is a list of the segment attributes:

- Detectability
- Highlighting
- Priority
- Transformation
- Visibility

Segment detectability is an attribute that you use during pick input. Chapter 6, Requesting Input, discusses segment detectability in greater detail.

The following example shows you the code you need to add to the SEG_ATTS subroutine to see the effects of segment highlighting, priority, and visibility:

```
                .
                .
                .
C       ************************************************************
C       Illustrate segment transformations...
        SUBROUTINE SEG_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
      * ROAD, HOUSE, HORIZON )
                .
                .
                .
C       Obtain the workstation type.
        CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
      * WS_TYPE, DUMMY_INTEGER )

C       Find out if changes to the segment attributes require
C       implicit regenerations or if changes occur immediately...
❶       CALL GKS$INQ_DYN_MOD_SEG_ATTB( WS_TYPE, ERROR_STATUS,
      * SEG_XFORM, VIS_TO_INVIS, DUMMY_INTEGER, HIGH_CHANGE,
      * PRIOR_CHANGE, DUMMY_INTEGER, DUMMY_INTEGER )
                .
                .
                .
C       Inquire about the segment priority capabilities...
❷       CALL GKS$INQ_SEG_PRIORITY( WS_ID, ERROR_STATUS,
      * NUM_PRIORITIES )

C       Give the land a higher priority than the house...
❸       IF ( NUM_PRIORITIES .EQ. 0 ) THEN
             CALL GKS$SET_SEG_PRIORITY( HOUSE, 0.1 )
             CALL GKS$SET_SEG_PRIORITY( HORIZON, 0.2 )
        ELSE
             CALL GKS$SET_SEG_PRIORITY( HOUSE, 0.1 )
             CALL GKS$SET_SEG_PRIORITY( HORIZON, 1.0 )
        ENDIF

C       Pause.  Type RETURN when finished viewing the picture.
C       GKS$UPDATE_WS would not redraw the segments since a change
C       in priority does not require a new frame.
❹       IF ( PRIOR_CHANGE .EQ. GKS$K_IRG ) THEN
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
C             CALL GKS$REDRAW_SEG_ON_WS( WS_ID )
             READ(5,*)
        ELSE
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
             READ(5,*)
        ENDIF

C       Change some of the segment attributes...
        CALL GKS$SET_SEG_HIGHLIGHTING( TREE, GKS$K_HIGHLIGHTED )
        CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_INVISIBLE )
```

```
  C     Pause.  Type RETURN when finished viewing the picture.
❺       IF (( HIGH_CHANGE .EQ. GKS$K_IRG ) .OR.
       *   ( VIS_TO_INVIS .EQ. GKS$K_IRG )) THEN
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
             READ(5,*)
        ELSE
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
             READ(5,*)
        ENDIF

        RETURN
        END
            .
            .
            .
```

The following numbers correspond to the numbers in the previous example:

❶ This code inquires whether or not DEC GKS needs to regenerate the workstation surface in order to implement changes to segment transformations and other segment attributes. The arguments SEG_XFORM, VIS_TO_INVIS, HIGH_CHANGE, and PRIOR_CHANGE can be compared to the constant GKS$K_IRG (Implicit ReGeneration) to see if the graphics handler requires a surface regeneration or if the change is immediate.

❷ This code inquires how many segment priorities the graphics handler supports.

❸ If the previous inquiry returned the number 0, then the graphics handler supports a theoretically infinite number of priorities between 0.0 and 1.0; the difference between the priorities of the house and the horizon can be minimal.

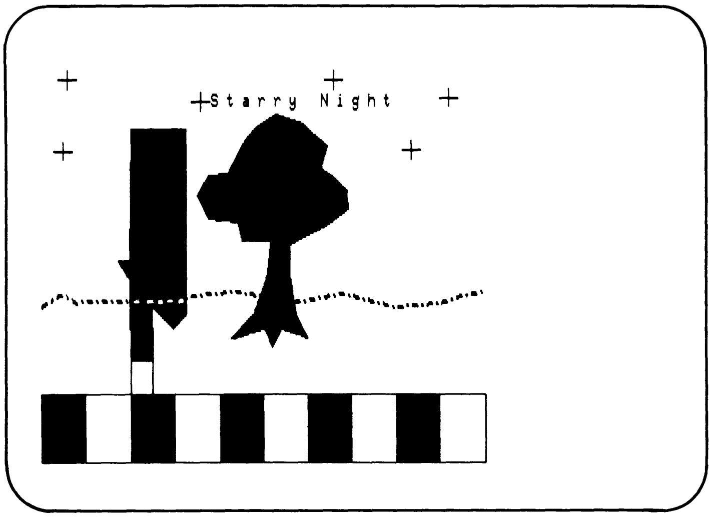
If the inquiry returned a number other than zero, DEC GKS divides the range 0.0 to 1.0 into the number specified by the inquiry. For instance, if the inquiry returned the number 2, DEC GKS assigns the same priority to all priorities between 0.0 and 0.5 inclusive, and assigns the same priority to all priorities between 0.5 and 1.0. If two segments have the same priority, the resulting picture is device dependent.

This code reassigns the segment priorities so that the horizon line appears in front of the house.

❹ Figure 5–16 illustrates the effect of this code on the picture.

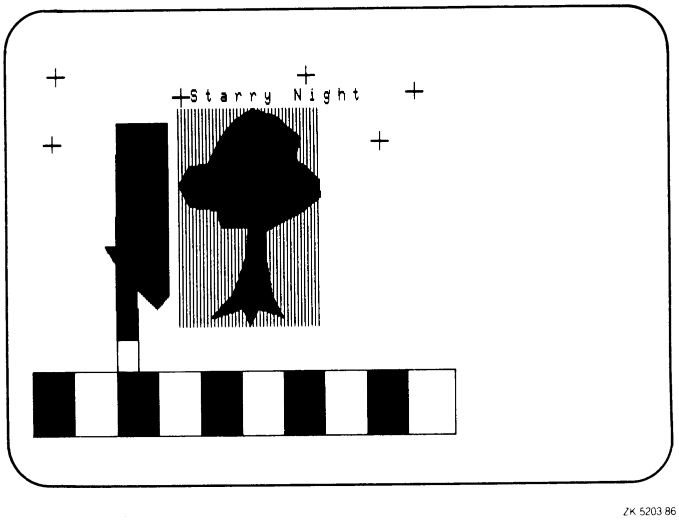
❺ Figure 5–17 illustrates the effects of segment highlighting (the tree is highlighted) and visibility (the horizon line is made invisible).

Figure 5–16: Changing Segment Priorities—VT241



ZK-5211-86

Figure 5–17: Segment Highlighting and Visibility—VT241



5.3 Surface Regeneration

Two issues to remember when generating output are output deferral and surface regeneration. The deferral of output can delay generation of a picture so that you cannot predict how much of a picture is on the workstation surface at any given moment in the program. The regeneration of the workstation surface clears the screen and only redraws visible segments; output not contained in segments is lost.

All of the programs presented so far in this manual have controlled output deferral and surface regeneration so that you have complete control over the picture presented on the workstation surface. This section serves as a quick reminder and guide to the control of the DEC GKS output deferral and implicit regeneration modes.

5.3.1 Controlling Output Deferral

If your device supports output deferral, you can use the following code to control the deferral mode:

```
      CALL GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,
     * DEF_MODE, REGEN_FLAG )

C     Defer output as long as possible and suppress implicit
C     regenerations.
      IF (( DEF_MODE .NE. GKS$K_ASTI ) .OR.
     *    ( REGEN_FLAG .NE. GKS$K_IRG_SUPPRESSED )) THEN
          CALL GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI,
     *                              GKS$K_IRG_SUPPRESSED )
      ENDIF
```

When you need to see all generated output on the workstation surface, you can flush the output buffer with the following call:

```
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
```

In calling this function, you allow the transmission of deferred data only, without causing an implicit regeneration of the workstation surface.

NOTE

When debugging your DEC GKS programs, you may wish to see generated output as you debug. To do this, set the deferral mode to GKS$K_ASAP. After you debug your program, you can set the deferral mode to any desired mode.

5.3.2 Controlling Implicit Regenerations

When you make changes to the following bundle representations, workstation state list entries, and transformations, you need to be aware that DEC GKS may implement the change dynamically or by implicitly regenerating only the segments on the workstation's surface:

- Polyline bundle representations
- Polymarker bundle representations
- Text bundle representations
- Fill area representations
- Pattern representations
- Color representations
- Segment attributes
- Workstation transformations

If you make any of these changes in your program, you should go through the following process to control possible regeneration to prevent the unnecessary loss of generated output.

First, you must check to see if the workstation is suppressing or allowing implicit regenerations, as follows:

```
      CALL GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,
     * DEF_MODE, REGEN_FLAG )

C     Defer output as long as possible and suppress implicit
C     regenerations.
      IF (( DEF_MODE .NE. GKS$K_ASTI ) .OR.
     *   ( REGEN_FLAG .NE. GKS$K_IRG_SUPPRESSED )) THEN
            CALL GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI,
     *                                GKS$K_IRG_SUPPRESSED )
      ENDIF
```

Once you have made sure that regenerations are suppressed, you can check to see if a requested change places the screen out of date in one of the following two ways:

1. If you do not request many bundle representation or workstation transformation changes, you can inquire directly about the state of the workstation surface. Calling GKS$UPDATE_WS and passing GKS$K_PERFORM_FLAG forces an implicit regeneration if the surface is out of date, causing the deletion of any output that is not part of a segment. The following code illustrates how to conditionally force an implicit regeneration:

```
C     Check to see if the picture on the screen is out of date.
      CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NEW_FRAME_FLAG )

C     Release deferred output. Regenerate if necessary.  Type
C     RETURN when you are finished viewing the picture.
      IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
C     If a regeneration is not required, just release deferred
C     output.
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF
```

2. If you request many changes of one type, you may wish to see if the graphics handler requires a regeneration to implement the change, set a flag, and then conditionally regenerate every time you make that particular change. You can use either GKS$INQ_DYN_MOD_WS_ATTB or GKS$INQ_DYN_MOD_SEG_ATTB to obtain the necessary information

about required regenerations. The following code shows how to set a regeneration flag for workstation transformations:

```
C     Find out if workstation transformations require implicit
C     regenerations, or if the change is made immediately...
      CALL GKS$INQ_DYN_MOD_WS_ATTB( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * WS_XFORMS )

      CALL GKS$SET_WS_WINDOW( WS_ID, START_X, MAX_X,
     * START_Y, MAX_Y )

C     If a regeneration is needed, do it...
      IF ( WS_XFORMS .EQ. GKS$K_IRG ) THEN
          CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF
```

5.4 Program Example Used in this Chapter

Example 5–1 presents all of the changes that you need to make to Example 3–2 in order to follow the code examples in this chapter.

Example 5–1: Using DEC GKS Output Functions

```
      IMPLICIT NONE
      INTEGER WS_ID, HOUSE, TREE, HORIZON, STARS, TITLE,
     * SIDE, ROAD, WISS

      DATA WS_ID / 1 /, WISS / 2 /, TITLE / 1 /, STARS / 2 /,
     * TREE / 3 /, SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /,
     * HOUSE / 7 /

      CALL SET_UP( WS_ID, WISS )

      CALL DRAW_PICTURE( WS_ID, WISS, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      CALL TITLE_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      CALL SEG_ATTS( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON )

      CALL CLEAN_UP( WS_ID, WISS )

      END

C     ************************************************************
C     Set up the DEC GKS and the workstation environments...
      SUBROUTINE SET_UP( WS_ID, WISS )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, ERROR_STATUS, CATEGORY, INQUIRY_OKAY,
     * DUMMY_INTEGER, DEF_MODE, REGEN_FLAG, WISS, GKS_LEVEL
          .
          .
          .

C     Make sure that WISS is supported.
      CALL GKS$INQ_LEVEL( ERROR_STATUS, GKS_LEVEL )

      IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
     *   ( GKS_LEVEL .LT. GKS$K_LEVEL_2A )) THEN
             WRITE(6,*)
     *       'This level of GKS does not support WISS.'
             WRITE(6,*) 'Error status:', ERROR_STATUS
             STOP
      ENDIF

      CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT,
     * GKS$K_WSTYPE_DEFAULT )
      CALL GKS$OPEN_WS( WISS, GKS$K_CONID_DEFAULT,
     * GKS$K_WSTYPE_WISS )
          .
          .
          .
      RETURN
200   STOP
      END
```

(continued on next page)

Example 5–1 (Cont.): Using DEC GKS Output Functions

```
C     ***********************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, WISS, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WISS,
          .
          .
          .
     * UNITY, HOUSE_NORM

      DATA WS_ID / 1 /, WISS / 2 /
          .
          .
          .
     * UNITY / 0 /, HOUSE_NORM / 1 /
          .
          .
          .
      DATA HOUSE_X / 100.0, 300.0, 300.0, 325.00, 300.0, 300.0,
     * 250.0, 250.0, 200.0, 75.0, 100.0, 100.0 /
      DATA HOUSE_Y / 300.0, 300.0, 600.0, 600.0, 640.0, 750.0,
     * 750.0, 700.0, 750.0, 600.0, 600.0, 3.00 /
          .
          .
          .

C     Store the segments in Workstation Independent Storage.
      CALL GKS$ACTIVATE_WS( WISS )

      CALL GKS$CREATE_SEG( TITLE )
      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y,
     * 'Starry Night' )
      CALL GKS$CLOSE_SEG()
          .
          .
          .
C      ********************************************************
C     Use a bundle index for some of the fill area attributes.
C     ***NOTE*** If you execute this subroutine, all fill areas will
C                be hatched from this point on in the program.
C      CALL FILL_ATTS( WS_ID, WS_TYPE )
C      ********************************************************
```

(continued on next page)

Example 5–1 (Cont.): Using DEC GKS Output Functions

```
      CALL GKS$CREATE_SEG( TREE )
      CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y )
      CALL GKS$CLOSE_SEG()
         .
         .
         .

C     Map the house onto a small portion of the NDC space...
      CALL GKS$SET_WINDOW( HOUSE_NORM, 75.0, 325.0, 300.0,
     * 750.0 )
      CALL GKS$SET_VIEWPORT( HOUSE_NORM, 0.075, 0.325,
     * 0.3, 0.75 )
      CALL GKS$SELECT_XFORM( HOUSE_NORM )

      CALL GKS$CREATE_SEG( HOUSE )
C     Only change the color index if working with a color workstation
C     (or a VT125/240 or a VAXstation).
      IF ( NUM_INDEXES .GE. THREE ) THEN
          CALL GKS$SET_FILL_COLOR_INDEX( DARK )
      ENDIF

      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()

C     Reset the normalization transformation to the default...
      CALL GKS$SELECT_XFORM( UNITY )

C     Do not store any more segments in WISS.
      CALL GKS$DEACTIVATE_WS( WISS )

      RETURN
      END

C     ************************************************************
C     Clean up the DEC GKS and the workstation environments...
      SUBROUTINE CLEAN_UP( WS_ID, WISS )

      IMPLICIT NONE
      INTEGER WS_ID, WISS
         .
         .
         .
      CALL GKS$CLOSE_WS( WS_ID )
      CALL GKS$CLOSE_WS( WISS )
      CALL GKS$CLOSE_GKS()

      RETURN
      END

C     ************************************************************
C     From this point forward, all code is additional code that you
C     need to add to the "Starry Night" program.
C     ************************************************************
```

(continued on next page)

Example 5–1 (Cont.): Using DEC GKS Output Functions

```
C     ***********************************************************
C     Use bundled attribute values for most of the fill attributes...
      SUBROUTINE FILL_ATTS( WS_ID, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, FILL_INDEX, STYLE_INDEX, ASP_SOURCE_FLAGS( 13 ),
     * RED, WS_TYPE, DUMMY_INT_ARRAY( 150 ), DUMMY_INTEGER,
     * ERROR_STATUS, INQUIRY_OKAY, NUM_HATCH, NUM_FILL_INDEXES

      DATA FILL_INDEX / 2 /, STYLE_INDEX / -5 /, RED / 2 /,
     * INQUIRY_OKAY / 0 /
C     Set all attributes to be individual (the default)...
      DATA ASP_SOURCE_FLAGS / 1,1,1,1,1,1,1,1,1,1,1,1,1 /


C     Make sure that the fill index number 2 and the hatch
C     style -5 are valid.
      CALL GKS$INQ_FILL_FAC( WS_TYPE, ERROR_STATUS, DUMMY_INTEGER,
     * %DESCR( DUMMY_INT_ARRAY ), NUM_HATCH,
     * %DESCR( DUMMY_INT_ARRAY ), NUM_FILL_INDEXES, DUMMY_INTEGER )


C     If the workstation does not have enough indexes or hatch styles,
C     signal the errors.
      IF (( NUM_HATCH .LT. 5 ) .OR.
     *   (( NUM_FILL_INDEXES .LT. FILL_INDEX ) .OR.
     *   ( ERROR_STATUS .NE. INQUIRY_OKAY ))) THEN
         WRITE(6,*)
     * 'Fill area facilities not adequate for this program.'
         WRITE(6,*) ERROR_STATUS, NUM_FILL_INDEXES, NUM_HATCH
         GO TO 300
      ENDIF

      ASP_SOURCE_FLAGS( 11 ) = GKS$K_ASF_BUNDLED
      ASP_SOURCE_FLAGS( 12 ) = GKS$K_ASF_BUNDLED
      ASP_SOURCE_FLAGS( 13 ) = GKS$K_ASF_INDIVIDUAL
      CALL GKS$SET_ASF( ASP_SOURCE_FLAGS )

C     Set the representation for bundle index 2.
      CALL GKS$SET_FILL_REP( WS_ID, FILL_INDEX,
     * GKS$K_INTSTYLE_HATCH, STYLE_INDEX, RED )

C     Set the current fill bundle index.
      CALL GKS$SET_FILL_INDEX( FILL_INDEX )

      RETURN
300   STOP
      END
```

(continued on next page)

Example 5–1 (Cont.): Using DEC GKS Output Functions

```
C     ************************************************************
C     Adjusting text according to normalization transformations.
      SUBROUTINE TITLE_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, UNITY, UPPER_QUARTER

      REAL TEXT_START_X, TEXT_START_Y, HEIGHT_RATIO,
     * NEW_MAX_X, NEW_MIN_Y, WIDTH_RATIO, LARGER, TEMP

      DATA TEXT_START_X / 0.05 /, TEXT_START_Y / 0.9 /,
     * UNITY / 0 /, UPPER_QUARTER / 1 /, NEW_MAX_X / 0.6 /,
     * NEW_MIN_Y / 0.5 /, LARGER / 0.04 /

C     Clear the screen, which deletes all of the segments from WS_ID.
      CALL GKS$CLEAR_WS( WS_ID, GKS$K_CLEAR_ALWAYS )


C     Set the text height as in the previous subroutine.
      CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )

C     Pause.  Type RETURN when finished viewing the picture. Then
C     clear the screen.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)
      CALL GKS$CLEAR_WS( WS_ID, GKS$K_CLEAR_ALWAYS )

C     Map the default normalization window to the upper left quarter
C     of the NDC space.  When text is mapped to a smaller viewport,
C     the character height, character width, and character spacing
C     is effected.
      CALL GKS$SET_WINDOW( UPPER_QUARTER, 0.0, 1.0,
     * 0.0, 1.0 )
      CALL GKS$SET_VIEWPORT( UPPER_QUARTER, 0.0, NEW_MAX_X,
     * NEW_MIN_Y, 1.0 )
      CALL GKS$SELECT_XFORM( UPPER_QUARTER )

C     Show the effects on text height...
      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )

C     Pause.  Type RETURN when finished viewing the picture. Then
C     clear the screen.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)
      CALL GKS$CLEAR_WS( WS_ID, GKS$K_CLEAR_ALWAYS )

C     Determine the change to the Y values made from the normalization
C     transformation change.
      HEIGHT_RATIO = ( 1.0 - NEW_MIN_Y) / 1.0
      WIDTH_RATIO = NEW_MAX_X / 1.0
```

(continued on next page)

```
C     Turn off clipping and adjust the text height.
      CALL GKS$SET_CLIPPING( GKS$K_NOCLIP )
      CALL GKS$SET_TEXT_HEIGHT( LARGER + (LARGER * HEIGHT_RATIO ))

      CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

C     Reset the normalization viewport.
      CALL GKS$SET_CLIPPING( GKS$K_CLIP )
      CALL GKS$SELECT_XFORM( UNITY )

C     Restore the picture. Associate segments on WISS with WS_ID.
C     WS_ID stores the primitives as segments.
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, TITLE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, STARS )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, TREE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, SIDE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, ROAD )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, HOUSE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, HORIZON )

C     Redraw the segments unconditionally.  A call to GKS$UPDATE_WS
C     with the argument GKS$K_PERFORM_FLAG will do the same thing,
C     but only if the workstation surface is out of date.
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

      RETURN
      END

C     ************************************************************
C     Illustrate segment transformations...
      SUBROUTINE SEG_ATTS( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, ERROR_STATUS, NUM_PRIORITIES, DUMMY_INTEGER,
     * WS_TYPE, SEG_XFORM, VIS_TO_INVIS, HIGH_CHANGE,
     * PRIOR_CHANGE

      CHARACTER*80 DUMMY_STRING
```

(continued on next page)

Example 5–1 (Cont.): Using DEC GKS Output Functions

```
      REAL HOUSE_XFORM_MATRIX( 6 ), DUMMY_REAL,
     * HOUSE_ROTATION, TREE_ROTATION, HOUSE_FIXED_X,
     * TREE_XFORM_MATRIX( 6 ), HOUSE_FIXED_Y, TREE_FIXED_X,
     * TREE_FIXED_Y, VECTOR_X, VECTOR_Y, SCALE_X_1,
     * SCALE_Y_1, SCALE_X_2, SCALE_Y_2,
     * TITLE_XFORM_MATRIX( 6 ), IDENTITY( 6 ), DUMMY_SCALE

      DATA  HOUSE_FIXED_X / 0.2 /,
     * HOUSE_FIXED_Y / 0.525 /, TREE_FIXED_X / 0.52 /,
     * TREE_FIXED_Y / 0.35 /, VECTOR_X / 0.33 /,
     * VECTOR_Y / -0.1 /, SCALE_X_1 / 0.25 /,
     * SCALE_Y_1 / 0.25 /, DUMMY_REAL / 0.0 /,
     * SCALE_X_2 / 5.2 /, SCALE_Y_2 / 5.2 /,
     * DUMMY_SCALE / 1.0 /

C     The house's rotation is 180 degrees and the tree's rotation
C     is -20 degrees.
      HOUSE_ROTATION = 3.14
      TREE_ROTATION = -3.14/9.0

C     Obtain the workstation type.
      CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
     * WS_TYPE, DUMMY_INTEGER )

C     Find out if changes to the segment attributes require
C     implicit regenerations or if changes occur immediately...
      CALL GKS$INQ_DYN_MOD_SEG_ATTB( WS_TYPE, ERROR_STATUS,
     * SEG_XFORM, VIS_TO_INVIS, DUMMY_INTEGER, HIGH_CHANGE,
     * PRIOR_CHANGE, DUMMY_INTEGER, DUMMY_INTEGER )

C     Establish an identity segment transformation...
      CALL GKS$EVAL_XFORM_MATRIX( DUMMY_REAL, DUMMY_REAL,
     * DUMMY_REAL, DUMMY_REAL, DUMMY_REAL, DUMMY_SCALE,
     * DUMMY_SCALE, GKS$K_COORDINATES_NDC, IDENTITY )

C     Flip the house onto its roof...
      CALL GKS$EVAL_XFORM_MATRIX( HOUSE_FIXED_X, HOUSE_FIXED_Y,
     * DUMMY_REAL, DUMMY_REAL, HOUSE_ROTATION, DUMMY_SCALE,
     * DUMMY_SCALE, GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
      CALL GKS$SET_SEG_XFORM( HOUSE, HOUSE_XFORM_MATRIX )

C     Shrink the tree...
      CALL GKS$EVAL_XFORM_MATRIX( TREE_FIXED_X, TREE_FIXED_Y,
     * DUMMY_REAL, DUMMY_REAL, DUMMY_REAL, SCALE_X_1,
     * SCALE_Y_1, GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
      CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )

C     Move the title...
      CALL GKS$EVAL_XFORM_MATRIX( DUMMY_REAL, DUMMY_REAL,
     * VECTOR_X, VECTOR_Y, DUMMY_REAL, DUMMY_SCALE,
     * DUMMY_SCALE, GKS$K_COORDINATES_NDC, TITLE_XFORM_MATRIX )
      CALL GKS$SET_SEG_XFORM( TITLE, TITLE_XFORM_MATRIX )
```

(continued on next page)

```
C     Pause.  Type RETURN when finished viewing the picture.
      IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF

C     By accumulating the tree's matrix, we can add to the
C     translation increment by increment...
      CALL GKS$ACCUM_XFORM_MATRIX( TREE_XFORM_MATRIX,
     * TREE_FIXED_X, TREE_FIXED_Y, DUMMY_REAL, DUMMY_REAL,
     * TREE_ROTATION, SCALE_X_2, SCALE_Y_2,
     * GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
      CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )

C     Pause.  Type RETURN when finished viewing the picture.
      IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF

C     By accumulation, shift the tree's X coordinate points by 0.15...
      CALL GKS$ACCUM_XFORM_MATRIX( TREE_XFORM_MATRIX,
     * DUMMY_REAL, DUMMY_REAL, 0.15, 0.0, DUMMY_REAL,
     * DUMMY_SCALE, DUMMY_SCALE, GKS$K_COORDINATES_NDC,
     * TREE_XFORM_MATRIX )
      CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )

C     Pause.  Type RETURN when finished viewing the picture.
      IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF

C     Return the tree to its original size and position...
      CALL GKS$SET_SEG_XFORM( TREE, IDENTITY )

C     Pause.  Type RETURN when finished viewing the picture.
      IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF
```

(continued on next page)

Example 5–1 (Cont.): Using DEC GKS Output Functions

```
C     Shift the house past its normalization viewport boundary to show
C     how segments are clipped...
      CALL GKS$ACCUM_XFORM_MATRIX( HOUSE_XFORM_MATRIX,
     * DUMMY_REAL, DUMMY_REAL, 0.1, 0.0, DUMMY_REAL, DUMMY_SCALE,
     * DUMMY_SCALE, GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
      CALL GKS$SET_SEG_XFORM( HOUSE, HOUSE_XFORM_MATRIX )

C     Pause.  Type RETURN when finished viewing the picture.
      IF ( SEG_XFORM .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF

C     Inquire about the segment priority capabilities...
      CALL GKS$INQ_SEG_PRIORITY( WS_ID, ERROR_STATUS,
     * NUM_PRIORITIES )

C     Give the land a higher priority than the house...
      IF ( NUM_PRIORITIES .EQ. 0 ) THEN
           CALL GKS$SET_SEG_PRIORITY( HOUSE, 0.1 )
           CALL GKS$SET_SEG_PRIORITY( HORIZON, 0.2 )
      ELSE
           CALL GKS$SET_SEG_PRIORITY( HOUSE, 0.1 )
           CALL GKS$SET_SEG_PRIORITY( HORIZON, 1.0 )
      ENDIF

C     Pause.  Type RETURN when finished viewing the picture.
      IF ( PRIOR_CHANGE .EQ. GKS$K_IRG ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
           READ(5,*)
      ELSE
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)
      ENDIF

C     Change some of the segment attributes...
      CALL GKS$SET_SEG_HIGHLIGHTING( TREE, GKS$K_HIGHLIGHTED )
      CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_INVISIBLE )
```

(continued on next page)

Example 5–1 (Cont.): Using DEC GKS Output Functions

```
C     Pause.  Type RETURN when finished viewing the picture.
      IF (( HIGH_CHANGE .EQ. GKS$K_IRG ) .OR.
     *   ( VIS_TO_INVIS .EQ. GKS$K_IRG )) THEN
          CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
          READ(5,*)
      ELSE
          CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
          READ(5,*)
      ENDIF

      RETURN
      END
```

Chapter 6
Requesting Input

The DEC GKS input process allows the application program to obtain data from a user. The data is classified according to the DEC GKS data types (coordinate points, segment names, real numbers, and integer values). The input process can either operate synchronously or asynchronously with the application program.

This chapter discusses general input concepts and the DEC GKS request input mode. Request mode allows the application program and the input process to operate synchronously, causing the application program to pause until the user responds to the input prompt. The two asynchrounous input operating modes, sample and event mode, are described in Chapter 7, Sampling Input and Generating Events.

This chapter discusses the following concepts in detail:

- Logical input devices
- Input requests
- Input initialization
- Prompt and echo types
- Data records
- Input viewport priority

NOTE

Section 6.8 contains the code that you must add to the Starry Night program in Example 3–2 to produce the program example contained in this chapter. You may wish to add this code to the base program so that you can execute the program while reading this chapter. The lines of blue code in the example signify the new code that you need to add to Example 3–2.

6.1 Logical Input Devices

When you input information using DEC GKS, you use two types of related devices. These devices are the *physical input devices* and the *logical input devices.*

The physical input devices on a workstation can be numerous and varied. A single workstation can use a keyboard, a mouse, a tablet, and cross-hairs to input information. Depending on the software specifications, you can use any combination of physical input devices to specify input. For instance, on a given workstation, you can type a text string using the keyboard, and signal the end of input by pressing a button on the mouse.

Since there is a wide variety of physical input devices, DEC GKS maintains device independence by using logical input devices. A logical input device is DEC GKS software that accepts input of *one* data type, from *one* open workstation, using only *one* combination of physical input devices. With care, you can use logical input devices without worry about the differences in physical input devices on various workstations.

DEC GKS logical input devices consist of the following three components:

- Workstation identifier
- Logical device number
- Input class

The workstation identifier specifies the particular workstation on which to input information. When DEC GKS prompts for input, it needs to know the workstation's physical capabilities, and it needs to access information in the workstation description table and state list.

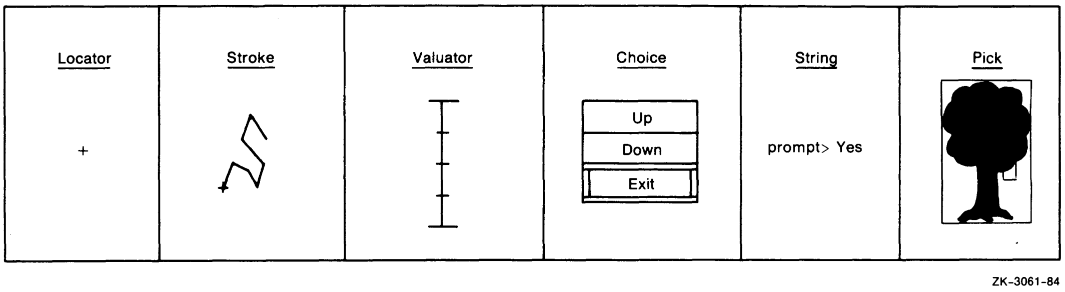
The *input class* specifies the data type of the input information. The DEC GKS input classes and return data types are as follows:

- Locator—Accepts a device coordinate point (two real numbers), transforms that point, and returns a corresponding world coordinate value.
- Stroke—Accepts a series of device coordinate points (pairs of real numbers), transforms those points, and returns a corresponding series of world coordinate points.
- Valuator—Accepts a real value from a specified range (some graphics handlers may prompt the user with a picture that looks like a radio dial with a pointer to the current value).
- Choice—Accepts an integer value that specifies a choice (some graphics handlers may prompt the user with a menu with the current choice highlighted).

- String—Accepts a character string.
- Pick—Accepts a device coordinate point (two real numbers), transforms that point, and returns the name of the segment (an integer) whose primitive(s) contain that point.

Figure 6–1 illustrates the visual prompts that a graphics handler may use to implement the DEC GKS logical input classes. For each logical input class, you choose from the graphic handlers' available prompt and echo types. Each graphic handler can support a different number of prompt and echo types for a given input class. Section 6.3 discusses prompt and echo types in detail.

Figure 6–1: Possible Prompts for DEC GKS Logical Input Classes



The *logical device number* differentiates between different physical input devices used to enter the same class of data on the same workstation. For instance, a workstation may use both a mouse and the arrow keys on a keyboard for two distinct stroke logical input devices. You can distinguish between the two different logical input devices by their logical device numbers. For instance, the graphics handler could assign the logical device number 1 to the stroke device using the mouse, and could assign the number 2 to the stroke device using the arrow keys on the keyboard. When you request input on such a workstation, you specify whether you wish to use stroke logical input device 1 or 2 (the mouse or the keyboard).

6.2 DEC GKS Input Modes

As a level 2c implementation, DEC GKS offers three input operating modes as follows:

- Request mode
- Sample mode
- Event mode

Request mode allows the application program to operate synchronously with the input process. This chapter uses only request mode functions to illustrate the general DEC GKS input concepts.

Using sample and event modes, the application program and the input process operate asynchronously, allowing the application program to continue processing while the user enters input values. Chapter 7, Sampling Input and Generating Events, describes sample and event mode in detail, and illustrates instances when one operating mode may be more appropriate than the others.

6.2.1 Requesting Input from Logical Input Devic

In request mode, the application pauses, and DEC GKS waits for the user to signal the end of input (this process is called *triggering*) or to signal a *break*. You can envision the application process and the request input process as operating synchronously.

When altering the values of a given logical input device by manipulating the prompt, the user changes the device's *measure*. When the user triggers the device, the handler writes the measure at the time of triggering to the application. A user can also signify "no returned input" to the application by performing a break. Each graphics handler defines the way in which a user breaks the input process for a given input class.

To request input, you call one of the functions GKS$REQUEST_LOCATOR, GKS$REQUEST_STROKE, GKS$REQUEST_VALUATOR, GKS$REQUEST_CHOICE, GKS$REQUEST_STRING, or GKS$REQUEST_PICK. Once you call one of these functions, the input prompt for the specified logical input device appears on the surface of the workstation. The following code illustrates a call to GKS$REQUEST_LOCATOR.

```
          .
          .
          .
  C    Make sure that the device supports locator input...
❶      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * NUM_LOCATOR_DEVICES, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER )

       IF ( NUM_LOCATOR_DEVICES .EQ. 0 ) THEN
            WRITE(6,*) 'The workstation does not support'
            WRITE(6,*) 'locator input.'
            STOP
       ENDIF

❷      CALL GKS$REQUEST_LOCATOR( 1, 1, INPUT_STATUS,
     * XFORM, WORLD_COORD_X, WORLD_COORD_Y )
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

❶ These lines of code check to make sure that the workstation type supports at least one device. If the workstation type does not support a locator device, then this code stops program execution.

❷ This call to GKS$REQUEST_LOCATOR requests input from the open workstation identified by the integer value 1, and from the locator logical device number 1. The workstation identifier is 1; the logical input class is locator; and, the logical device number is 1.

The call to GKS$REQUEST_LOCATOR uses the default prompt and echo type that is determined by the graphics handler. After the user triggers or breaks input in a manner determined by the default prompt and echo type, GKS$REQUEST_LOCATOR writes the following values to its arguments:

Argument	**Written Value**
INPUT_STATUS	This argument specifies whether the user returned valid input (GKS$K_STATUS_OK) or a break in the input process (GKS$K_STATUS_NONE). Choice and pick devices can return an additional value in this argument. If the user triggers input without moving through the choices or without picking a valid segment, this argument specifies that the user does not want to input a value (GKS$K_STATUS_NOCHOICE or GKS$K_STATUS_NOPICK).

Argument	Written Value
XFORM	When transforming the device coordinate to NDC points, DEC GKS uses the current workstation window and viewport. However, when transforming a point from the NDC space to world coordinate space, DEC GKS must use a normalization viewport corresponding to the input viewport priority. This operation is crucial when the returned device coordinate point transforms to a point on the NDC space where two or more normalization viewports overlap. After requesting input, this argument specifies the normalization transformation number that corresponds to the normalization viewport and window used to transform the NDC point to a world coordinate point. Section 6.7 discusses input viewport priority in detail.
WORLD_COORD_X WORLD_COORD_Y	These arguments specify the X and Y components of the world coordinate point corresponding to the device coordinate point that the user entered.

Depending on the needs of your application, you may wish to control whether DEC GKS echos the input on the workstation surface. For instance, you may wish to enter a point on the workstation surface without seeing cross hairs, tracking plus signs or any other prompt on the surface. To eliminate prompting, you can call one of the SET MODE functions.

The following code illustrates how to control the echoing of input:

```
        .
        .
        .
  CALL GKS$SET_LOCATOR_MODE( 1, 1,
 * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

  CALL GKS$REQUEST_LOCATOR( 1, 1, INPUT_STATUS,
 * XFORM, WORLD_COORD_X, WORLD_COORD_Y )
        .
        .
        .
```

When you call GKS$SET_LOCATOR_MODE, you pass the workstation identifier and the device number. The argument GKS$K_INPUT_MODE_REQUEST specifies that the input device operates in request mode. Whenever you call one of the SET MODE functions, you must pass GKS$K_INPUT_MODE_REQUEST as the third argument. Otherwise, you generate an error.

The argument GKS$K_ECHO specifies that you want the prompt to echo on the workstation surface. This is the default setting.

For more information concerning the SET MODE functions, refer to the *DEC GKS Reference Manual*.

6.3 Prompt and Echo Types

Many times, you may find that the default prompt and echo type does not suit your application. For instance, you may wish to have the prompt for the choice logical input device look like a menu; you may want control of the number of items in a menu; and, you may want to label the menu items so that the labels apply to your application.

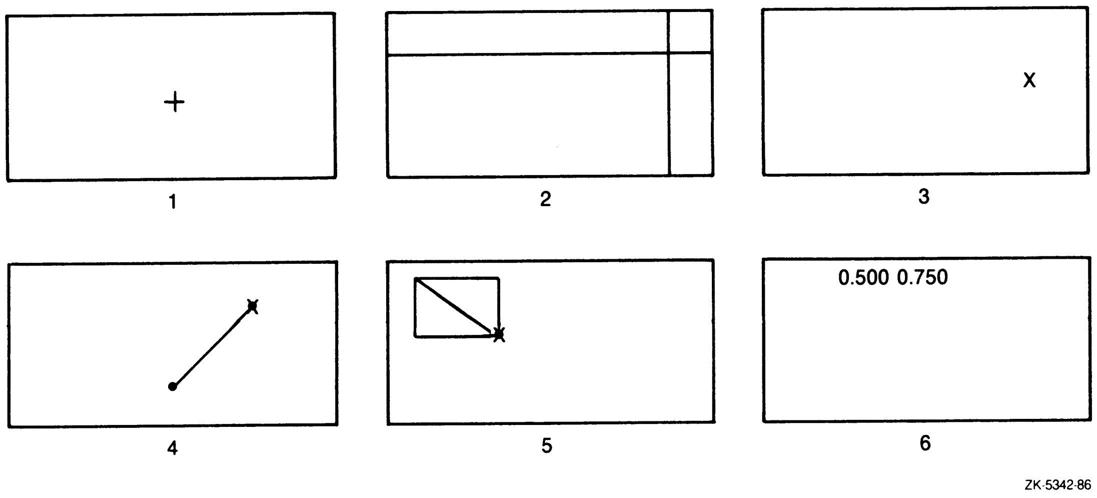
To have greater control of the input prompt, you need to choose from one of the prompt and echo types available on your workstation. The prompt and echo types determine the different visual interfaces used to prompt the user for a given logical input device.

For example, the GKS standard specifies six different prompt and echo types for a locator input device. A graphics handler may implement any of the following locator prompts:

1. A tracking plus sign (+)
2. A cross hair
3. A tracking cross (X)
4. A line from the initial locator position to the current locator position
5. A rectangle whose diagonal connects the initial and current positions
6. A numeric representation of the current locator position

Figure 6–2 illustrates possible implementations of the six standard prompt and echo types.

Figure 6–2: Possible Locator Prompts



Of the prompt and echo types defined by the GKS standard, a graphics handler may implement any number of them. If you are unaware of the capabilities of your device, or if you write an application that runs on several devices of varying prompt and echo type support, you need to inquire about the supported prompt and echo types of each device. You can choose from the prompt and echo types available on a particular workstation.

To choose a prompt and echo type other than the default, you must pass the desired prompt and echo type number to the appropriate INITIALIZE input function. Section 6.4 contains an example of a call to GKS$INIT_PICK.

For a complete description of the GKS standard prompt and echo types for each class of logical input device, refer to Chapter 8, Input Functions, in the *DEC GKS Reference Manual.*

6.3.1 Data Records

Since the graphics handlers use DEC GKS primitives such as lines, markers, and fill areas to construct input prompts, the graphics handler optionally uses additional information that determines how the prompt and echoed input appears on the surface. For instance, a handler may use a polyline output attribute that would affect the appearance of cross hairs on the surface. The requirements depend on the needs of the different prompt and echo types on different physical devices.

To pass information to meet the requirements of a certain prompt and echo type on a given logical input device, you adjust components of the *input data record*. The input data record is a series of components, contiguous in memory, that specify additional information needed to implement a certain prompt interface (according to a prompt and echo type value).

The GKS standard establishes a data record for all of the prompt and echo types for each logical input class. For instance, the GKS standard specifies that stroke prompt and echo type 4 is a line connecting stroke points that the user enters. The GKS standard also specifies a standard 6-component/13-component input data record for a stroke logical input device prompt and echo type 4.

The following table contains the GKS standard input data record for stroke prompt and echo type 4. The column marked "Required" specifies whether all GKS graphics handlers requires (R) or does not require (N) a component. A graphics handler can optionally use any of the remaining components. The standard data record for stroke prompt and echo type 4 is as follows:

Standard Data Record:

Position	Data Type	Required	Description
1	Integer	R	Input buffer size, in number of stroke points.
2	Integer	N	Editing position expressed as a stroke point.
3	Real	N	X world coordinate change vector.
4	Real	N	Y world coordinate change vector.
5	Real	N	Time interval, in seconds.
6	Integer	N	Attribute control flag. GKS$K_ACF_CURRENT (0) or GKS$K_ACF_SPECIFIED (1). Use the currently set output attributes or specify new attributes in this data record.

If component 6 is GKS$K_ACF_SPECIFIED:

Position	Data Type	Required	Description
7	Integer	N	Line type aspect source flag. GKS$K_ASF_BUNDLED (0) or GKS$K_ASF_INDIVIDUAL (1).
8	Integer	N	Line width scale factor aspect source flag. GKS$K_ASF_BUNDLED (0) or GKS$K_ASF_INDIVIDUAL (1).

Position	Data Type	Required	Description
9	Integer	N	Polyline color index aspect source flag. GKS$K_ASF_BUNDLED (0) or GKS$K_ASF_INDIVIDUAL (1).
10	Integer	N	Polyline index.
11	Integer	N	Line type index.
12	Real	N	Line width scale factor.
13	Integer	N	Polyline color index.

In order to compare the difference between a GKS standard data record and a given graphics handler's implementation of a data record, the following list presents the data record for stroke prompt and echo type 4 on the VAXstations.

The column labeled "Used" specifies whether the VAXstation graphics handler uses (U) or ignores (I) a given component of the stroke data record. As stated in the GKS standard data record description, the VAXstation graphics handler *must* use component number 1, the stroke buffer. The VAXstation handler chooses to use components 2, 3, 4, and 6, but ignores component 5, and components 7 through 13, if passed in the data record. The VAXstation input data record for stroke prompt and echo type 4 is as follows.

VAXstation Data Record:

Position	Data Type	Used	Description
1	Integer	U	Input buffer size, in number of stroke points.
2	Integer	U	Editing position expressed as a stroke point.
3	Real	U	X world coordinate change vector.
4	Real	U	Y world coordinate change vector.
5	Real	I	Time interval, in seconds.
6	Integer	U	Attribute control flag. GKS$K_ACF_CURRENT (0) or GKS$K_ACF_SPECIFIED (1). Use the currently set output attributes or specify new attributes in this data record.

If component 6 is GKS$K_ACF_SPECIFIED, you must pass the following components:

Position	Data Type	Used	Description
7	Integer	I	Line type aspect source flag. GKS$K_ASF_BUNDLED (0) or GKS$K_ASF_INDIVIDUAL (1).
8	Integer	I	Line width scale factor aspect source flag. GKS$K_ASF_BUNDLED (0) or GKS$K_ASF_INDIVIDUAL (1).
9	Integer	I	Polyline color index aspect source flag. GKS$K_ASF_BUNDLED (0) or GKS$K_ASF_INDIVIDUAL (1).
10	Integer	I	Polyline index.
11	Integer	I	Line type index.
12	Real	I	Line width scale factor.
13	Integer	I	Polyline color index.

You establish data record values by initializing a device. To initialize a device, you pass the data record to the appropriate GKS$INIT_class function. (Section 6.4 contains an example of a call to GKS$INIT_PICK.) To initialize a logical input device, you must make sure that the device's prompt is not currently on the workstation surface. To make sure that the prompt is removed, you call one of the GKS$SET_class_MODE functions to set the device to request mode.

For more information concerning the stroke data record components, refer to Chapter 6, Input Functions, in the *DEC GKS Reference Manual.* For more information concerning the VAXstation logical input devices, refer to Chapter 1, VAXstation Workstation Specifics in the *DEC GKS Device Specific Reference Manual.*

6.3.2 Inquiry Functions and Data Record Buffer Sizes

When you decide to change the default input values by calling one of the GKS$INIT_class input functions, you need to pass a valid data record. To obtain a valid data record, you can either construct the record according to the GKS standard data record specifications for your chosen prompt and echo type, or you can call an inquiry function to obtain the default (or currently specified) data record.

If you choose to call an inquiry function, you must use caution when defining data record buffer sizes. The buffer size is a modifiable variable (read/write), and when passed to the inquiry function, must contain the exact size of the buffer in order for the inquiry function to properly return the contents of the data record. DEC GKS requires that you pass a data record at least as large as the record specified for your chosen prompt and echo type. For instance, if you choose valuator prompt and echo type number 2, the GKS standard specifies that the data record must be 2 components (a total size of 8 bytes). When you call GKS$INIT_VALUATOR, your data record buffer must be at least 8 bytes long, and you must specify to the function that 8 bytes of the buffer contain the data record components (in some cases, the data record may not fill your declared buffer).

To obtain input data records and other values, you can either inquire from the workstation description table (default values) or from the workstation state list (current values). To obtain default values, you call the functions GKS$INQ_DEF_LOCATOR_DATA, GKS$INQ_DEF_STROKE_DATA, and so forth. To obtain the current values, you call the functions GKS$INQ_LOCATOR_STATE, GKS$INQ_STROKE_STATE, and so forth.

After the function call, the graphics handler writes the amount of the buffer actually used to the buffer size argument. You can compare this value to the data record size to see if the entire data record fits in the buffer, or to see if the graphics handler truncates the record when writing it to the buffer. If the graphics handler truncates the data record, you need to decide whether to continue execution or to alter the buffer size so that the entire data record fits.

The following is an example of a call to GKS$INQ_LOCATOR_STATE:

```
              .
              .
              .
        INTEGER WS_ID, ERROR_STATUS, INPUT_MODE, ECHO_FLAG,
       * XFORM, RECORD_BUFFER_LENGTH, RECORD_SIZE,
       * INPUT_STATUS, PROMPT_ECHO_TYPE, DATA_SIZE, DEVICE_NUM
❶       INTEGER  DATA_RECORD( 1 )
        REAL WORLD_X, WORLD_Y, ECHO_AREA( 4 )
        DATA WS_ID / 1 /, DEVICE_NUM / 1 /

        CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
        CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_VT240 )
        CALL GKS$ACTIVATE_WS( WS_ID )

    C   The argument RECORD_BUFFER_LENGTH is four bytes long...
❷       RECORD_BUFFER_LENGTH = 4

    C   Inquire about the current values...
❸       CALL GKS$INQ_LOCATOR_STATE( WS_ID, DEVICE_NUM,
       * GKS$K_VALUE_REALIZED, ERROR_STATUS, INPUT_STATUS,
       * ECHO_FLAG, XFORM, WORLD_X, WORLD_Y, PROMPT_ECHO_TYPE,
       * ECHO_AREA, DATA_RECORD, RECORD_BUFFER_LENGTH,
       * RECORD_SIZE )
```

```
IF ( RECORD_BUFFER_LENGTH .LT. RECORD_SIZE ) THEN
    WRITE(6,*) 'DEC GKS wrote only part of the record'
    WRITE(6,*) 'to the buffer.'
ENDIF
     .
     .
     .
```

The following numbers correspond to the numbers in the previous example:

❶ The locator prompt and echo type does not use the data record. This is a dummy variable.

❷ You must specify the input data record buffer size (4 bytes) before you call GKS$INQ_LOCATOR_STATE, GKS$INQ_STROKE_STATE, and so forth. After the function call, the argument RECORD_BUFFER_LENGTH contains the value 0 (the amount of the buffer containing the data record).

❸ The argument GKS$K_VALUE_REALIZED tells the graphics handler to pass the input values as they are implemented, as opposed to the way that the application may have previously set the values (GKS$K_VALUE_SET).

The functions GKS$INQ_DEF_CHOICE_DATA and GKS$INQ_CHOICE_STATE functions are designed so that you can call them twice. If you pass the value 0 in the first component of the data record, these inquiry functions only write the number of default or current choices back to the first component of the data record and ignore all other arguments.

Before you call the function a second time, you need to perform the following steps:

- Check to make sure that your buffers can hold the number of choice strings returned in the first call.
- Initialize the second component of your choice data record so that it contains the address of the array containing the sizes of your allocated string buffers.
- Initialize the third component of your data record so that it contains the address of the array containing the addresses of your string buffers. The graphics handler only uses as much of the string buffer as specified in the array of string sizes to write the choice strings.

When you call these inquiry functions a second time, DEC GKS performs the following tasks:

- Writes values to all output arguments.
- Writes the sizes of the returned strings in the array whose address is located in the second component of the data record.

- Uses the address in the third component of the data record to locate the array of string addresses, and uses the string addresses to write the strings into the buffers.

If you do not establish the correct pointers, your program generates errors. To see an example of two calls to these inquiry functions in a single program, refer to Chapter 12, Inquiry Functions, in the *DEC GKS Reference Manual*. For more information on the remaining input inquiry functions, refer to the program examples in this chapter.

6.4 Requesting Pick Input

Since the program examples in this book use segments to store primitives, it can be useful to require the user to tell the application which segment to alter. You can do this by initializing and requesting pick input.

Pick input allows the user to move a cursor, or *aperture*, on the surface of the workstation. When the aperture comes in contact with a primitive in a segment, the graphics handler highlights the segment in a device-dependent manner. When the user triggers input, the pick logical input device returns the segment name (DEC GKS uses integer values as segment names), and the *pick identifier* of the picked primitive.

The pick identifier is a primitive output attribute that allows you to define another level of naming individual primitives within a single segment. At the time of primitive generation, DEC GKS assigns the current pick identifier integer value to the primitive. If you want to assign a primitive a different pick identifier value, you must call GKS$SET_PICK_ID before generating the primitive. DEC GKS assigns the new pick identifier value to all subsequently generated primitives. For examples illustrating the use of pick identifiers, refer to Chapter 9, Segment Functions, in the *DEC GKS Reference Manual*.

The code that you have to add to the DRAW_PICTURE subroutine in Example 3–2 to use DEC GKS pick logical input devices is as follows:

```
    C      ************************************************************
    C      Draw the picture, and place each primitive in a segment...
           SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
          * ROAD, HOUSE, HORIZON )
             .
             .
             .
           CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
           CALL GKS$CLOSE_SEG()

    C      Ask the user for input...
❶         CALL GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS, TREE,
          * SIDE, ROAD, HOUSE, HORIZON )
```

```
        RETURN
        END

C       ***********************************************************
C       Coordinate user input...
        SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
      * TREE, SIDE, ROAD, HOUSE, HORIZON )

        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
      * HORIZON, CATEGORY, ERROR_STATUS, FILL_PTS, FOREGROUND,
      * BACKGROUND, UNITY, PICKED_SEGMENT, WS_TYPE
        REAL FILL_X( 5 ), FILL_Y( 5 )

        DATA FILL_PTS / 5 /, FOREGROUND / 1 /, BACKGROUND / 0 /,
      * UNITY / 0 /

❷       DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
        DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

❸       CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
      * ERROR_STATUS, CATEGORY )

        IF ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ) THEN
             WRITE(6,*) 'The workstation does not support'
             WRITE(6,*) 'both input and output.'
             STOP
        ENDIF

C       Make sure that you are using the unity transformation...
        CALL GKS$SELECT_XFORM( UNITY )

C       Fill an area on which to send the user a message...
❹       CALL GKS$SET_FILL_COLOR_INDEX( FOREGROUND )
        CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )

C       Write the message...
        CALL GKS$SET_TEXT_HEIGHT( 0.028 )
        CALL GKS$SET_TEXT_COLOR_INDEX( BACKGROUND )
        CALL GKS$TEXT( 0.05, 0.25,
      * 'Which segment would you like to scale?' )
        CALL GKS$TEXT( 0.05, 0.2,
      * 'Move the cursor, outline your chosen' )
        CALL GKS$TEXT( 0.05, 0.15,
      * 'segment, and then trigger input.' )
        CALL GKS$TEXT( 0.05, 0.02,
      * '(Press RETURN when ready.)' )
```

```
C     The user presses RETURN when ready to pick...
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

C     Erase the message and redraw the segments...
❺     CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

C     Make sure that all of the segments are detectable...
❻     CALL GKS$SET_SEG_DETECTABILITY( TITLE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( STARS, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( TREE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( SIDE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( ROAD, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HOUSE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HORIZON, GKS$K_DETECTABLE )

C     Initialize and request pick input...
❼     CALL PICK_IT( WS_ID, WS_TYPE, PICKED_SEGMENT )
          .
          .
          .
      RETURN
      END
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

❶ This code calls subroutine GO_FOR_INPUT. To initialize pick input, you need to pass the segment names as arguments. You can use the arguments WS_ID and WS_TYPE when calling inquiry functions.

❷ This code defines a fill area border at the bottom of the default normalization window. After calling GKS$FILL_AREA, you have an area of the workstation surface on which to output messages to the user.

❸ The call to GKS$INQ_WS_CATEGORY returns the workstation category. To perform pick input, you must be working with a workstation of category GKS$K_WSCAT_OUTIN.

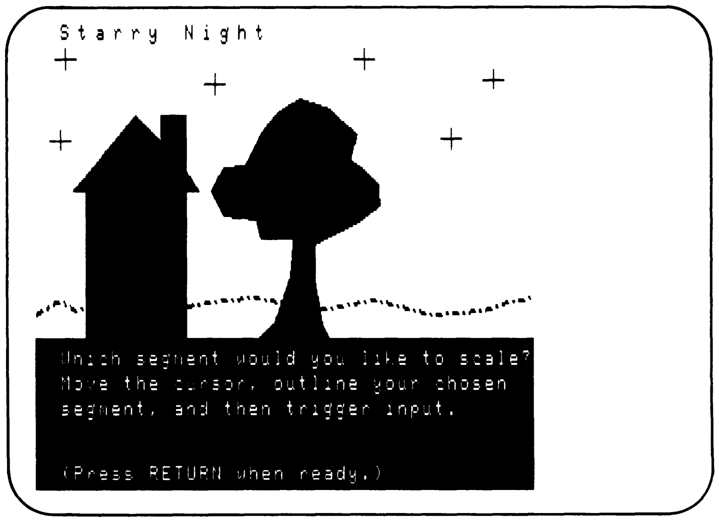
❹ This code outputs a message telling the user to press RETURN when ready to pick a segment. Figure 6–3 illustrates the workstation surface after execution of this code. Remember that the figure illustrates output from a VT241 terminal and that the picture you see on the surface of your workstation may differ.

❺ At this point in the program, you need to call GKS$REDRAW_SEG_ON_WS to update the surface of the workstation. A call to GKS$UPDATE_WS passing the argument GKS$K_PERFORM_FLAG does not work since you have not requested a change that switches the *new frame necessary at update* DEC GKS state list flag to YES (possibly changes such as workstation transformations, segment transformations, and so forth). GKS$REDRAW_SEG_ON_WS clears the surface and redraws all segments despite the setting of the *new frame* flag.

❻ This code sets the segment detectability attribute so that the user can pick any of the defined segments. Segments must be both visible and detectable in order to be picked. By default, segments are not detectable.

❼ The subroutine PICK_IT contains the code that initializes and requests pick input.

Figure 6–3: Sending a Message to the User—VT241



```
Which segment would you like to scale?
Move the cursor, outline your chosen
segment, and then trigger input.

(Press RETURN when ready.)
```

ZK-5310-86

The following code example presents the subroutine PICK_IT:

```
C       ***************************************************************
C       This function allows a user to pick a segment...
        SUBROUTINE PICK_IT( WS_ID, WS_TYPE, PICKED_SEGMENT )

        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, INITIAL_STATUS, PICKED_SEGMENT,
     * PICK_ID, PROMPT_ECHO_TYPE, ERROR_STATUS,
     * INPUT_MODE, ECHO_FLAG, RECORD_BUFFER_LENGTH,
     * RECORD_SIZE, INPUT_STATUS, DEVICE_NUM,
     * INPUT_CHOICE, DUMMY_INTEGER, DATA_RECORD( 10 ),
     * NUM_PICK_DEVICES, FILL_PTS, STATUS, WS_TYPE
        REAL ECHO_AREA( 4 ), FILL_X( 5 ), FILL_Y( 5 )
        DATA DEVICE_NUM / 1 /, FILL_PTS / 5 /

        DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
        DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /
```

```
   C     Make sure that the device supports pick input...
❶        CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
       * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
       * DUMMY_INTEGER, NUM_PICK_DEVICES, DUMMY_INTEGER )

         IF ( NUM_PICK_DEVICES .EQ. 0 ) THEN
             WRITE(6,*) 'The workstation does not support'
             WRITE(6,*) 'pick input.'
             STOP
         ENDIF

   C     Give the data record the size of your data record buffer and
   C     inquire about the realized pick values.
❷        RECORD_BUFFER_LENGTH = 40
         CALL GKS$INQ_PICK_STATE( WS_ID, DEVICE_NUM,
       * GKS$K_VALUE_REALIZED, ERROR_STATUS, INPUT_MODE,
       * ECHO_FLAG, INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
       * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
       * RECORD_BUFFER_LENGTH, RECORD_SIZE )

   C     Make sure that the data record was not truncated...
❸        IF ( RECORD_SIZE .LT. RECORD_BUFFER_LENGTH ) THEN
             WRITE(6,*) 'The data record was truncated.'
             WRITE(6,*) 'Declare a larger buffer.'
             STOP
         ENDIF
   C     Allow entrance into the loop...
❹        INPUT_STATUS = GKS$K_STATUS_NOPICK
   C     Loop until the user picks a segment...
❺        DO WHILE ( ( INPUT_STATUS .EQ. GKS$K_STATUS_NOPICK ) .OR.
       *            ( INPUT_STATUS .EQ. GKS$K_STATUS_NONE ))

   C     Make sure that the pick aperture is not placed on any segment...
         INITIAL_STATUS = GKS$K_STATUS_NOPICK

   C     Since the device is in request mode by default, initialize the device...
❻        CALL GKS$INIT_PICK( WS_ID, DEVICE_NUM, INITIAL_STATUS,
       * PICKED_SEGMENT, PICK_ID, PROMPT_ECHO_TYPE, ECHO_AREA,
       * DATA_RECORD, RECORD_BUFFER_LENGTH )

   C     Make sure that echo is enabled...
❼        CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
       * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

   C     Request input. PICKED_SEGMENT contains the chosen segment's
   C     name.
❽        CALL GKS$REQUEST_PICK( WS_ID, DEVICE_NUM,
       * INPUT_STATUS, PICKED_SEGMENT, PICK_ID )

   C     Send a message to the user if a segment is not picked...
         IF ( INPUT_STATUS .EQ. GKS$K_STATUS_NOPICK ) THEN
             CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
             CALL GKS$TEXT( 0.05, 0.25,
       *          'I cannot let you go until you pick' )
             CALL GKS$TEXT( 0.05, 0.2,
       *          'a segment!' )
             CALL GKS$TEXT( 0.05, 0.02,
       *          '(Press RETURN when ready.)' )
```

```
C     The user presses RETURN when ready to pick again...
          CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
          READ(5,*)

C     Erase the message and redraw the segments...
          CALL GKS$REDRAW_SEG_ON_WS( WS_ID )
      ENDIF
      ENDDO

      RETURN
      END
         .
         .
         .
```

The following numbers correspond to the numbers in the previous example:

❶ The call to GKS$INQ_INPUT_DEV obtains the number of pick devices supported by the workstation. If the number of devices is not at least one, then execution of this program ceases.

❷ The call to GKS$INQ_PICK_STATE obtains the current input values. In this example, the current values also happen to be the default values; the code has not reset the values in a previous call to GKS$INIT_PICK. The code contains GKS$INQ_PICK_STATE instead of GKS$INQ_DEF_PICK_DATA because inquiring about the pick state initializes all of the input values you need to call GKS$INIT_PICK.

This code uses the default pick prompt and echo type for the given workstation. Since the GKS standard does not define a pick data record, you do not know the size or contents of a data record required by a graphics handler. Consequently, this code declares a large, 10-component data record (as an integer array). The code initializes the modifiable argument RECORD_BUFFER_LENGTH to be the size of DATA_RECORD (40 bytes). After the call to GKS$INQ_PICK_STATE, RECORD_BUFFER_LENGTH contains the amount of the buffer containing the data record.

❸ This code checks to make sure that the graphics handler did not truncate the data record when writing it to the buffer. After the call to GKS$INQ_PICK_STATE, RECORD_SIZE contains the size of the data record and RECORD_BUFFER_LENGTH contains the amount of the buffer containing data record components. RECORD_SIZE should not be larger than RECORD_BUFFER_LENGTH.

❹ This code sets the initial input status to be GKS$K_STATUS_NOPICK. When you pass this status value to GKS$INIT_PICK, the graphics handler attempts to place the pick aperture on a portion of the surface not occupied by a segment. If the aperture does appear on a segment, the segment may or may not be highlighted, depending on the graphics handler. If you want to guide a user to an initially picked segment, you can specify the segment name and the status GKS$K_STATUS_OK to GKS$INIT_PICK.

If the user triggers input without highlighting a segment on the workstation surface using the pick aperture, the function GKS$REQUEST_PICK returns the status GKS$K_STATUS_NOPICK. This program does not allow a returned status of GKS$K_STATUS_NOPICK or GKS$K_STATUS_NONE (a break in input).

Figure 6–4 illustrates what happens when the user moves the aperture so that it does not touch a segment. Figure 6–5 illustrates the message sent to the user if the return status is GKS$K_STATUS_NOPICK or GKS$K_STATUS_NONE.

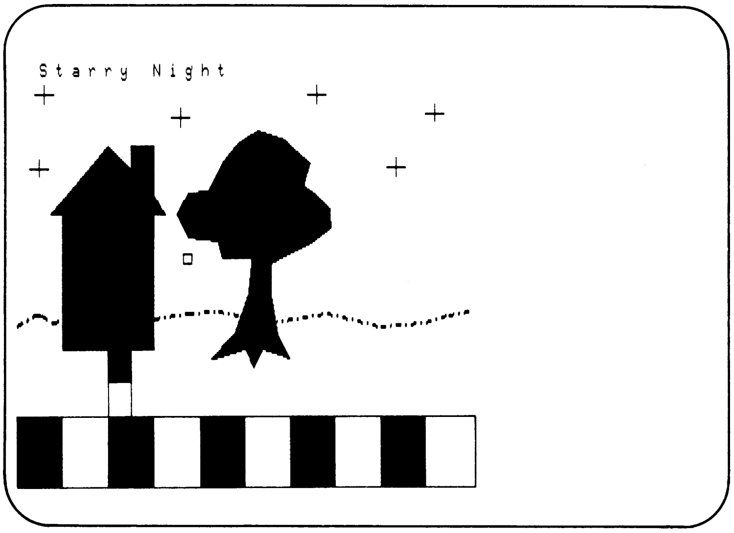
❺ This code creates a loop that continues until the user picks a segment. For this example, assume that the user picks the house. Figure 6–6 illustrates the highlighted house. To pick the house, the user needs to trigger input while the house is highlighted.

❻ The call to GKS$INIT_PICK establishes the default pick prompt and echo type, but assures an initial status of GKS$K_STATUS_NOPICK. (If you do not alter any of the initial status, you can just call GKS$REQUEST_PICK and use the default input values.) Notice that you pass the argument RECORD_BUFFER_LENGTH to GKS$INIT_PICK. You need to tell the graphics handler how much of the data record contains valid information. If your data record was truncated during the call to GKS$INQ_PICK_STATE, the RECORD_BUFFER_LENGTH argument to GKS$INIT_PICK would generate an error.

❼ This code assures that the graphics handler echos input on the workstation surface (the default situation). You do not need to call this function unless you want to turn off the echo.

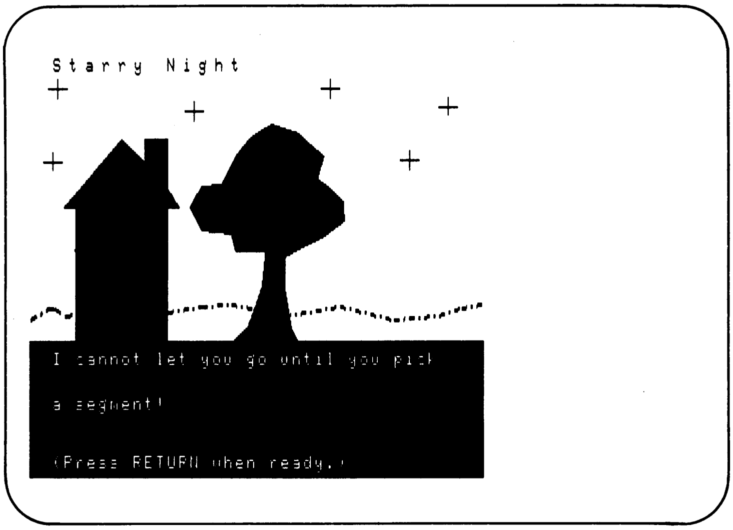
❽ The call to GKS$REQUEST_PICK initiates the pick input process. After the user triggers input, the value of the argument INPUT_STATUS contains either GKS$K_STATUS_NOPICK, GKS$K_STATUS_OK, or GKS$K_STATUS_NONE. The segment name written to PICKED_SEGMENT is not valid unless the input status is GKS$K_STATUS_OK.

Figure 6–4: Specifying GKS$K_STATUS_NOPICK—VT241



ZK-5320-86

Figure 6–5: Forcing the User to Pick a Segment—VT241


Starry Night

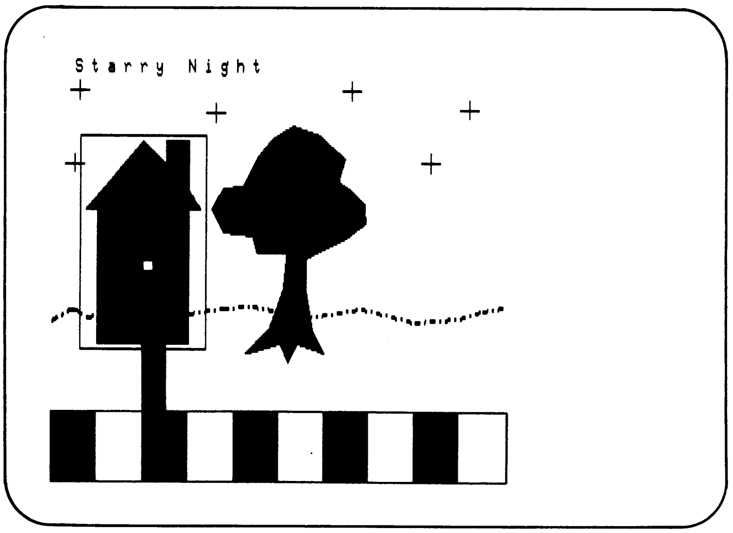
I cannot let you go until you pick

a segment!

(Press RETURN when ready.)

ZK-5315-86

Figure 6–6: Picking a Segment—VT241



ZK-5415-86

6.5 Requesting Valuator Input

The valuator logical input device returns a real number value. After the user picks a segment, you can use the valuator logical input device to retrieve a segment scaling value. The following code example performs this task.

```
C     ************************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )
          .
          .
          .
C     Scale the segment...
      CALL SPECIFY_VALUE( WS_ID, WS_TYPE, PICKED_SEGMENT,
     * TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON )
          .
          .
          .
      RETURN
      END
          .
          .
          .
```

```
   C     ************************************************************
   C     Specify a value for scaling...
         SUBROUTINE SPECIFY_VALUE( WS_ID, WS_TYPE, PICKED_SEGMENT,
        * TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON )

         IMPLICIT NONE
         INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
         INTEGER WS_ID, WS_TYPE, PICKED_SEGMENT,
        * TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON,
        * PROMPT_ECHO_TYPE, ERROR_STATUS, INPUT_MODE,
        * ECHO_FLAG, RECORD_BUFFER_LENGTH, RECORD_SIZE,
        * INPUT_STATUS, DEVICE_NUM, FILL_PTS, NUM_VAL_DEVICES,
        * DUMMY_INTEGER
         REAL ECHO_AREA( 4 ), DATA_RECORD( 2 ), UPPER_LIMIT,
        * LOWER_LIMIT, VALUE, FILL_X( 5 ), FILL_Y( 5 ),
        * FIXED_X, FIXED_Y, XFORM_MATRIX( 6 ), NULL, NO_CHANGE
         DATA DEVICE_NUM / 1 /, FILL_PTS / 5 /, NULL / 0.0 /,
        * NO_CHANGE / 1.0 /

         DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
         DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

   C     According to the standard, the elements in the data record are
   C     the upper and lower limits for all prompt and echo types.
❶        EQUIVALENCE( DATA_RECORD( 1 ), LOWER_LIMIT )
         EQUIVALENCE( DATA_RECORD( 2 ), UPPER_LIMIT )

   C     Make sure that the device supports valuator input...
❷        CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
        * DUMMY_INTEGER, DUMMY_INTEGER, NUM_VAL_DEVICES,
        * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER )

         IF ( NUM_VAL_DEVICES .EQ. 0 ) THEN
              WRITE(6,*) 'The workstation does not support'
              WRITE(6,*) 'valuator input.'
              STOP
         ENDIF

❸        RECORD_BUFFER_LENGTH = 8
         CALL GKS$INQ_VALUATOR_STATE( WS_ID, DEVICE_NUM,
        * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, VALUE,
        * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
        * RECORD_BUFFER_LENGTH, RECORD_SIZE )

❹        CALL GKS$EVAL_XFORM_MATRIX( NULL, NULL, NULL, NULL,
        * NULL, NO_CHANGE, NO_CHANGE, GKS$K_COORDINATES_NDC,
        * XFORM_MATRIX )

❺        VALUE = 1.0
         UPPER_LIMIT = 1.5
         LOWER_LIMIT = 0.5

   C     Write the message...
         CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
         CALL GKS$TEXT( 0.05, 0.25,
        * '1.5 increases the segment size 50%' )
         CALL GKS$TEXT( 0.05, 0.15,
        * '0.5 decreases the segment size 50%' )
         CALL GKS$TEXT( 0.05, 0.02,
        * '(Move indicator and trigger input.)' )
```

```
  C     Since the device is in request mode by default, initialize the device...
❻       CALL GKS$INIT_VALUATOR( WS_ID, DEVICE_NUM,
       * VALUE, PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
       * RECORD_BUFFER_LENGTH )

        CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
       * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

❼       CALL GKS$REQUEST_VALUATOR( WS_ID, DEVICE_NUM,
       * INPUT_STATUS, VALUE )

  C     Establish fixed points for segments depending on which
❽       IF ( PICKED_SEGMENT .EQ. TITLE ) THEN
           FIXED_X = 0.3
           FIXED_Y = 0.925
        ELSEIF ( PICKED_SEGMENT .EQ. STARS ) THEN
           FIXED_X = 0.5
           FIXED_Y = 0.8
        ELSEIF ( PICKED_SEGMENT .EQ. TREE ) THEN
           FIXED_X = 0.52
           FIXED_Y = 0.51
        ELSEIF ( PICKED_SEGMENT .EQ. SIDE ) THEN
           FIXED_X = 0.225
           FIXED_Y = 0.22
        ELSEIF ( PICKED_SEGMENT .EQ. ROAD ) THEN
           FIXED_X = 0.5
           FIXED_Y = 0.075
        ELSEIF ( PICKED_SEGMENT .EQ.  HORIZON ) THEN
           FIXED_X = 0.1
           FIXED_Y = 0.35
        ELSEIF ( PICKED_SEGMENT .EQ. HOUSE ) THEN
           FIXED_X = 0.2
           FIXED_Y = 0.5
        ENDIF

❾       CALL GKS$ACCUM_XFORM_MATRIX( XFORM_MATRIX, FIXED_X,
       * FIXED_Y, NULL, NULL, NULL, VALUE, VALUE,
       * GKS$K_COORDINATES_NDC, XFORM_MATRIX )

  C     Transform the segment and update the screen.
        CALL GKS$SET_SEG_XFORM( PICKED_SEGMENT, XFORM_MATRIX )

  C     Erase the message and redraw the segments...
        CALL GKS$REDRAW_SEG_ON_WS( WS_ID )
            .
            .
            .
        RETURN
        END
            .
            .
            .
```

The following numbers correspond to the numbers in the previous example:

❶ This code allows the arguments DATA_RECORD(1) and LOWER_LIMIT to share the same memory location, and DATA_RECORD(2) and UPPER_LIMIT to share the same location. In this way, UPPER_LIMIT and LOWER_LIMIT are properly documented as real number range specifications.

❷ This code checks to make sure that the workstation supports at least one valuator logical input device.

❸ This code specifies a data record buffer of 8 bytes. The prompt and echo type has no bearing on the data record size when you use a valuator class device. The GKS standard states that all graphics handlers must implement and use a two component data record containing the lower and upper bound values of the real number range.

❹ This code creates an identity matrix. Setting this matrix to be the current transformation matrix causes no change to the segment stored on the NDC plane.

❺ This code specifies that the upper limit is the real value 1.5, that the lower limit is 0.5, and that the indicator points to the current value of 1.0 (the exact middle of the real number range). The value that the user returns is used for scaling the picked segment. A value of 1.5 increases the segment size by 50 percent; a value of 1.0 maintains the current size; and, the value 0.5 decreases the segment size by 50 percent.

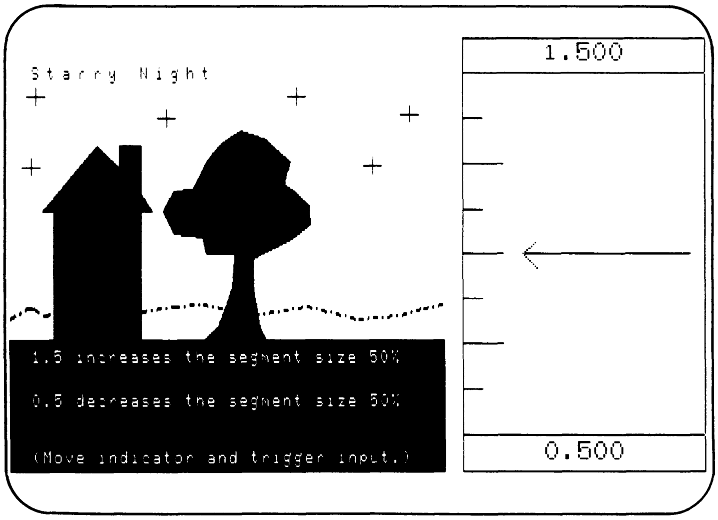
❻ This code initializes the valuator logical input device using the new range specification, the current value specification, and the default prompt and echo type.

❼ This code requests input. After the call to GKS$REQUEST_VALUATOR, the argument VALUE contains the user's value specification. Figure 6–7 illustrates the valuator prompt on the VT241 before the user moves the indicator.

❽ This code defines a valid fixed point on the NDC plane for each of the segments. Each of the fixed points are located in the center of the given segment. You need to define a fixed point in order to scale a segment.

❾ This code accumulates the scaling value specified by the user with the identity matrix, and then transforms the segment accordingly. In this example, the user picked the house to be scaled.

Figure 6–7: The VT241 Valuator Prompt—VT241



1.5 increases the segment size 50%

0.5 decreases the segment size 50%

(Move indicator and trigger input.)

6.6 Requesting Choice Input

In the previous code examples, the user is only allowed to specify a single scaling value. To improve the program, you can ask if the user is satisfied with the scaling. If the user is not satisfied with the scaling, then you can offer another chance to increase or decrease the scaling.

By adding code to the GO_FOR_INPUT and SPECIFY_VALUE subroutines, you can use a choice logical input device to offer the user the option of continued segment scaling. The following code performs this task. The numbers in this code example are not sequential because they follow the order in which the lines of code are executed.

```
C     ************************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )
          .
          .
          .
C     Initialize and request pick input...
      CALL PICK_IT( WS_ID, WS_TYPE, PICKED_SEGMENT )
```

```
C     Scale the segment...
      CALL SPECIFY_VALUE( WS_ID, WS_TYPE, PICKED_SEGMENT,
     * TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON )

C     Show the final picture...
⓯     FILL_Y( 3 ) = 0.1
      FILL_Y( 4 ) = 0.1
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
      CALL GKS$TEXT( 0.05, 0.05,
     * 'Here is the scaled segment.' )

C     Press RETURN when finished viewing the picture.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)
          .
          .
          .
      RETURN
      END

C     ************************************************************
C     Specify a value for scaling...
      SUBROUTINE SPECIFY_VALUE( WS_ID, WS_TYPE, PICKED_SEGMENT,
     * TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_TYPE, PICKED_SEGMENT,
          .
          .
          .
     * DUMMY_INTEGER, FINISHED_FLAG
          .
          .
          .
      CALL GKS$EVAL_XFORM_MATRIX( NULL, NULL, NULL, NULL,
     * NULL, NO_CHANGE, NO_CHANGE, GKS$K_COORDINATES_NDC,
     * XFORM_MATRIX )

❶ 300 CONTINUE

      VALUE = 1.0
      UPPER_LIMIT = 1.5
      LOWER_LIMIT = 0.5
          .
          .
          .
      CALL GKS$ACCUM_XFORM_MATRIX( XFORM_MATRIX, FIXED_X,
     * FIXED_Y, NULL, NULL, NULL, VALUE, VALUE,
     * GKS$K_COORDINATES_NDC, XFORM_MATRIX )

C     Transform the segment and update the screen.
      CALL GKS$SET_SEG_XFORM( PICKED_SEGMENT, XFORM_MATRIX )

C     Erase the message and redraw the segments...
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )
```

```
❷         CALL SATISFIED_CHOICE( WS_ID, WS_TYPE, FINISHED_FLAG )
      C   If the user isn't satisfied with the scaling...
          IF ( FINISHED_FLAG .EQ. 2 ) THEN
               GO TO 300
          ENDIF

      C   When the user is satisfied, redraw the segments.
          CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

          RETURN
          END

      C   ******************************************************
      C   This function makes sure that the user is satisfied with input...
          SUBROUTINE SATISFIED_CHOICE( WS_ID, WS_TYPE, FINISHED_FLAG )

          IMPLICIT NONE
          INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
          INTEGER WS_ID, WS_TYPE, FINISHED_FLAG, DATA_RECORD( 3 ),
         * NUM_CHOICES, SIZES( 10 ), ADDRESSES( 10 ), PROMPT_ECHO_TYPE,
         * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, RECORD_BUFFER_LENGTH,
         * RECORD_SIZE, INPUT_STATUS, INITIAL_CHOICE, DEVICE_NUM,
         * INPUT_CHOICE, INITIAL_STATUS, FILL_PTS, NUM_CHOICE_DEVICES,
         * LIST_PROMPT_TYPES( 6 ), PROMPT_RETURN_SIZE, PROMPT_FLAG,
         * INCR, DUMMY_INTEGER
          REAL ECHO_AREA( 4 ), FILL_X( 5 ), FILL_Y( 5 )

          CHARACTER*80 DEFAULT_STRINGS( 2 )

          DATA DEVICE_NUM / 1 /, FILL_PTS / 5 /, PROMPT_FLAG / 0 /

          DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
          DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

      C   First element in the data record is the number of choices.
❸         EQUIVALENCE( DATA_RECORD(1), NUM_CHOICES )

      C   Make sure that the device supports choice input...
❹         CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
         * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
         * NUM_CHOICE_DEVICES, DUMMY_INTEGER, DUMMY_INTEGER )

          IF ( NUM_CHOICE_DEVICES .EQ. 0 ) THEN
               WRITE(6,*) 'The workstation does not support'
               WRITE(6,*) 'choice input.'
               STOP
          ENDIF

      C   Establish the size of the record buffer: 12 bytes.
❺         RECORD_BUFFER_LENGTH = 12

      C   The second element in the choice data record for prompt and echo type 1
      C   is the pointer to the array containing sizes of each choice character
      C   string. You need to initialize the pointer so that the array can be
      C   initialized.
❻         DATA_RECORD( 2 ) = %LOC( SIZES( 1 ) )

      C   The third element in the VT241 choice data record is the pointer to the
      C   array containing the pointers to the strings to be used. You need
      C   to initialize the pointer so that the array can be initialized.
❼         DATA_RECORD( 3 ) = %LOC( ADDRESSES( 1 ) )
❽         ADDRESSES( 1 ) = %LOC( DEFAULT_STRINGS( 1 ) )
          ADDRESSES( 2 ) = %LOC( DEFAULT_STRINGS( 2 ) )
```

```
  C     Inquire about the default values.
❾       NUM_CHOICES = 2

  C     Obtain the available prompt and echo types...
        CALL GKS$INQ_DEF_CHOICE_DATA( WS_TYPE, DEVICE_NUM,
       * ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
       * %DESCR( LIST_PROMPT_TYPES), ECHO_AREA, DATA_RECORD,
       * PROMPT_RETURN_SIZE, RECORD_BUFFER_LENGTH,
       * RECORD_SIZE )

  C     Obtain the remaining default input values...
        CALL GKS$INQ_CHOICE_STATE( WS_ID, DEVICE_NUM,
       * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, INITIAL_STATUS,
       * INITIAL_CHOICE, PROMPT_ECHO_TYPE, ECHO_AREA,
       * DATA_RECORD, RECORD_BUFFER_LENGTH, RECORD_SIZE )

❿       DO 400 INCR = 1, PROMPT_RETURN_SIZE, 1
        IF ( LIST_PROMPT_TYPES( INCR ) .EQ. 3 ) THEN
             PROMPT_FLAG = 1
        ENDIF
  400   CONTINUE

  C     If the workstation does not support prompt and echo type 3...
        IF ( PROMPT_FLAG .EQ. 0 ) THEN
             WRITE(6,*) 'The workstation does not support'
             WRITE(6,*) 'choice prompt and echo type 3.'
             STOP
        ENDIF

        PROMPT_ECHO_TYPE = 3
        INITIAL_CHOICE = 1
        INITIAL_STATUS = GKS$K_STATUS_OK

  C     Establish sizes of prompt strings...
⓫       SIZES( 1 ) = 3
        SIZES( 2 ) = 2
  C     Establish locations of prompt strings...
        ADDRESSES( 1 ) = %LOC( 'Yes' )
        ADDRESSES( 2 ) = %LOC( 'No' )

  C     Write the message...
⓬       CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
        CALL GKS$TEXT( 0.05, 0.25,
       * 'Choose YES if you are satisfied with' )
        CALL GKS$TEXT( 0.05, 0.15,
       * 'your input, otherwise choose NO.' )
        CALL GKS$TEXT( 0.05, 0.02,
       * '(Move indicator and trigger input.)' )

  C     Since the device is in request mode by default, initialize the device...
⓭       CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
       * INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
       * ECHO_AREA, DATA_RECORD, RECORD_BUFFER_LENGTH )

        CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
       * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

⓮       CALL GKS$REQUEST_CHOICE( WS_ID, DEVICE_NUM, INPUT_STATUS,
       * FINISHED_FLAG )

        RETURN
        END
```

The following numbers correspond to the numbers in the previous example:

❶ This FORTRAN label provides a means for looping.

❷ This code calls the subroutine SATISFIED_CHOICE. After the subroutine call, the argument FINISHED_FLAG contains either the value 1 (corresponding to Yes) or the value 2 (corresponding to No). If the user chooses No, then control transfers to the label 300, and the program offers the user another chance to scale the segment. The effects of the scaling are cumulative, with each additional specified scale value being combined with the last transformation matrix to form a new matrix.

❸ This code makes it easier to understand that the first data record component is the number of choices.

❹ This code checks to make sure that the given workstation supports at least one choice logical input device.

❺ This code specifies the size of the data record buffer.

❻ This code initializes the second component of the choice data record. This component is the address of the array that contains the size of the string buffers.

❼ This code initializes the third component of the choice data record. This component is the address of the array that contains the addresses of the string buffers.

❽ This code initializes the array of string addresses to point to each of the string buffers. Once you have initialized all of the string buffers, you can call the choice inquiry functions.

The choice inquiry functions require string buffers in which to write the default or current choice strings. If you do not initialize all of the buffer addresses, calls to the choice inquiry functions generate errors.

❾ Once you tell the graphics handler how many choice string buffers you have, you can call the inquiry functions. This subroutine calls GKS$INQ_DEF_CHOICE_DATA to obtain the supported prompt and echo types for the workstation, and it calls GKS$INQ_CHOICE_STATE to obtain the rest of the choice input data values.

❿ This subroutine chooses to use choice prompt and echo type number 3. This code checks to make sure that the device supports that prompt and echo type.

All of the DEC GKS devices currently implement choice prompt and echo type number 3 using three data record components. For more information concerning the GKS standard choice data records, refer to Chapter 6, Input Functions, in the *DEC GKS Reference Manual*. For more information concerning the DEC GKS supported graphics handlers and their choice logical input devices, refer to the appropriate chapter in the *DEC GKS Device Specific Reference Manual*.

⓫ This code initializes the choice strings so that they apply to the requirements of the program. The choices are Yes and No.

⓬ This code sends a message to the user. The message asks if the user is satisfied with the scaled segment.

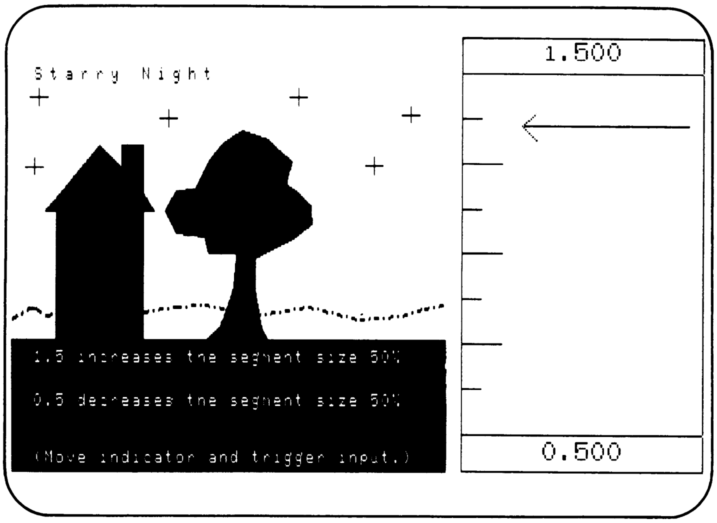
⓭ This code initializes the choice logical input device with the appropriate prompt and echo type, and with the new choice labels.

⓮ This code requests input. The argument FINISHED_FLAG contains either the value 1 (Yes) or 2 (No).

⓯ This code generates the picture including the scaled segment as specified by the user.

The following figures represent samples of user input using the looping mechanism and the SATISFIED_CHOICE subroutine. Figure 6–8 illustrates a large scaling specification. Figure 6–9 illustrates a No response using the choice logical input device. Figure 6–10 illustrates a second, smaller scaling specification. Figure 6–11 illustrates a Yes response using the choice logical input device. Figure 6–12 illustrates the picture of the scaled segment as specified by the user.

Figure 6–8: Specifying a Large Scaling Value—VT241



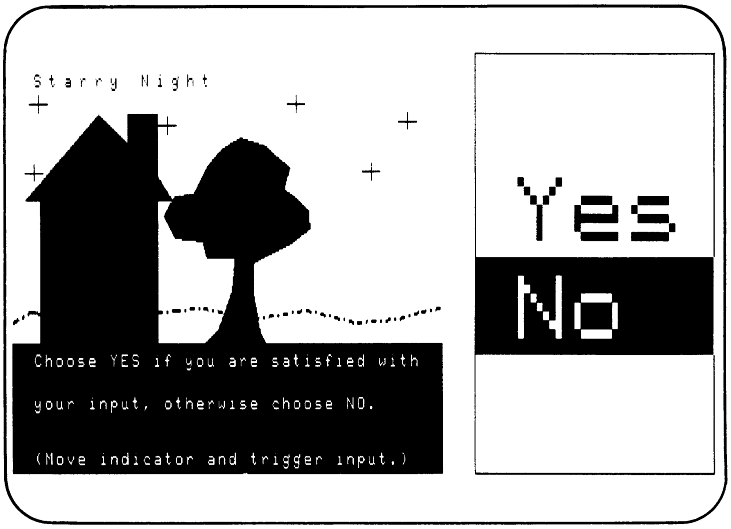
1.5 increases the segment size 50%

0.5 decreases the segment size 50%

(Move indicator and trigger input.)

ZK-5318-86

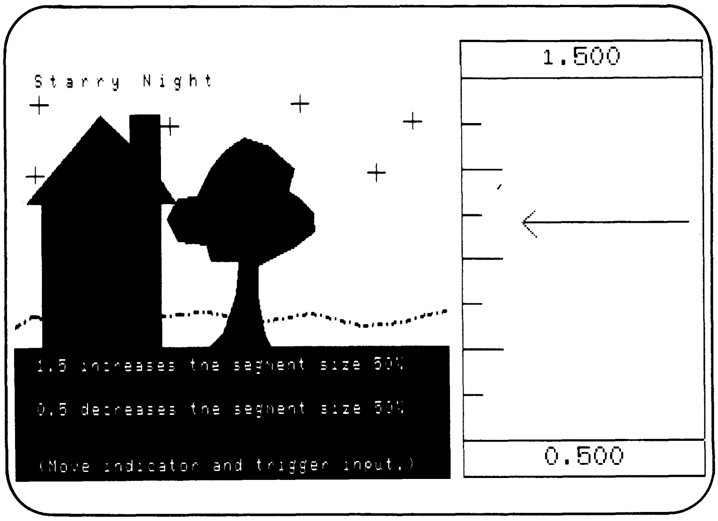
Figure 6–9: Choosing "No"—VT241



```
Choose YES if you are satisfied with
your input, otherwise choose NO.
(Move indicator and trigger input.)
```

ZK-5308-86

Figure 6–10: Specifying an Additional, Smaller Scaling Value—VT241



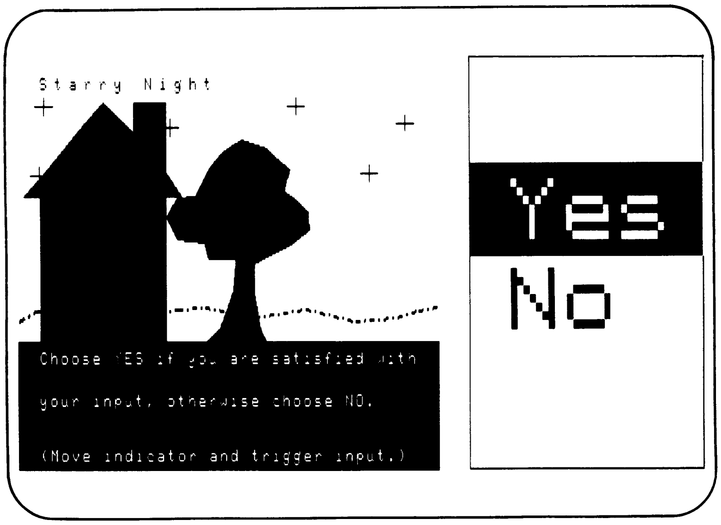
1.5 increases the segment size 50%

0.5 decreases the segment size 50%

(Move indicator and trigger input.)

ZK-5317-86

Figure 6–11: Choosing "Yes"—VT241

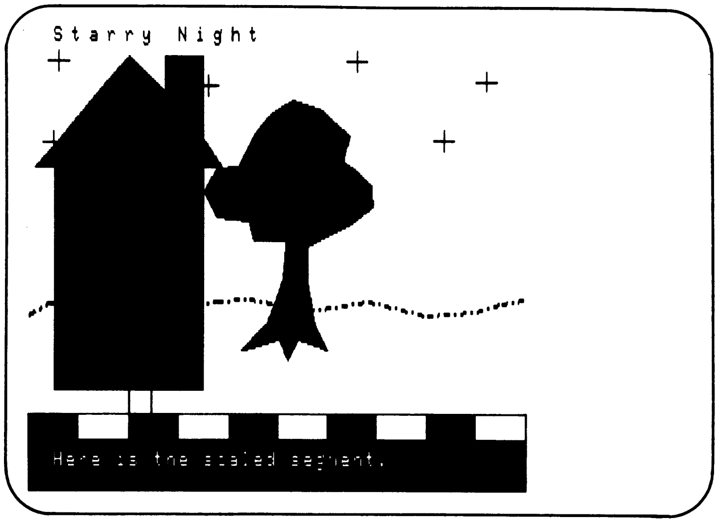


Choose YES if you are satisfied with your input, otherwise choose NO.

(Move indicator and trigger input.)

ZK-5309-86

Figure 6–12: Presenting the Scaled Segment as Specified—VT241



6.7 Input Viewport Priority

During locator and stroke input, the user positions the prompt on the workstation surface and returns one point or a series of points in device coordinates. DEC GKS translates the device coordinates to NDC points, and then uses the viewport input priority to determine which normalization transformation to use when translating the points to world coordinates.

To decide which normalization viewport has a higher input priority, DEC GKS maintains a priority list. By default, DEC GKS assigns the highest priority to the unity transformation (0). The viewports of all remaining transformations decrease in priority as their transformation numbers increase (viewport 0 higher than viewport 1, 1 higher than 2, 2 higher than 3, and so forth).

When using a locator class device, DEC GKS uses the normalization transformation of the highest input priority that contains the input point. When using stroke input, DEC GKS uses the normalization transformation of the highest priority that contains *all* of the points in the stroke. Since a locator or stroke input device could not return device coordinate points that could fall outside of the default normalization viewport ([0,1] x [0,1]), you can always use the unity transformation to transform stroke input data.

The subroutine VIEW_PRIORITY illustrates the use of input viewport priority when using a locator logical input device:

```
          .
          .
          .
C     ************************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )
          .
          .
          .
C     Make the segments invisible so that you can run the viewport
C     priority subroutine...
      CALL GKS$SET_SEG_VISIBILITY( TITLE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( STARS, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( TREE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( SIDE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( ROAD, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HOUSE, GKS$K_INVISIBLE )
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

      CALL VIEW_PRIORITY( WS_ID, WS_TYPE )

      RETURN
      END
          .
          .
          .
C     ************************************************************
C     This program accepts input twice from the same spot on the
C     workstation surface.  When the input priority is changed,
C     the world coordinates returned are that of the other overlapping
C     viewport.
      SUBROUTINE VIEW_PRIORITY( WS_ID, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, ERROR_STATUS, INPUT_MODE, ECHO_FLAG, XFORM,
     * RECORD_BUFFER_LENGTH, RECORD_SIZE, INPUT_STATUS, DEFAULT,
     * LOW_LEFT_CORNER, RIGHT_HALF, DEVICE_NUM, NUM_POINTS,
     * PROMPT_ECHO_TYPE, DATA_RECORD( 1 ), WS_TYPE, DUMMY_INTEGER,
     * NUM_LOC_DEVICES, FOREGROUND
      REAL WORLD_COORD_X, WORLD_COORD_Y,
     * ECHO_AREA( 4 ), PX( 5 ), PY( 5 ), PX_2( 5 ), PY_2( 5 ),
     * LARGER
      DATA PX / 0.0, 1.0, 1.0, 0.0, 0.0 /
      DATA PY / 0.0, 0.0, 1.0, 1.0, 0.0 /
      DATA DEFAULT / 0 /, DEVICE_NUM / 1 /, LARGER / 0.04 /,
     * RIGHT_HALF / 1 /, LOW_LEFT_CORNER / 1 /, NUM_POINTS / 5 /,
     * FORGROUND / 1 /

C     Make sure that the device supports locator input...
❶     CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * NUM_LOC_DEVICES, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER )
```

```
          IF ( NUM_LOC_DEVICES .EQ. 0 ) THEN
               WRITE(6,*) 'The workstation does not support'
               WRITE(6,*) 'locator input.'
               STOP
          ENDIF

    C     When you outline the entire default world coordinate space, you
    C     also outline the entire NDC space, the entire workstation window,
    C     and the entire workstation viewport.
          CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_SOLID )
          CALL GKS$POLYLINE( NUM_POINTS, PX, PY )

    C     This window and viewport are associated with the
    C     normalization transformation number 1.
          CALL GKS$SET_WINDOW( LOW_LEFT_CORNER, 0.0, 0.5, 0.0, 0.5 )
          CALL GKS$SET_VIEWPORT( RIGHT_HALF, 0.5, 1.0, 0.0, 1.0 )

    C     Select the new transformation and outline the new windows
    C     and viewports.
          CALL GKS$SELECT_XFORM( 1 )
          CALL GKS$POLYLINE( NUM_POINTS, PX, PY )

    C     Assign a value to RECORD_BUFFER_LENGTH: 4 bytes.  On output,
    C     this argument should contain the value 0 since
    C     GKS$INQ_LOCATOR_STATE does not write anything to the buffer.
❷         RECORD_BUFFER_LENGTH = 4
          CALL GKS$INQ_LOCATOR_STATE( WS_ID, DEVICE_NUM,
         * GKS$K_VALUE_REALIZED, ERROR_STATUS, INPUT_MODE, ECHO_FLAG,
         * XFORM, WORLD_COORD_X, WORLD_COORD_Y, PROMPT_ECHO_TYPE,
         * ECHO_AREA, DATA_RECORD, RECORD_BUFFER_LENGTH,
         * RECORD_SIZE )

    C     Since the device is in request mode by default, initialize the device...
❸         CALL GKS$INIT_LOCATOR( WS_ID, DEVICE_NUM, 0.7, 0.5, DEFAULT,
         * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
         * RECORD_BUFFER_LENGTH )

          CALL GKS$SET_LOCATOR_MODE( WS_ID, DEVICE_NUM,
         * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

    C     ********************************
    C     At this pause, just type RETURN.
    C     ********************************
❹         CALL GKS$REQUEST_LOCATOR( WS_ID, DEVICE_NUM, INPUT_STATUS,
         * XFORM, WORLD_COORD_X, WORLD_COORD_Y )
    C     Write the returned world coordinates.
❺         WRITE(5, *) WORLD_COORD_X, WORLD_COORD_Y
          CALL GKS$SELECT_XFORM( DEFAULT )
          CALL GKS$SET_TEXT_COLOR_INDEX( FOREGROUND )
          CALL GKS$SET_TEXT_HEIGHT( LARGER )
          CALL GKS$TEXT( 0.01, 0.4, 'Higher priority VP: 0')

    C     Set the current viewport (associated with the selected
    C     transformation number 1) to be a higher priority than the
    C     default viewport.
❻         CALL GKS$SET_VIEWPORT_PRIORITY( RIGHT_HALF, DEFAULT,
         * GKS$K_INPUT_PRIORITY_HIGHER )
```

```
C     Since the device is in request mode by default, initialize the device...
      CALL GKS$INIT_LOCATOR( WS_ID, DEVICE_NUM, 0.7, 0.5, DEFAULT,
     * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
     * RECORD_BUFFER_LENGTH )
      CALL GKS$SET_LOCATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

C     **********************************
C     At this pause, just type RETURN.
C     **********************************
      CALL GKS$REQUEST_LOCATOR( WS_ID, DEVICE_NUM, INPUT_STATUS,
     * XFORM, WORLD_COORD_X, WORLD_COORD_Y )

C     Write the returned world coordinates, this time from the smaller
C     viewport on the right half of the screen.
❼     WRITE(5, *) WORLD_COORD_X, WORLD_COORD_Y
      CALL GKS$SELECT_XFORM( DEFAULT )
      CALL GKS$TEXT( 0.01, 0.3, 'Higher priority VP: 1')

      RETURN
      END
        .
        .
        .
```

The following numbers correspond to the numbers in the previous example:

❶ This code makes sure that the device supports locator input.

❷ This code inquires about the default locator input values.

❸ This code initializes the locator logical input device with default input values, and with the initial locator world coordinate values 0.7 and 0.5. Using the normalization transformation specified by the argument DEFAULT (0, the unity transformation), the graphics handler transforms the initial point and places the locator prompt on the corresponding position on the workstation surface (device coordinate plane).

❹ Figure 6–13 illustrates the workstation surface after the request for locator input.

❺ After the user triggers the locator input device, the graphics handler returns the same world coordinate points specified as the original prompt position (0.7, 0.5). Since the unity normalization transformation has the highest viewport input priority, the handler uses the unity transformation to transform the prompt position back to world coordinates. Figure 6–14 illustrates the surface of the workstation after the user triggers input.

❻ This code sets normalization transformation number 1 (whose viewport occupies the right half of the default NDC space) to be higher than the unity transformation (0).

❼ The second time that the user triggers input from the same prompt position, the graphics handler uses the window and viewport associated with normalization transformation number 1, since it now has a higher priority. As a result, the handler transforms the same prompt position to the world coordinate point (0.2, 0.25). Figure 6–15 illustrates the workstation surface after the user triggers input a second time.

Figure 6–13: The Initial Locator Prompt Position—VT241

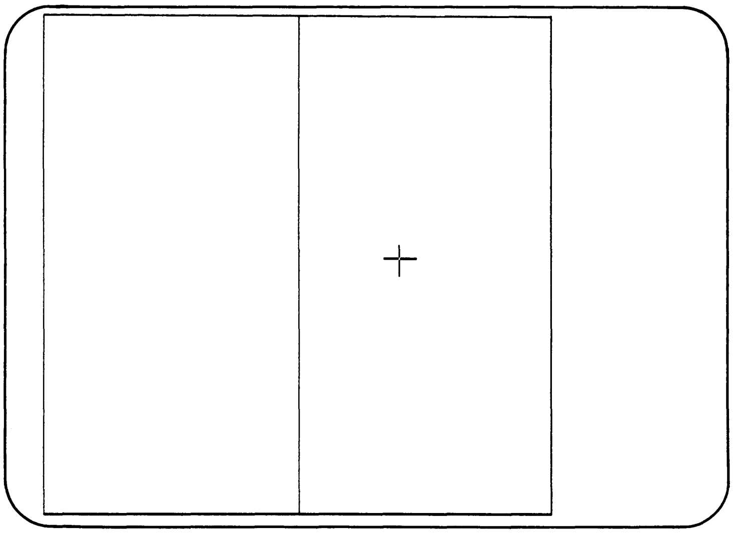
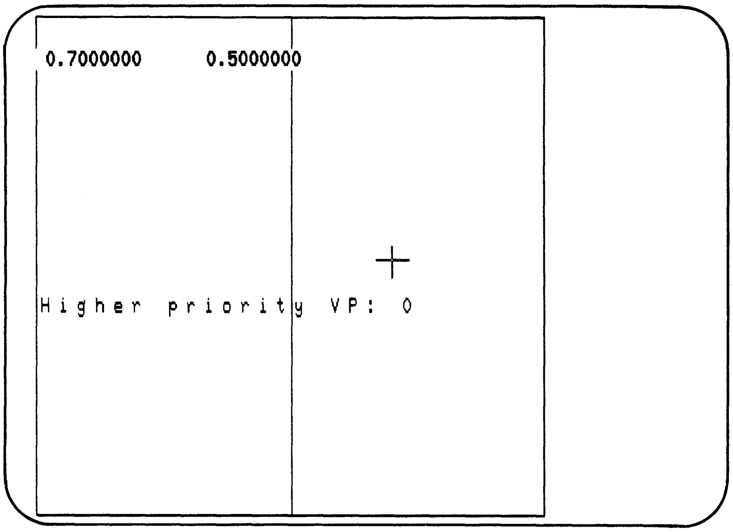


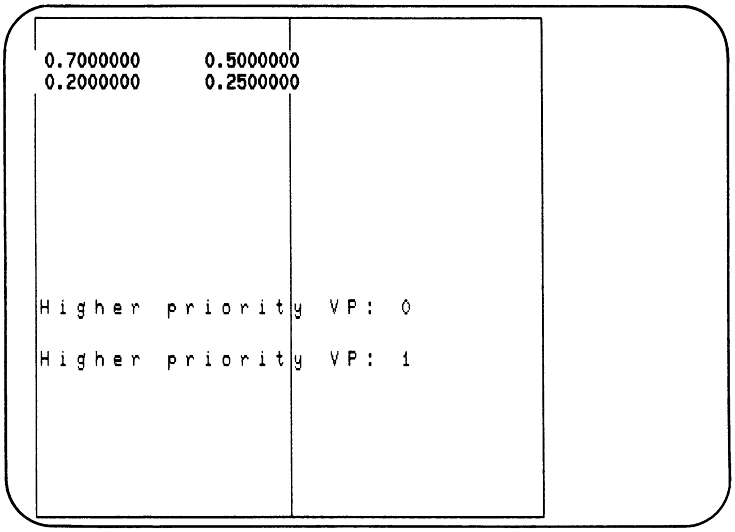
Figure 6–14: Unity Transformation Measure—VT241



```
0.7000000      0.5000000

Higher priority VP: 0
```

ZK-5313-86

Figure 6–15: Transformation Number 1 Measure—VT241

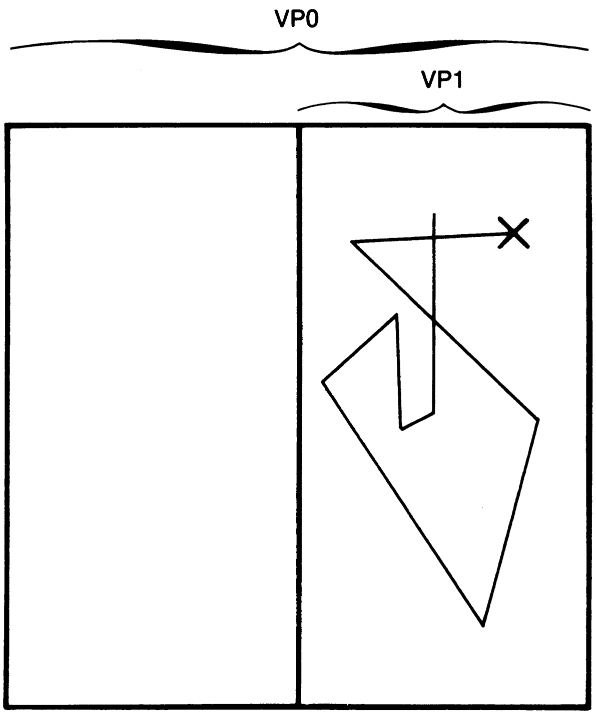


```
0.7000000       0.5000000
0.2000000       0.2500000





Higher priority VP: 0

Higher priority VP: 1
```

ZK-5314-86

Given the set normalization transformations and input viewport priorities in the previous code example (normalization transformation number 1 being higher priority than 0), the following figures illustrate which normalization transformation the graphics handler uses to transform a given stroke. Figure 6–16 illustrates a stroke located within the transformed range of normalization transformation number 1. Figure 6–17 illustrates a stroke that exceeds the transformed range of normalization number 1.

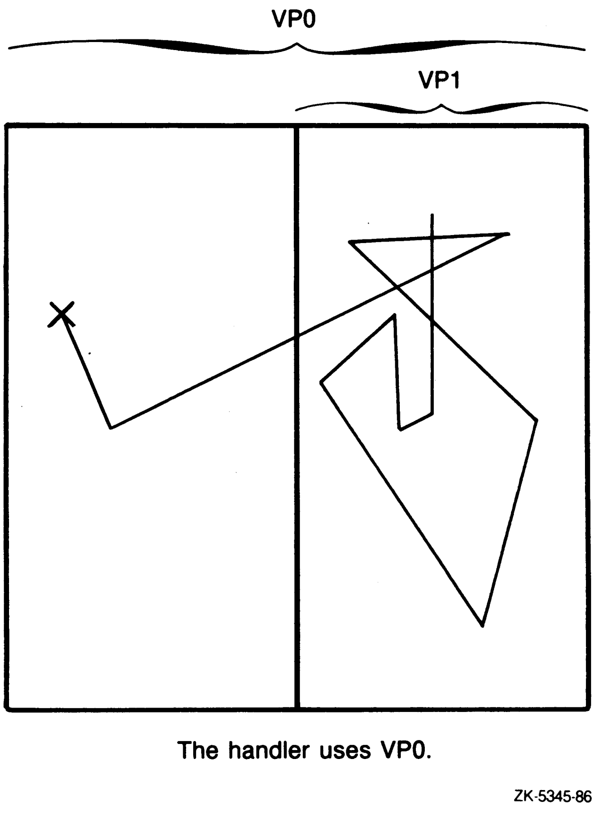
Figure 6–16: Using Normalization Transformation Number 1



The handler uses VP1.

ZK-5344-86

Figure 6–17: Using Normalization Transformation Number 0



Since in Figure 6–17 normalization transformation number 1 (which has the viewport of higher priority) cannot contain the entire stroke, the handler uses the normalization transformation with the next highest priority that contains all stroke points (in this case, transformation 0).

6.7.1 Restricting Movement of Locator, Stroke, and Pick Prompts

In some applications, you may wish to limit the range of the locator, stroke, or pick prompt so that you have complete control over the input values returned by the input device. Using stroke and locator input with a picture composed of overlapping viewports, you can restrict the input prompt to a single portion of the workstation surface. By restricting the movement of the prompt, you limit the number of normalization transformation numbers DEC GKS can use to translate a given input point or series of input points. Using a pick device, you can use the echo area—along with the visibility and detectability attributes of each segment—to control which segments the user can and cannot pick.

Assuming the normalization transformations established in the previous code example, you have two possible normalization transformations used to transform an input point (0 and 1). If you want the user to enter a point or a series of points to be transformed using normalization transformation number 1, you need to make sure that the input device cannot return a point that falls outside of normalization viewport 1.

To accomplish this, you adjust the input echo area. Adjusting the echo area restricts the amount of the workstation surface that the user can use to move the prompt.

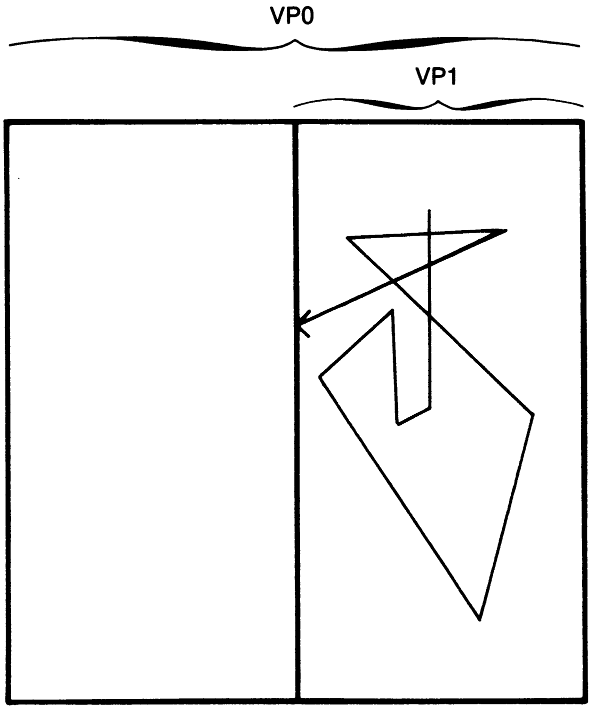
For example, the following code illustrates a method used to limit the echo area for a stroke device:

```
          .
          .
          .
C     Assume that there are no pending workstation transformations.
C     Obtain the dimensions of the current workstation viewport...
      CALL GKS$INQ_WS_XFORM( WS_ID, ERROR_STATUS,
     * XFORM_PENDING_FLAG, REQUESTED_WS_WINDOW, CURRENT_WS_WINDOW,
     * REQUESTED_WS_VIEWPORT, CURRENT_WS_VIEWPORT )

C     Use the right half of the current viewport as the echo area.
      CURRENT_WS_VIEWPORT( 1 ) = CURRENT_WS_VIEWPORT( 2 ) / 2
      ECHO_AREA( 1 ) = CURRENT_WS_VIEWPORT ( 1 )
      ECHO_AREA( 2 ) = CURRENT_WS_VIEWPORT ( 2 )
      ECHO_AREA( 3 ) = CURRENT_WS_VIEWPORT ( 3 )
      ECHO_AREA( 4 ) = CURRENT_WS_VIEWPORT ( 4 )

C     Initialize the new echo area...
      CALL GKS$INIT_STROKE( WS_ID, DEVICE_NUM, NUMBER_PTS,
     * X_PTS, Y_PTS, XFORM, PROMPT_ECHO_TYPE, ECHO_AREA,
     * DATA_RECORD, RECORD_BUFFER_LENGTH )
```

```
C     Give normalization viewport 1 a higher priority than 0...
      CALL GKS$SET_VIEWPORT_PRIORITY( 1, 0,
     * GKS$K_INPUT_PRIORITY_HIGHER )
          .
          .
          .
```

Figure 6–18 illustrates the surface of the workstation when the user attempts to move the prompt past the currently defined echo area.

Figure 6–18: Restricting the Echo Area



The handler still uses VP1.

ZK-5854-HC

6.8 Program Example Used in this Chapter

Example 6–1 presents all of the changes that you need to make to Example 3–2 in order to follow the code examples in this chapter.

Example 6–1: Using the DEC GKS Input Functions

```
      IMPLICIT NONE
      INTEGER WS_ID, HOUSE, TREE, HORIZON, STARS, TITLE,
     * SIDE, ROAD

      DATA WS_ID / 1 /, TITLE / 1 /, STARS / 2 /, TREE / 3 /,
     * SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /, HOUSE / 7 /

      CALL SET_UP( WS_ID )

      CALL DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )

      CALL CLEAN_UP( WS_ID )

      END
         .
         .
         .
C     ***********************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
         .
         .
         .
      CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y )
      CALL GKS$CLOSE_SEG()

C     Ask the user for input...
      CALL GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS, TREE,
     * SIDE, ROAD, HOUSE, HORIZON )

      RETURN
      END

C     ***********************************************************
C     From this point forward, all code is additional code that you
C     need to add to the "Starry Night" program.
C     ***********************************************************

C     ***********************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )
```

(continued on next page)

Example 6–1 (Cont.): Using the DEC GKS Input Functions

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, CATEGORY, ERROR_STATUS, FILL_PTS, FOREGROUND,
     * BACKGROUND, UNITY, PICKED_SEGMENT, WS_TYPE
      REAL FILL_X( 5 ), FILL_Y( 5 )

      DATA FILL_PTS / 5 /, FOREGROUND / 1 /, BACKGROUND / 0 /,
     * UNITY / 0 /


      DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
      DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

      CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
     * ERROR_STATUS, CATEGORY )

      IF ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'both input and output.'
           STOP
      ENDIF

C     Make sure that you are using the unity transformation...
      CALL GKS$SELECT_XFORM( UNITY )

C     Fill an area on which to send the user a message...
      CALL GKS$SET_FILL_COLOR_INDEX( FOREGROUND )
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )

C     Write the message...
      CALL GKS$SET_TEXT_HEIGHT( 0.028 )
      CALL GKS$SET_TEXT_COLOR_INDEX( BACKGROUND )
      CALL GKS$TEXT( 0.05, 0.25,
     * 'Which segment would you like to scale?' )
      CALL GKS$TEXT( 0.05, 0.2,
     * 'Move the cursor, outline your chosen' )
      CALL GKS$TEXT( 0.05, 0.15,
     * 'segment, and then trigger input.' )
      CALL GKS$TEXT( 0.05, 0.02,
     * '(Press RETURN when ready.)' )

C     The user presses RETURN when ready to pick...
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

C     Erase the message and redraw the segments...
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )
```

(continued on next page)

```
C     Make sure that all of the segments are detectable...
      CALL GKS$SET_SEG_DETECTABILITY( TITLE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( STARS, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( TREE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( SIDE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( ROAD, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HOUSE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HORIZON, GKS$K_DETECTABLE )

C     Initialize and request pick input...
      CALL PICK_IT( WS_ID, WS_TYPE, PICKED_SEGMENT )

C     Scale the segment...
      CALL SPECIFY_VALUE( WS_ID, WS_TYPE, PICKED_SEGMENT,
     * TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON )

C     Show the final picture...
      FILL_Y( 3 ) = 0.1
      FILL_Y( 4 ) = 0.1
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
      CALL GKS$TEXT( 0.05, 0.05,
     * 'Here is the scaled segment.' )

C     Press RETURN when finished viewing the picture.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

C     Make the segments invisible so that you can run the viewport
C     priority  subroutine...
      CALL GKS$SET_SEG_VISIBILITY( TITLE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( STARS, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( TREE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( SIDE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( ROAD, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HOUSE, GKS$K_INVISIBLE )
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

      CALL VIEW_PRIORITY( WS_ID, WS_TYPE )

      RETURN
      END

C     ************************************************************
C     This function allows a user to pick a segment...
      SUBROUTINE PICK_IT( WS_ID, WS_TYPE, PICKED_SEGMENT )
```

(continued on next page)

Example 6–1 (Cont.): Using the DEC GKS Input Functions

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, INITIAL_STATUS, PICKED_SEGMENT,
     * PICK_ID, PROMPT_ECHO_TYPE, ERROR_STATUS,
     * INPUT_MODE, ECHO_FLAG, RECORD_BUFFER_LENGTH,
     * RECORD_SIZE, INPUT_STATUS, DEVICE_NUM,
     * INPUT_CHOICE, DUMMY_INTEGER, DATA_RECORD( 10 ),
     * NUM_PICK_DEVICES, FILL_PTS, STATUS, WS_TYPE
      REAL ECHO_AREA( 4 ), FILL_X( 5 ), FILL_Y( 5 )
      DATA DEVICE_NUM / 1 /, FILL_PTS / 5 /
      DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
      DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

C     Make sure that the device supports pick input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, NUM_PICK_DEVICES, DUMMY_INTEGER )

      IF ( NUM_PICK_DEVICES .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'pick input.'
           STOP
      ENDIF

C     Give the data record the size of your data record buffer and
C     inquire about the realized pick values.
      RECORD_BUFFER_LENGTH = 40
      CALL GKS$INQ_PICK_STATE( WS_ID, DEVICE_NUM,
     * GKS$K_VALUE_REALIZED, ERROR_STATUS, INPUT_MODE,
     * ECHO_FLAG, INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
     * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
     * RECORD_BUFFER_LENGTH, RECORD_SIZE )

C     Make sure that the data record was not truncated...
      IF ( RECORD_SIZE .LT. RECORD_BUFFER_LENGTH ) THEN
           WRITE(6,*) 'The data record was truncated.'
           WRITE(6,*) 'Declare a larger buffer.'
           STOP
      ENDIF


C     Allow entrance into the loop...
      INPUT_STATUS = GKS$K_STATUS_NOPICK
C     Loop until the user picks a segment...
      DO WHILE ( ( INPUT_STATUS .EQ. GKS$K_STATUS_NOPICK ) .OR.
     *           ( INPUT_STATUS .EQ. GKS$K_STATUS_NONE ))

C     Make sure that the pick aperture is not placed on any segment...
      INITIAL_STATUS = GKS$K_STATUS_NOPICK

C     Since the device is in request mode by default, initialize the device...
      CALL GKS$INIT_PICK( WS_ID, DEVICE_NUM, INITIAL_STATUS,
     * PICKED_SEGMENT, PICK_ID, PROMPT_ECHO_TYPE, ECHO_AREA,
     * DATA_RECORD, RECORD_BUFFER_LENGTH )
```

(continued on next page)

Example 6–1 (Cont.): Using the DEC GKS Input Functions

```
C     Make sure that echo is enabled...
      CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

C     Request input. PICKED_SEGMENT contains the chosen segment's
C     name.
      CALL GKS$REQUEST_PICK( WS_ID, DEVICE_NUM,
     * INPUT_STATUS, PICKED_SEGMENT, PICK_ID )

C     Send a message to the user if a segment is not picked...
      IF ( INPUT_STATUS .EQ. GKS$K_STATUS_NOPICK ) THEN
           CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
           CALL GKS$TEXT( 0.05, 0.25,
     *          'I cannot let you go until you pick' )
           CALL GKS$TEXT( 0.05, 0.2,
     *          'a segment!' )
           CALL GKS$TEXT( 0.05, 0.02,
     *          '(Press RETURN when ready.)' )

C     The user presses RETURN when ready to pick again...
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
           READ(5,*)

C     Erase the message and redraw the segments...
           CALL GKS$REDRAW_SEG_ON_WS( WS_ID )
      ENDIF
      ENDDO

      RETURN
      END


C     ***********************************************************
C     Specify a value for scaling...
      SUBROUTINE SPECIFY_VALUE( WS_ID, WS_TYPE, PICKED_SEGMENT,
     * TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_TYPE, PICKED_SEGMENT,
     * TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON,
     * PROMPT_ECHO_TYPE, ERROR_STATUS, INPUT_MODE,
     * ECHO_FLAG, RECORD_BUFFER_LENGTH, RECORD_SIZE,
     * INPUT_STATUS, DEVICE_NUM, FILL_PTS, NUM_VAL_DEVICES,
     * DUMMY_INTEGER, FINISHED_FLAG
      REAL ECHO_AREA( 4 ), DATA_RECORD( 2 ), UPPER_LIMIT,
     * LOWER_LIMIT, VALUE, FILL_X( 5 ), FILL_Y( 5 ),
     * FIXED_X, FIXED_Y, XFORM_MATRIX( 6 ), NULL, NO_CHANGE
      DATA DEVICE_NUM / 1 /, FILL_PTS / 5 /, NULL / 0.0 /,
     * NO_CHANGE / 1.0 /

      DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
      DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /
```

(continued on next page)

Example 6–1 (Cont.): Using the DEC GKS Input Functions

```
C     According to the standard, the elements in the data record are
C     the upper and lower limits for all prompt and echo types.
      EQUIVALENCE( DATA_RECORD( 1 ), LOWER_LIMIT )
      EQUIVALENCE( DATA_RECORD( 2 ), UPPER_LIMIT )

C     Make sure that the device supports valuator input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, NUM_VAL_DEVICES,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER )

      IF ( NUM_VAL_DEVICES .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'valuator input.'
           STOP
      ENDIF

      RECORD_BUFFER_LENGTH = 8
      CALL GKS$INQ_VALUATOR_STATE( WS_ID, DEVICE_NUM,
     * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, VALUE,
     * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
     * RECORD_BUFFER_LENGTH, RETURN_SIZE )

      CALL GKS$EVAL_XFORM_MATRIX( NULL, NULL, NULL, NULL,
     * NULL, NO_CHANGE, NO_CHANGE, GKS$K_COORDINATES_NDC,
     * XFORM_MATRIX )


300   CONTINUE

      VALUE = 1.0
      UPPER_LIMIT = 1.5
      LOWER_LIMIT = 0.5

C     Write the message...
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
      CALL GKS$TEXT( 0.05, 0.25,
     * '1.5 increases the segment size 50%' )
      CALL GKS$TEXT( 0.05, 0.15,
     * '0.5 decreases the segment size 50%' )
      CALL GKS$TEXT( 0.05, 0.02,
     * '(Move indicator and trigger input.)' )

C     Since the device is in request mode by default, initialize the device...
      CALL GKS$INIT_VALUATOR( WS_ID, DEVICE_NUM,
     * VALUE, PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
     * RECORD_BUFFER_LENGTH )

      CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

      CALL GKS$REQUEST_VALUATOR( WS_ID, DEVICE_NUM,
     * INPUT_STATUS, VALUE )
```

(continued on next page)

```
C     Establish fixed points for segments depending on which
      IF ( PICKED_SEGMENT .EQ. TITLE ) THEN
         FIXED_X = 0.3
         FIXED_Y = 0.925
      ELSEIF ( PICKED_SEGMENT .EQ. STARS ) THEN
         FIXED_X = 0.5
         FIXED_Y = 0.8
      ELSEIF ( PICKED_SEGMENT .EQ. TREE ) THEN
         FIXED_X = 0.52
         FIXED_Y = 0.51
      ELSEIF ( PICKED_SEGMENT .EQ. SIDE ) THEN
         FIXED_X = 0.225
         FIXED_Y = 0.22
      ELSEIF ( PICKED_SEGMENT .EQ. ROAD ) THEN
         FIXED_X = 0.5
         FIXED_Y = 0.075
      ELSEIF ( PICKED_SEGMENT .EQ.  HORIZON ) THEN
         FIXED_X = 0.1
         FIXED_Y = 0.35
      ELSEIF ( PICKED_SEGMENT .EQ. HOUSE ) THEN
         FIXED_X = 0.2
         FIXED_Y = 0.5
      ENDIF

      CALL GKS$ACCUM_XFORM_MATRIX( XFORM_MATRIX, FIXED_X,
     * FIXED_Y, NULL, NULL, NULL, VALUE, VALUE,
     * GKS$K_COORDINATES_NDC, XFORM_MATRIX )

C     Transform the segment and update the screen.
      CALL GKS$SET_SEG_XFORM( PICKED_SEGMENT, XFORM_MATRIX )

C     Erase the message and redraw the segments...
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

      CALL SATISFIED_CHOICE( WS_ID, WS_TYPE, FINISHED_FLAG )
C     If the user isn't satisfied with the scaling...
      IF ( FINISHED_FLAG .EQ. 2 ) THEN
          GO TO 300
      ENDIF

C     When the user is satisfied, redraw the segments.
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

      RETURN
      END

C     ************************************************************
C     This function makes sure that the user is satisfied with input...
      SUBROUTINE SATISFIED_CHOICE( WS_ID, WS_TYPE, FINISHED_FLAG )
```

(continued on next page)

```
      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_TYPE, FINISHED_FLAG, DATA_RECORD( 3 ),
     * NUM_CHOICES, SIZES( 10 ), ADDRESSES( 10 ), PROMPT_ECHO_TYPE,
     * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, RECORD_BUFFER_LENGTH,
     * RECORD_SIZE, INPUT_STATUS, INITIAL_CHOICE, DEVICE_NUM,
     * INPUT_CHOICE, INITIAL_STATUS, FILL_PTS, NUM_CHOICE_DEVICES,
     * LIST_PROMPT_TYPES( 6 ), PROMPT_RETURN_SIZE, PROMPT_FLAG,
     * DUMMY_INTEGER, INCR
      REAL ECHO_AREA( 4 ), FILL_X( 5 ), FILL_Y( 5 )

      CHARACTER*80 DEFAULT_STRINGS( 2 )

      DATA DEVICE_NUM / 1 /, FILL_PTS / 5 /, PROMPT_FLAG / 0 /

      DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
      DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

C     First element in the data record is the number of choices.
      EQUIVALENCE( DATA_RECORD(1), NUM_CHOICES )


C     Make sure that the device supports choice input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NUM_CHOICE_DEVICES, DUMMY_INTEGER, DUMMY_INTEGER )

      IF ( NUM_CHOICE_DEVICES .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'choice input.'
           STOP
      ENDIF

C     Establish the size of the record buffer: 12 bytes.
      RECORD_BUFFER_LENGTH = 12

C     The second element in the choice data record for prompt and echo type 1
C     is the pointer to the array containing sizes of each choice character
C     string. You need to initialize the pointer so that the array can be
C     initialized.
      DATA_RECORD( 2 ) = %LOC( SIZES( 1 ) )

C     The third element in the VT241 choice data record is the pointer to the
C     array containing the pointers to the strings to be used. You need
C     to initialize the pointer so that the array can be initialized.
      DATA_RECORD( 3 ) = %LOC( ADDRESSES( 1 ) )
      ADDRESSES( 1 ) = %LOC( DEFAULT_STRINGS( 1 ) )
      ADDRESSES( 2 ) = %LOC( DEFAULT_STRINGS( 2 ) )

C     Inquire about the default values.
      NUM_CHOICES = 2
```

(continued on next page)

```
C       Obtain the available prompt and echo types...
        CALL GKS$INQ_DEF_CHOICE_DATA( WS_TYPE, DEVICE_NUM,
      * ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
      * %DESCR( LIST_PROMPT_TYPES), ECHO_AREA, DATA_RECORD,
      * PROMPT_RETURN_SIZE, RECORD_BUFFER_LENGTH,
      * RECORD_SIZE )

C       Obtain the remaining default input values...
        CALL GKS$INQ_CHOICE_STATE( WS_ID, DEVICE_NUM,
      * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, INITIAL_STATUS,
      * INITIAL_CHOICE, PROMPT_ECHO_TYPE, ECHO_AREA,
      * DATA_RECORD, RECORD_BUFFER_LENGTH, RECORD_SIZE )

        DO 400 INCR = 1, PROMPT_RETURN_SIZE, 1
        IF ( LIST_PROMPT_TYPES( INCR ) .EQ. 3 ) THEN
             PROMPT_FLAG = 1
        ENDIF

400     CONTINUE

C       If the workstation does not support prompt and echo type 3...
        IF ( PROMPT_FLAG .EQ. 0 ) THEN
             WRITE(6,*) 'The workstation does not support'
             WRITE(6,*) 'choice prompt and echo type 3.'
             STOP
        ENDIF

        PROMPT_ECHO_TYPE = 3
        INITIAL_CHOICE = 1
        INITIAL_STATUS = GKS$K_STATUS_OK

C       Establish sizes of prompt strings...
        SIZES( 1 ) = 3
        SIZES( 2 ) = 2
C       Establish locations of prompt strings...
        ADDRESSES( 1 ) = %LOC( 'Yes' )
        ADDRESSES( 2 ) = %LOC( 'No' )

C       Write the message...
        CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
        CALL GKS$TEXT( 0.05, 0.25,
      * 'Choose YES if you are satisfied with' )
        CALL GKS$TEXT( 0.05, 0.15,
      * 'your input, otherwise choose NO.' )
        CALL GKS$TEXT( 0.05, 0.02,
      * '(Move indicator and trigger input.)' )

C       Since the device is in request mode by default, initialize the device...
        CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
      * INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
      * ECHO_AREA, DATA_RECORD, RECORD_BUFFER_LENGTH )
```

(continued on next page)

```
      CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

      CALL GKS$REQUEST_CHOICE( WS_ID, DEVICE_NUM, INPUT_STATUS,
     * FINISHED_FLAG )

      RETURN
      END

C     ************************************************************
C     This program accepts input twice from the same spot on the
C     workstation surface.  When the input priority is changed,
C     the world coordinates returned are that of the other overlapping
C     viewport.
      SUBROUTINE VIEW_PRIORITY( WS_ID, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, ERROR_STATUS, INPUT_MODE, ECHO_FLAG, XFORM,
     * RECORD_BUFFER_LENGTH, RECORD_SIZE, INPUT_STATUS, DEFAULT,
     * LOW_LEFT_CORNER, RIGHT_HALF, DEVICE_NUM, NUM_POINTS,
     * PROMPT_ECHO_TYPE, DATA_RECORD( 1 ), WS_TYPE, DUMMY_INTEGER,
     * NUM_LOC_DEVICES, FOREGROUND
      REAL WORLD_COORD_X, WORLD_COORD_Y,
     * ECHO_AREA( 4 ), PX( 5 ), PY( 5 ), PX_2( 5 ), PY_2( 5 ),
     * LARGER
      DATA PX / 0.0, 1.0, 1.0, 0.0, 0.0 /
      DATA PY / 0.0, 0.0, 1.0, 1.0, 0.0 /
      DATA DEFAULT / 0 /, DEVICE_NUM / 1 /, LARGER / 0.04 /,
     * RIGHT_HALF / 1 /, LOW_LEFT_CORNER / 1 /, NUM_POINTS / 5 /,
     * FOREGROUND / 1 /

C     Make sure that the device supports locator input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * NUM_LOC_DEVICES, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER )

      IF ( NUM_LOC_DEVICES .EQ. 0 ) THEN
          WRITE(6,*) 'The workstation does not support'
          WRITE(6,*) 'locator input.'
          STOP
      ENDIF

C     When you outline the entire default world coordinate space, you
C     also outline the entire NDC space, the entire workstation window,
C     and the entire workstation viewport.
      CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_SOLID )
      CALL GKS$POLYLINE( NUM_POINTS, PX, PY )
```

(continued on next page)

Example 6–1 (Cont.): Using the DEC GKS Input Functions

```
C     This window and viewport are associated with the
C     normalization transformation number 1.
      CALL GKS$SET_WINDOW( LOW_LEFT_CORNER, 0.0, 0.5, 0.0, 0.5 )
      CALL GKS$SET_VIEWPORT( RIGHT_HALF, 0.5, 1.0, 0.0, 1.0 )

C     Select the new transformation and outline the new windows
C     and viewports.
      CALL GKS$SELECT_XFORM( 1 )
      CALL GKS$POLYLINE( NUM_POINTS, PX, PY )

C     Assign a value to RECORD_BUFFER_LENGTH: 4 bytes.  On output,
C     this argument should contain the value 0 since
C     GKS$INQ_LOCATOR_STATE does not write anything to the buffer.
      RECORD_BUFFER_LENGTH = 4
      CALL GKS$INQ_LOCATOR_STATE( WS_ID, DEVICE_NUM,
     * GKS$K_VALUE_REALIZED, ERROR_STATUS, INPUT_MODE, ECHO_FLAG,
     * XFORM, WORLD_COORD_X, WORLD_COORD_Y, PROMPT_ECHO_TYPE,
     * ECHO_AREA, DATA_RECORD, RECORD_BUFFER_LENGTH,
     * RECORD_SIZE )

C     Since the device is in request mode by default, initialize the device...
      CALL GKS$INIT_LOCATOR( WS_ID, DEVICE_NUM, 0.7, 0.5, DEFAULT,
     * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
     * RECORD_BUFFER_LENGTH )

      CALL GKS$SET_LOCATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

C     ********************************
C     At this pause, just type RETURN.
C     ********************************
      CALL GKS$REQUEST_LOCATOR( WS_ID, DEVICE_NUM, INPUT_STATUS,
     * XFORM, WORLD_COORD_X, WORLD_COORD_Y )
C     Write the returned world coordinates.
      WRITE(5, *) WORLD_COORD_X, WORLD_COORD_Y
      CALL GKS$SELECT_XFORM( DEFAULT )
      CALL GKS$SET_TEXT_COLOR_INDEX( FOREGROUND )
      CALL GKS$SET_TEXT_HEIGHT( LARGER )
      CALL GKS$TEXT( 0.01, 0.4, 'Higher priority VP: 0')


C     Set the current viewport (associated with the selected
C     transformation number 1) to be a higher priority than the
C     default viewport.
      CALL GKS$SET_VIEWPORT_PRIORITY( RIGHT_HALF, DEFAULT,
     * GKS$K_INPUT_PRIORITY_HIGHER )

C     Since the device is in request mode by default, initialize the device...
      CALL GKS$INIT_LOCATOR( WS_ID, DEVICE_NUM, 0.7, 0.5, DEFAULT,
     * PROMPT_ECHO_TYPE, ECHO_AREA, DATA_RECORD,
     * RECORD_BUFFER_LENGTH )
      CALL GKS$SET_LOCATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
```

(continued on next page)

Example 6–1 (Cont.): Using the DEC GKS Input Functions

```
C     *********************************
C     At this pause, just type RETURN.
C     *********************************
      CALL GKS$REQUEST_LOCATOR( WS_ID, DEVICE_NUM, INPUT_STATUS,
     * XFORM, WORLD_COORD_X, WORLD_COORD_Y )

C     Write the returned world coordinates, this time from the smaller
C     viewport on the right half of the screen.
      WRITE(5, *) WORLD_COORD_X, WORLD_COORD_Y
      CALL GKS$SELECT_XFORM( DEFAULT )
      CALL GKS$TEXT( 0.01, 0.3, 'Higher priority VP: 1')

      RETURN
      END
         .
         .
         .
```

Chapter 7

Sampling Input and Generating Events

This chapter describes sample and event input operating modes, and discusses the following related issues in detail:

- Concurrent activation of several logical input devices
- Differences among the three input operating modes
- Help screens using segment visibility
- Storage and restoration of current input values

NOTE

Section 7.5 contains the code that you must add to the Starry Night program in Example 3–2 to produce the program example contained in this chapter. You may wish to add this code to the base program so that you can execute the program while reading this chapter. The lines of blue code in the example signify the new code that you need to add to Example 3–2.

7.1 Choosing an Input Operating Mode

Chapter 6, Requesting Input, discusses the general concepts concerning the DEC GKS input process. All program examples used in the chapter place the logical input devices in request input operating mode. Most of the programming techniques described in the chapter are also applicable when using either sample or event mode. For instance, no matter which input operating mode your application requires, you need to remember the following issues.

- That your workstation supports input.
- That your workstation supports the desired prompt and echo types.

- That you properly obtain current and default input values as needed.
- That you properly define the input data record.
- That you initialize logical input devices, if necessary, using one of the GKS$INIT_class functions.

  To use one of the GKS$INIT_class functions, you must make sure that the input device's prompt is not currently on the workstation surface. So, to initialize a device, you must make sure that the device is set to request mode (this is the DEC GKS default mode).
- That the logical input device be in the desired mode (if need be, by calling one of the GKS$SET_class_MODE functions).

This chapter concentrates on the differences among the three input operating modes so that you can decide which modes are necessary for your application. For this reason, this chapter does not explain, in detail, concepts already outlined in Chapter 6, Requesting Input.

The first decision that you must make when choosing an input operating mode is whether you need to use a synchronous or asynchronous input operating mode. A synchronous input operating mode synchronizes the process so that the application pauses when the user enters input and continues when the user triggers the input device. An asynchronous input operating mode allows the application to continue to execute while the user enters input on the logical input devices. Consequently, if your application cannot perform any work until it receives input from the user, then you should use the DEC GKS synchronous input operating mode (request mode). If your application should continue to process as the user enters input values, then you should use one of the DEC GKS asynchronous input operating modes (sample or event mode).

If you decide to use sample or event mode, you need to decide who, according to the needs of your application, should have more control over the entering of input values: the application program or the user.

During sample mode, the user can change the current measure of a given device by altering the position of the prompt, but the user cannot trigger the device. At any time determined by the application program, the application samples (takes) the current measure of the device. The user specifies possible input values, but the application controls when values are actually accepted. When the application chooses, it ends the sampling input process.

During event mode, the user can change the measure of the device and trigger a desired value in the same manner as in request input mode. However, in event mode, the input prompt does not dissappear when the user triggers the device. Every time the user triggers the device, the action generates an event *report*. The DEC GKS input process places these reports on a time-ordered queue (first in, first out). When the application chooses, it removes the reports and processes the input. Also, when the application chooses, it ends the event input process.

Consequently, if you want the application to control the timing of the acceptance of input values, use sample mode. If you want the user to control the entering of input values, use event mode.

The following subsections discuss issues involved in choosing an appropriate input operating mode for your application program.

7.1.1 Logical Input Devices and Asynchronous Input

When using only request mode, the device handler does not place the logical input device's prompt on the workstation surface until you call one of the GKS$REQUEST_class functions. Since the application pauses until one request for input is complete, you can have at most one logical input device prompt present on the workstation surface at any one time. If using only request mode, you have need for only one logical input device (numbered 1) for each of the input classes (choice, locator, pick, string, stroke, and valuator).

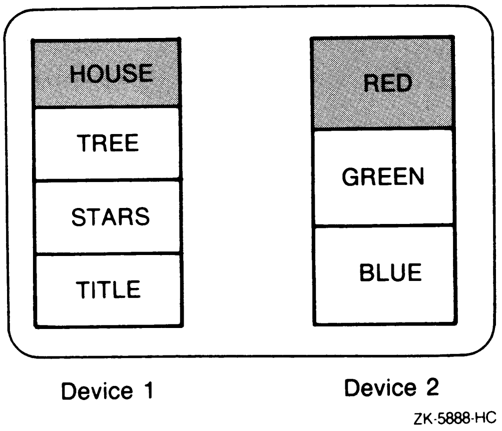
When you use sample and event input modes, the prompt appears on the workstation surface as soon as you call the appropriate GKS$SET_class_MODE function. Since you can call the GKS$SET_class_MODE function for several devices, possibly of the same class, setting each to any of the three modes, DEC GKS provides you with more than one logical input device number for each class. (To review the DEC GKS supported input devices, refer to Appendix J, DEC GKS Specific Input Values, in the *DEC GKS Reference Manual.*)

For example, depending on the needs of your application, you can place two choice devices in any of the following combination of input operating modes:

- Both in sample mode, by calling GKS$SET_CHOICE_MODE for each device.
- Both in event mode, by calling GKS$SET_CHOICE_MODE for each device.
- One in sample and one in event mode, by calling GKS$SET_CHOICE_MODE for each device.
- One device in either sample or event mode, and the other device in request mode. To do this, you call GKS$SET_CHOICE_MODE for each device, followed by a call to GKS$REQUEST_CHOICE to activate the other device's prompt. (The user can only view the prompt of a single device in request mode at any given time.)

When you activate two choice devices, you can use the choice device numbers 1 and 2. To do this, you must make sure that the echo areas for both choice menus do not overlap. Figure 7–1 illustrates two choice devices on the workstation surface.

Figure 7–1: Activating Two Input Devices of the Same Class



The following questions may arise when using several logical input devices concurrently:

- Does the user alter the measure and trigger the devices using the same physical device (for instance, keys on the keyboard)?
- If the input devices do use the same physical device, how can the user alter the measure and trigger one device without altering the measure and triggering the second device?
- If the input devices do not use the same physical device, how does the user know how to manipulate each device?

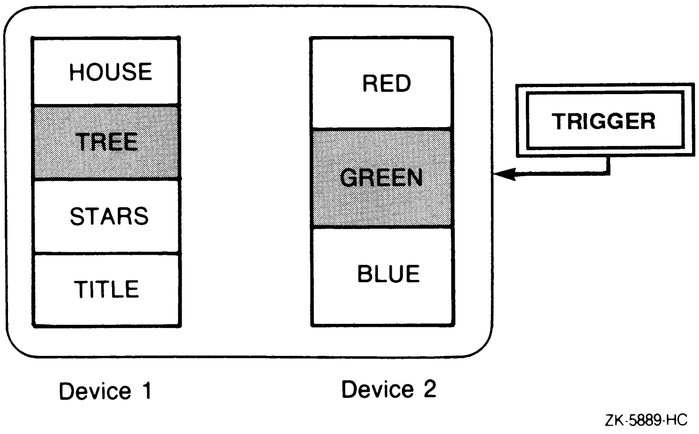
The answer to each question depends on the workstation you are using. Using a VT240, all devices are active concurrently. So, if the user presses a key that affects both devices, both devices reflect the change in their measures.

For example, choice device number 1 requires that the user press the arrow keys to change the measure and the RETURN key to trigger the device.

Choice device number 2 uses the VT240 keypad differently. If the user presses either the arrow keys or one of the numbered numeric keypad keys, the handler highlights the appropriate choice (if the user pressed a numbered key, the handler highlights the choice whose number corresponds to the key), and triggers the device immediately. Using device number 2, pressing the RETURN key has no effect.

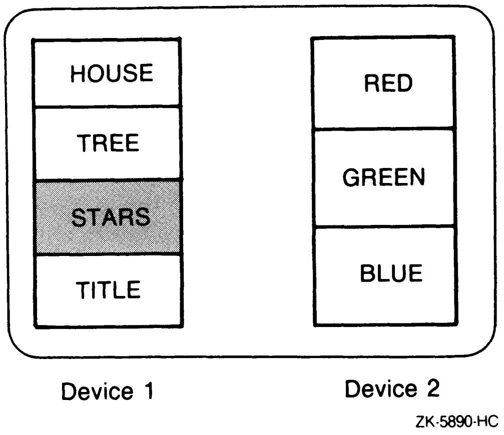
Figure 7–2 illustrates the effect of the user pressing the DOWN arrow key. Notice that the handler only changes the current measure of the first device, but triggers the second device. If the second device is in request mode, the prompt is removed from the workstation surface.

Figure 7–2: Changing the Measure of Two Active Choice Devices



Using the VT240, you need to press the PF1 key, on the numeric keypad, in order to *cycle* logical input devices. Each time the user presses the PF1 key, a different logical input device remains active, as determined by a handler-specific order. Figure 7–3 illustrates the effect of the user pressing the PF1 key followed by pressing the DOWN arrow key. Notice how none of the choices in device 2 are shaded since that device is currently inactive. To activate the second device (and to deactivate the first), the user must press the PF1 key again.

Figure 7–3: Changing the Measure of One Active Choice Device



Finally, if you are using some other device, the results may be very different. For instance, a workstation may not support choice device number 2 using the same keyboard keys as the VT240. As another example, the VAXstations do not support cycling, but they do support a method for locking the measure of a specified device prompt (refer to Appendix J, DEC GKS Specific Input Values, in the *DEC GKS Reference Manual*).

Consequently, the use of concurrently active device prompts requires that you provide the user with a greater level of documentation. If the user does not know how to cycle through devices, or how to trigger the different devices, the user could have great difficulty providing accurate input for your application program.

Finally, you must remember that the more you rely on device-specific input devices in an application, the more implementation-specific your program becomes. If you write code that is DEC GKS specific, you should use internal documentation to tell future programmers that they may need to alter the code that manipulates concurrently active logical input devices.

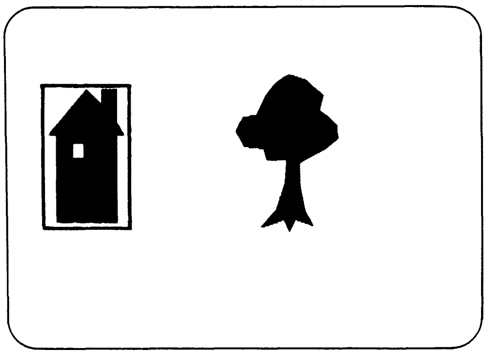
7.1.2 Sample Mode

To use sample mode, you initialize the logical input device (if the device is in request mode—the DEC GKS default mode) and set the input operating mode of the device to GKS$K_INPUT_MODE_SAMPLE. Once you call one of the GKS$SET_class_MODE functions, the device handler activates the specified logical input device and the prompt appears on the workstation surface.

Once the prompt appears on the workstation surface, the user can alter the current measure of the device. For instance, if you place a pick device in sample mode, the pick aperture appears on the surface of the workstation. At this time, the user can move the aperture, highlighting visible segments. Figure 7-4 illustrates this process.

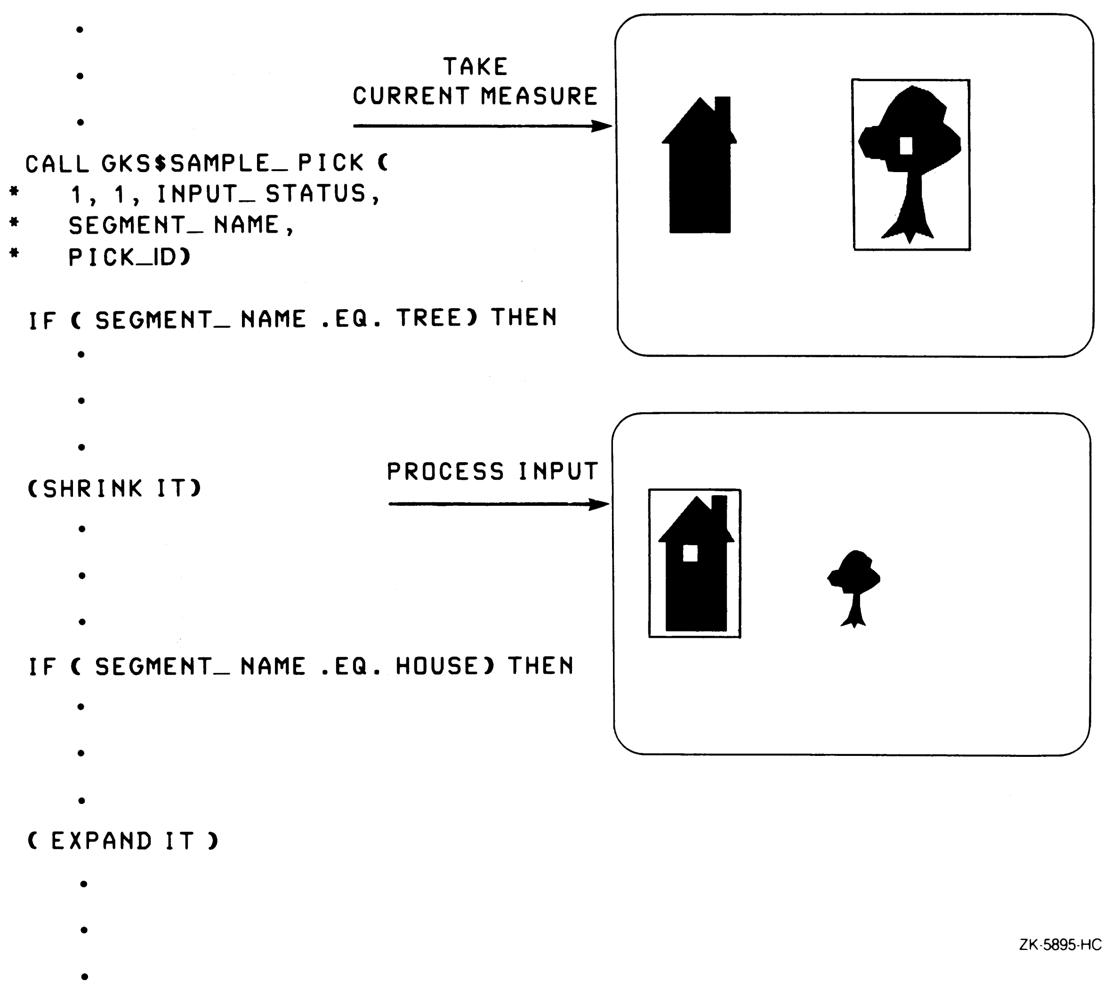
Figure 7-4: Activating a Pick Device in Sample Mode

```
         .
         .
         .
  CALL GKS$SET_PICK_MODE (
*     1, 1, GKS$K_MODE_SAMPLE,
*     GKS$K_ECHO )
         .
         .
         .
```



ZK-5892-HC

Figure 7–5: Sampling a Pick Device



```
      .
      .
      .
  CALL GKS$SAMPLE_ PICK (
*    1, 1, INPUT_ STATUS,
*    SEGMENT_ NAME,
*    PICK_ID)

  IF ( SEGMENT_ NAME .EQ. TREE) THEN
      .
      .
      .
  (SHRINK IT)
      .
      .
      .
  IF ( SEGMENT_ NAME .EQ. HOUSE) THEN
      .
      .
      .
  ( EXPAND IT )
      .
      .
      .
```

In Figure 7–5, the program samples the current measure of the pick device. Notice in Figure 7–5 that you specify the following arguments, in the following order, to GKS$SAMPLE_PICK:

1. Workstation identifier—To identify the logical input device.
2. Device number—To identify the logical input device.
3. Input status—To see if the user is picking a segment or specifying GKS$K_STATUS_NOPICK.

4. Segment name—To contain the name of the currently picked segment.
5. Pick identifier—To contain the identifying number of the set of primitives currently picked.

In Figure 7–5, GKS$SAMPLE_PICK writes the integer value TREE to the SEGMENT_NAME argument. Once the application program samples the device, the program can perform tasks depending on the sampled values. In Figure 7–5, the program shrinks the segment if the sampled name is TREE and expands the segment if the sampled name is HOUSE.

After the application samples the device, it does not matter if the user moves the prompt to a different segment. Notice that in Figure 7–5, the current measure of the pick device is HOUSE, but at the time of the sampling, it was TREE. Consequently, DEC GKS shrinks the size of the tree and leaves the size of the house unaltered. The value of SEGMENT_NAME does not change until the application program calls GKS$SAMPLE_PICK again. The application program controls when an input device's measure is accepted; the user can only supply possible measures.

If you want to remove the pick prompt from the workstation surface, place the logical input device into request mode, as follows:

```
          .
          .
          .
      CALL GKS$SET_PICK_MODE( 1, 1, GKS$K_INPUT_MODE_REQUEST,
     * GKS$K_ECHO )
```

Once you set a device to request mode, the device handler removes the device's prompt from the workstation surface. At this point, you can call one of the GKS$INIT_class functions to reinitialize the device, if you choose. You can only initialize devices that are in request mode. If you need to sample the pick device in some subsequent portion of your program, simply reset the input mode to GKS$K_INPUT_MODE_SAMPLE by calling GKS$SET_PICK_MODE again. At this point, the device handler places the prompt on the workstation surface again.

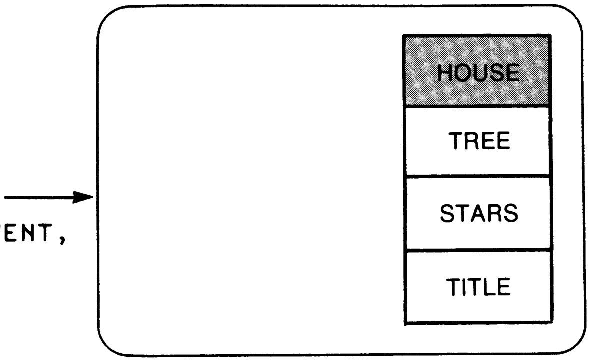
7.1.3 Event Mode

To use event mode, you initialize the logical input device (if the device is in request mode—the DEC GKS default mode) and set the input operating mode of the device to GKS$K_INPUT_MODE_EVENT. Once you call one of the GKS$SET_class_MODE functions, the device handler activates the specified logical input device and the prompt appears on the workstation surface.

Once the prompt appears on the workstation surface, the user can alter the measure and trigger the device as often as desired. For instance, if you place a choice device in event mode, the prompt appears on the workstation surface as soon as you call GKS$SET_CHOICE_MODE. Figure 7–6 illustrates the effect of such a call.

Figure 7–6: Placing a Choice Device in Event Mode

```
  .
  .
  .
  (TELL THE USER TO CHOOSE THREE
       ITEMS IN ANY COMBINATION)
  .
  .
  .
  CALL GKS$SET_ CHOICE_ MODE (
*      1, 1, GKS$K_ INPUT_ MODE_ EVENT,
*      GKS$K_ ECHO )
  .
  .
  .
```



ZK-5893-HC

Every time the user triggers the input device, the device handler places a report on the event input queue located in the DEC GKS state list. Each report includes the following information:

- The workstation identifier
- The input class of the logical input device
- The logical input device number
- Input data (varies according input class)

Figure 7–7 illustrates the generation of event input reports. If required by the application, the program can perform any number of tasks before removing reports from the queue, and can create a picture such as the one in Figure 7–8, using the information obtained from the queue.

Figure 7–7: Generating Event Input Reports

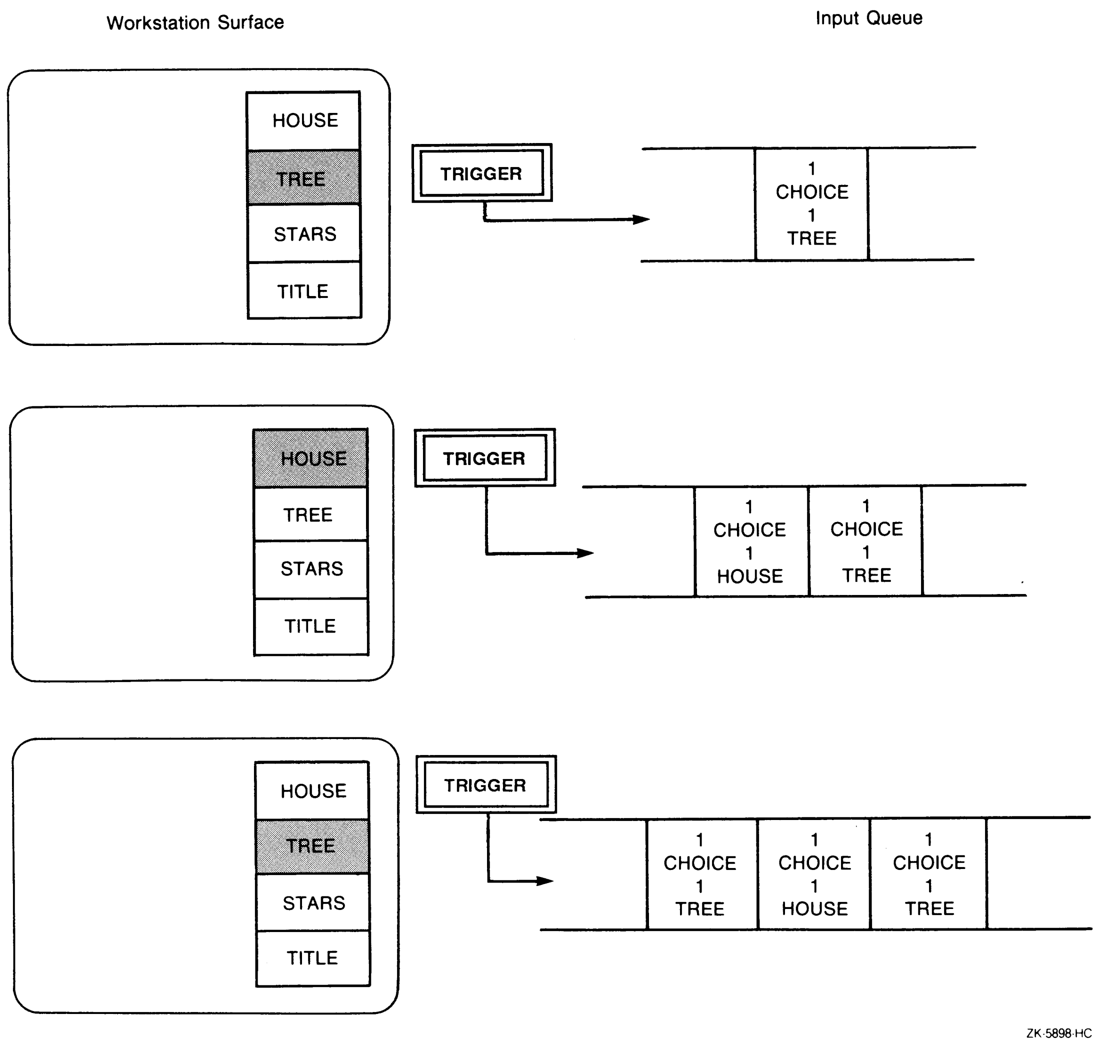
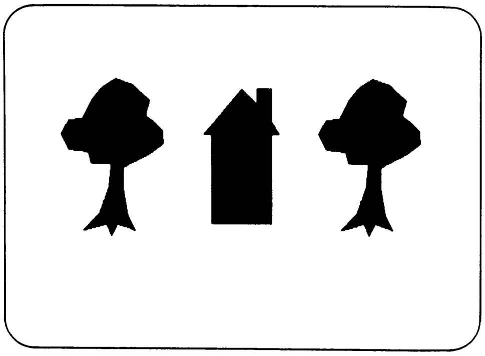


Figure 7–8: Processing Information from the Queue


ZK-5891-HC

If you want to remove the choice prompt from the workstation surface, place the logical input device into request mode, as follows:

```
          .
          .
          .
 CALL GKS$SET_CHOICE_MODE( 1, 1, GKS$K_INPUT_MODE_REQUEST,
* GKS$K_ECHO )
```

Once you set a device to request mode, the device handler removes the device's prompt from the workstation surface. At this point, you can call one of the GKS$INIT_class functions to reinitialize the device, if you choose. You can only initialize devices that are in request mode. If you need to have the user generate events using the pick device in some subsequent portion of your program, simply reset the input mode to GKS$K_INPUT_MODE_EVENT by calling GKS$SET_PICK_MODE again. At this point, the device handler places the prompt on the workstation surface again.

7.1.3.1 Removing Events from the Queue

At any time, the application program can attempt to remove reports from the event input queue. The application can reset the operating mode of the choice device to GKS$K_INPUT_MODE_REQUEST (removing the prompt from the workstation surface), perform any number of tasks, and then remove the reports from the queue when the application needs to process the input.

To remove an event input report from the queue, you call the function GKS$AWAIT_EVENT. This function provides the application with information about the oldest report on the event input queue. This function has four arguments, as follows:

- Time out period, in seconds (read only)
- Workstation identifier of the input device that generated the report (write only)
- Input class of the input device that generated the report (write only)
- Logical input device number (write only)

GKS$AWAIT_EVENT suspends the execution of your application, while checking the status of the event queue, for a length of time from zero (0) seconds up to the amount of time specified in its time-out argument. By specifying 0 seconds as a time-out argument, the handler checks the event queue immediately, without suspending program execution.

If the event queue contains at least one report, then a call to GKS$AWAIT_EVENT performs the following tasks:

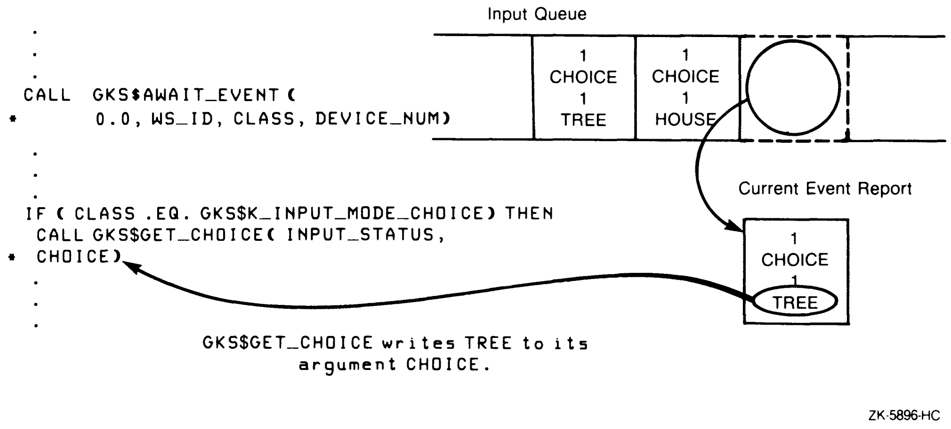
- Removes the oldest report.
- Places it in the *current event report* entry in the DEC GKS state list.
- Writes the workstation identifier, the input class, and the logical input device number of the current event report to its last three arguments.
- Allows the application to resume.

If the queue remains empty for the entire time-out period (meaning that the user has not triggered any of the devices), GKS$AWAIT_EVENT writes GKS$K_INPUT_CLASS_NONE to its input class argument and allows the application to resume.

Once you call GKS$AWAIT_EVENT, you can check the value of its input class argument. You need to know the input class of the device that generated the input located in the current event report before you can access that information. To access the current report, you call one of the GKS$GET_class functions. Figure 7–9 illustrates this process. In the figure, notice that the application checks the value of the CLASS argument (set by the call to

GKS$AWAIT_EVENT) before attempting to call GKS$GET_CHOICE. The code makes sure that a choice device generated the current event before calling GKS$GET_CHOICE. If you call a GKS$GET_class function whose class does not match the class of the input device that generated the current event report, you generate an error.

Figure 7–9: Removing Reports from the Queue



```
 .
 .
 .
 CALL  GKS$AWAIT_EVENT (
*      0.0, WS_ID, CLASS, DEVICE_NUM)
 .
 .
 .
 IF ( CLASS .EQ. GKS$K_INPUT_MODE_CHOICE) THEN
  CALL GKS$GET_CHOICE( INPUT_STATUS,
* CHOICE)
 .
 .
 .
```

Remember that the information in the current event report does not change until you call GKS$AWAIT_EVENT to retrieve another report from the queue. If you do not call GKS$AWAIT_EVENT, repeated calls to one of the GKS$GET_class functions obtain the same set of input information from the current event report.

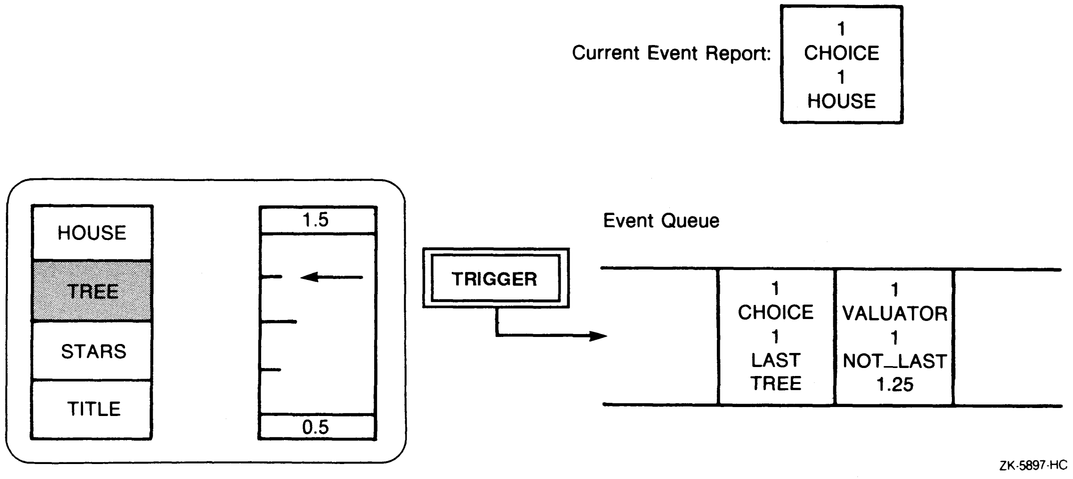
After you process the information in the current report, you can call GKS$AWAIT_EVENT, repeatedly, until either you obtain all the input you need or until the queue is empty. If the queue is empty, GKS$AWAIT_EVENT writes GKS$K_INPUT_CLASS_NONE to its class argument. Once you encounter the value GKS$K_INPUT_CLASS_NONE, you can either call GKS$AWAIT_EVENT to see if the user has generated an event since the last time you checked the queue, or you can end the event input portion of your application program.

7.1.3.2 Simultaneous Event Generation and Input Queue Overflow

When you use event input, there are two special circumstances to keep in mind: simultaneous events and input queue overflow. When these situations occur, you need to perform additional tasks to properly remove events from the input queue.

When the user generates an event report, there is an additional component to the event input report that specifies whether there are any remaining reports on the queue that were generated at the same time. For instance, if you have two active devices in event mode that recognize the RETURN key as a trigger, pressing the RETURN key once generates two reports simultaneously. Figure 7–10 illustrates this situation.

Figure 7–10: Generating Simultaneous Event Reports



Notice the current event report values in Figure 7–10. The currently chosen value is HOUSE. If the device handler chooses to enter the events into the queue in the order shown (report entry on the queue is completely handler dependent), and if the application processes the information one event report at a time, the application may scale HOUSE (the current value) when the user actually wanted to scale TREE (the value simultaneously entered). To prevent a situation like this, you need to check to see if the event taken from the queue is the last in a series of simultaneously generated events. To check for this situation, you use the following code:

```
         .
         .
         .
      CALL GKS$INQ_MORE_SIMUL_EVENTS( ERROR_STATUS, EVENTS_FLAG )
```

If the argument EVENTS_FLAG equals GKS$K_MORE_EVENTS, there are additional reports on the queue that the user generated at the same time as the report you just removed from the queue. If the order of report processing needs to be precise, you can perform the following tasks:

1. Remove a report from the queue.
2. Call GKS$INQ_MORE_SIMUL_EVENTS.
3. Check the EVENTS_FLAG argument.
4. If EVENT_FLAG equals GKS$K_MORE_EVENTS, remove another report from the queue and repeat this process. If EVENT_FLAG equals GKS$K_NOMORE_EVENTS, you have removed all simultaneously generated event reports.

Once you remove all simultaneously generated reports, your application can decide in which order to process the input.

If you activate more than one device in event mode, and if the user chooses not to cycle through the devices while entering input, rapid triggering of many devices may cause the input queue to overflow. In this situation, the device handler does not accept additional reports until you completely clear the queue of all reports. Usually, you check for input queue overflow immediately after a call to GKS$AWAIT_EVENT, since that is the point in your application when it would be most helpful to know the status of the event input queue. To test for input queue overflow, you use the following code:

```
         .
         .
         .
      CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )
```

```
 CALL GKS$INQ_INPUT_QUEUE_OVERFLOW( ERROR_STATUS, WS_ID,
* CLASS, DEVICE_NUM )
     .
     .
     .
```

To determine whether the input queue has overflowed, you need to check the ERROR_STATUS argument. If ERROR_STATUS is equal to the value 0, then the following is true:

- The event input queue has overflowed.
- Information about the overflow is available.
- GKS$INQ_INPUT_QUEUE_OVERFLOW writes the workstation identifier, the input class, and the logical device number of the device that caused the overflow, to its output arguments.

If ERROR_STATUS is not equal to the value 0, then the queue may or may not have overflowed. However, if the queue did overflow, there is no information available about what caused the overflow (the workstation associated with the input device that caused the overflow has been closed). In this case, ERROR_STATUS can equal one of the following values:

- GKS$K_ERROR_7—GKS not in proper state.
- GKS$K_ERROR_148—Input queue has not overflowed since GKS was opened or since the last invocation of INQUIRE INPUT QUEUE OVERFLOW.
- GKS$K_ERROR_149—Input queue has overflowed, but the associated workstation has been closed.

If the event input queue overflows, you should deactivate all devices currently in event mode (by using the appropriate GKS$SET_class_MODE funtion to place them in request mode) so that the user cannot attempt to generate additional reports. Then, you can continue to call GKS$AWAIT_EVENT, removing the reports one by one until a call returns GKS$K_INPUT_CLASS_NONE, signaling an empty queue. As you remove reports from the queue, you can continue to process the input if your application requires. Once the queue is empty, you can place the devices in event mode again, allowing the user to generate additional reports.

As a second option, you can call the function GKS$FLUSH_DEVICE_EVENTS to remove the remaining reports generated by a device of a single input class. By calling GKS$FLUSH_DEVICE_EVENTS for all possible logical input classes, you clear the buffer and allow the user to enter input again. Assuming the active logical input devices shown in Figure 7–10, the following code clears the event input queue.

```
         .
         .
         .
 CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )

 CALL GKS$INQ_INPUT_QUEUE_OVERFLOW( ERROR_STATUS, WS_ID,
* CLASS, DEVICE_NUM )

 IF ( ERROR_STATUS .EQ. 0 ) THEN
    CALL GKS$FLUSH_DEVICE_EVENTS( 1,
*        GKS$K_INPUT_CLASS_VALUATOR, 1 )
    CALL GKS$FLUSH_DEVICE_EVENTS( 1, GKS$K_INPUT_CLASS_CHOICE,
*        1 )
 ENDIF
```

7.1.4 Input Operating Mode Differences

When you choose one of the three available input operating modes, it is helpful to see the differences when all three modes are used in a similar application.

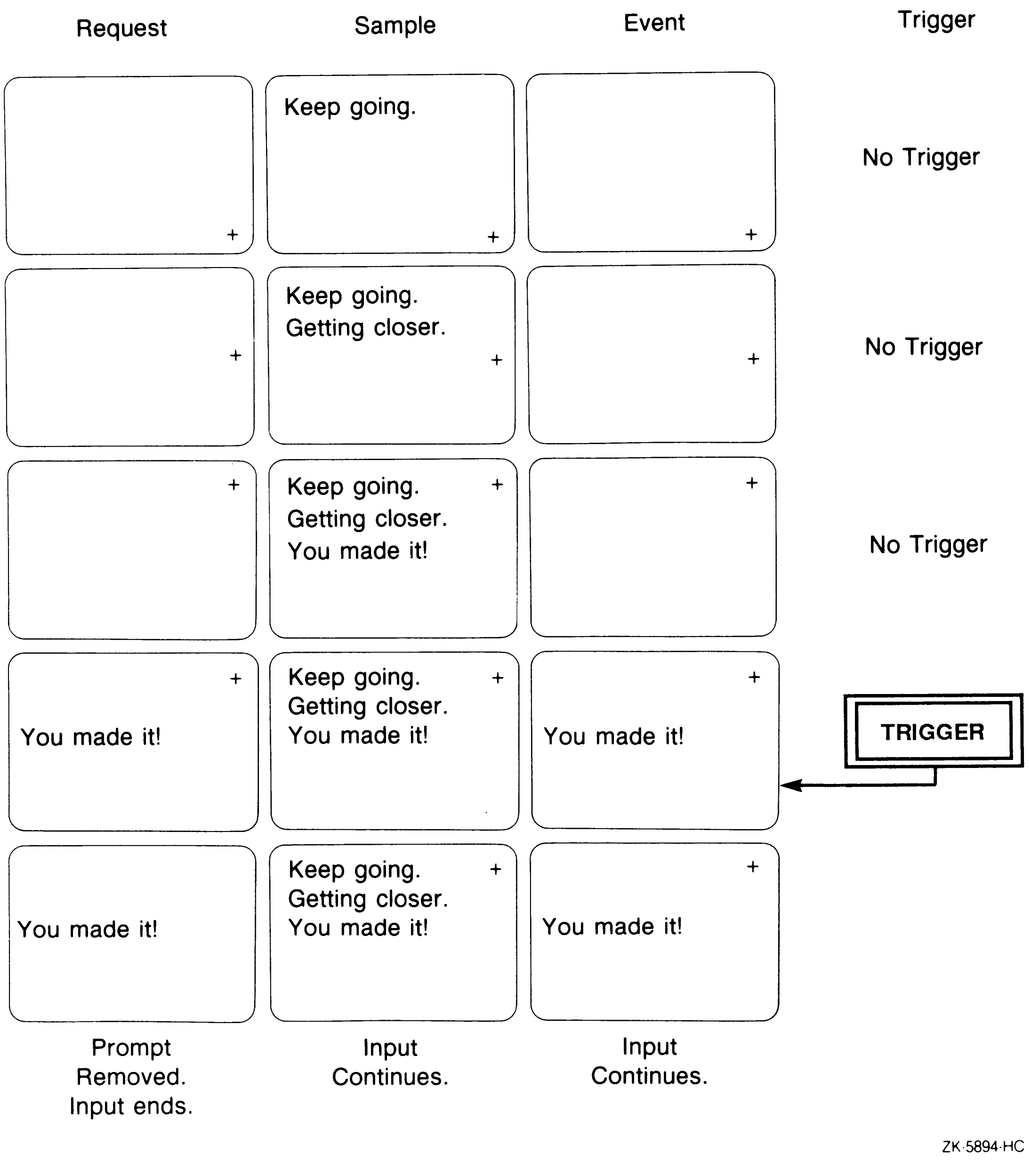
Imagine a program that uses locator input. The program places the cursor at the bottom of the workstation surface (a Y value of 0) and prompts the user to move the cursor to the top of the surface (a maximum Y value).

Using request mode, you can only determine the position of the locator prompt once. As soon as the user moves the locator prompt and then triggers the device, the input process ends (until you call GKS$REQUEST_LOCATOR again).

Using sample and event mode, the program can loop to continually check for a change in the input value. Using sample mode, the application can continually monitor the position of the prompt without a trigger from the user. Using event mode, the application is not aware of a change in the position of the locator prompt until the user triggers the device (placing a report on the queue), and until the application removes the report from the queue and processes the input.

Clearly, such an application favors sample input, since the application can easily and continually monitor the position of the locator prompt. Using this example, you can compare the differences in the ways in which the operating modes perform. In this way, you can judge which input operating mode best serves a particular type of application. Figure 7–11 illustrates the locator input example described in this section.

Figure 7–11: Comparing the Three Input Operating Modes



7.2 Documenting Logical Input Devices

When using only a single logical input device of each class in request mode, the user needs minimal documentation to understand how to operate the device. By default, most of the devices require the use of arrow keys and a mouse (to alter the measure of the device), the RETURN key and the mouse buttons (to trigger the device), and the keyboard (to type a string).

When using more than one active logical input device and when specifying more than one input operating mode, you need to provide the user with more documentation. For instance, the user needs to know the following:

- What type of input information each device accepts, and how the application uses this information.
- How to change the measure and trigger each device.
- How to cycle through devices.
- How to end the input process.

The following list presents several ways to provide the user with adequate documentation:

- Input instructions located in a distinct window. (This method is used in the program example in Chapter 6, Requesting Input.)
- Input instructions listed at the beginning of the input process and cleared from the surface once input begins.
- A help screen using segment visibility to control its presence on the workstation surface.
- Written documentation available to the users at the time of application execution.
- Labels placed on the top of the input device echo areas. (For more information, refer to Appendix J, DEC GKS Specific Input Values, in the *DEC GKS Reference Manual.*)

Although you can choose any method to document your application, these sections show you how to use segment visibility to hide and to present a help screen.

When you decide to use segment visibility to hide a help screen from the user (until the user requests help), you need to find a method of creating the segment without having the text appear on the active workstation's surface. One way of accomplishing this is to perform the following tasks.

- Deactivate the GKS$K_WSCAT_OUTIN workstation so that the help text does not appear on its surface.
- Activate workstation independent segment storage (WISS) to store the help screen segment.
- Create the help screen segment.
- Set the visibility of the help screen segment to GKS$K_INVISIBLE (you don't want the help text to be visible until the user asks for help).
- Associate the help screen segment to the GKS$K_WSCAT_OUTIN workstation.
- Deactivate WISS (no other segments need be stored in WISS).
- Activate the GKS$K_WSCAT_OUTIN workstation to enable further output.

When you finish performing these tasks, the GKS$K_WSCAT_OUTIN workstation contains an invisible segment. When requiring help, the user signals the application. The application makes the segments in the current picture invisible, deactivates the input devices, and makes only the help text visible. When the user chooses, the application resets visibility attributes, and the user can once again view the picture and enter valid input.

Example 7–1 uses a help screen to inform the user about logical input devices. The following sections explain the code in Example 7–1 that establishes a help screen.

7.2.1 Using Workstation Independent Storage (WISS)

The following code from Example 7–1 is identical to the code used to work with WISS in Chapter 5, Generating Output:

```
        IMPLICIT NONE
        INTEGER WS_ID, WISS, HOUSE, TREE, HORIZON, STARS, TITLE,
       * SIDE, ROAD

        DATA WS_ID / 1 /, WISS / 2 /, TITLE / 1 /, STARS / 2 /,
       * TREE / 3 /, SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /,
       * HOUSE / 7 /

❶       CALL SET_UP( WS_ID, WISS )
        CALL DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
       * ROAD, HOUSE, HORIZON )
        CALL CLEAN_UP( WS_ID, WISS )

        END

   C    *************************************************************
   C    Set up the DEC GKS and the workstation environments...
        SUBROUTINE SET_UP( WS_ID, WISS )
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, WISS,
            .
            .
            .
      * GKS_LEVEL
            .
            .
            .
  C     Make sure that WISS is supported.
❷       CALL GKS$INQ_LEVEL( ERROR_STATUS, GKS_LEVEL )

        IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
      *    ( GKS_LEVEL .LT. GKS$K_LEVEL_2A )) THEN
               WRITE(6,*)
      *        'This level of GKS does not support WISS.'
               WRITE(6,*) 'Error status:', ERROR_STATUS
               STOP
        ENDIF

  C     Open WISS so that you can store the help information.
        CALL GKS$OPEN_WS( WISS, GKS$K_CONID_DEFAULT,
      * GKS$K_WSTYPE_WISS )
            .
            .
            .
```

The following numbers correspond to the numbers in the previous example:

❶ Pass the argument WISS (2), the workstation identifier for WISS, to both the SET_UP and CLEAN_UP subroutines. In this way, each workstation is opened and activated, if appropriate.

❷ Check to make sure that the implementation level of GKS is at least level 2a. GKS level 2a and above support segments stored on WISS.

7.2.2 Defining the Input Subroutine

Example 7–1 establishes a single subroutine that controls the entire input process for the application. Chapter 6, Requesting Input, uses a similar subroutine. The subroutine GO_FOR_INPUT is as follows:

```
            .
            .
            .
  C     ************************************************************
  C     Draw the picture, and place each primitive in a segment...
        SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
      * ROAD, HOUSE, HORIZON )
            .
            .
            .
```

```
   C    Ask the user for input...
❶       CALL GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS, TREE,
       * SIDE, ROAD, HOUSE, HORIZON )

        RETURN
        END
            .
            .
            .
   C    ************************************************************
   C    Coordinate user input...
        SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
       * TREE, SIDE, ROAD, HOUSE, HORIZON )

        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
       * HORIZON, CATEGORY, ERROR_STATUS, FILL_PTS, FOREGROUND,
       * BACKGROUND, UNITY, HELP, HELP_BOX, WS_TYPE, WISS
        REAL FILL_X( 5 ), FILL_Y( 5 ), TEXT_EXTENT_X( 4 ),
       * TEXT_EXTENT_Y( 4 ), DUMMY_REAL( 4 )

        DATA FILL_PTS / 5 /, FOREGROUND / 1 /, BACKGROUND / 0 /,
❷      * UNITY / 0 /, HELP / 8 /, HELP_BOX / 9 /, WISS / 2 /

        DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
        DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

        CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
       * ERROR_STATUS, CATEGORY )

        IF ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ) THEN
             WRITE(6,*) 'The workstation does not support'
             WRITE(6,*) 'both input and output.'
             STOP
        ENDIF

   C    Make sure that you are using the unity transformation...
        CALL GKS$SELECT_XFORM( UNITY )

   C    Fill an area on which to send the user a message...
        CALL GKS$SET_FILL_COLOR_INDEX( FOREGROUND )
        CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )

   C    Create the help segment.
❸       CALL CREATE_HELP( WS_ID, WISS, HELP )

   C    Set the proper text attributes.
❹       CALL GKS$SET_TEXT_HEIGHT( 0.04 )
        CALL GKS$SET_TEXT_COLOR_INDEX( BACKGROUND )
        CALL GKS$SET_TEXT_SPACING( -0.3 )
```

```
  C     Create the HELP/EXIT segments.
❺       CALL GKS$CREATE_SEG( HELP_BOX )
        CALL GKS$INQ_TEXT_EXTENT( WS_ID, 0.65, 0.9, 'HELP/EXIT',
       * ERROR_STATUS, DUMMY_REAL, DUMMY_REAL, TEXT_EXTENT_X,
       * TEXT_EXTENT_Y )
        TEXT_EXTENT_X( 1 ) = TEXT_EXTENT_X( 1 ) - 0.01
        TEXT_EXTENT_X( 4 ) = TEXT_EXTENT_X( 4 ) - 0.01
        TEXT_EXTENT_X( 2 ) = TEXT_EXTENT_X( 2 ) + 0.01
        TEXT_EXTENT_X( 3 ) = TEXT_EXTENT_X( 3 ) + 0.01
        TEXT_EXTENT_Y( 1 ) = TEXT_EXTENT_Y( 1 ) - 0.01
        TEXT_EXTENT_Y( 2 ) = TEXT_EXTENT_Y( 2 ) - 0.01
        TEXT_EXTENT_Y( 3 ) = TEXT_EXTENT_Y( 3 ) + 0.01
        TEXT_EXTENT_Y( 4 ) = TEXT_EXTENT_Y( 4 ) + 0.01
        CALL GKS$FILL_AREA( 4, TEXT_EXTENT_X, TEXT_EXTENT_Y )
        CALL GKS$TEXT( 0.65, 0.9,
       * 'HELP/EXIT' )
        CALL GKS$CLOSE_SEG()

  C     Initialize all input devices.
❻       CALL INIT_DEVICES( WS_ID, WS_TYPE )

  C     Make sure that all of the segments are detectable...
❼       CALL GKS$SET_SEG_DETECTABILITY( TITLE, GKS$K_DETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( STARS, GKS$K_DETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( TREE, GKS$K_DETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( SIDE, GKS$K_DETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( ROAD, GKS$K_DETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( HOUSE, GKS$K_DETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( HORIZON, GKS$K_DETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( HELP, GKS$K_UNDETECTABLE )
        CALL GKS$SET_SEG_DETECTABILITY( HELP_BOX, GKS$K_DETECTABLE )

  C     Reset the attribute values and the message board.
        CALL GKS$SET_TEXT_HEIGHT( 0.033 )
        CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )

❽       CALL GKS$TEXT( 0.05, 0.25,
       * 'When prompted, scale the picture elements.' )
        CALL GKS$TEXT( 0.05, 0.20,
       * 'If you need help or if you are ready to finish,' )
        CALL GKS$TEXT( 0.05, 0.15,
       * 'move the square prompt to HELP/EXIT.' )
        CALL GKS$TEXT( 0.05, 0.02,
       * '(Press RETURN when ready.)' )

  C     The user presses RETURN when ready to pick...
        CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
        READ(5,*)

  C     Erase the message and redraw the segments...
        CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

  C     Get the input values.
❾       CALL GET_VALUES( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
       * HOUSE, HORIZON, HELP, HELP_BOX, WS_TYPE)
```

```
C     Show the final picture...
      FILL_Y( 3 ) = 0.1
      FILL_Y( 4 ) = 0.1
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
      CALL GKS$TEXT( 0.05, 0.05,
     * 'Here is the altered picture.' )

C     Press RETURN when finished viewing the picture.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

      RETURN
      END
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

❶ This code calls the GO_FOR_INPUT subroutine from DRAW_PICTURE. GO_FOR_INPUT coordinates all of the subroutines to used to manage the input process for this application.

❷ GO_FOR_INPUT defines the two new segment names HELP and HELP_BOX. HELP is the segment containing the help screen. HELP_BOX is the rectangular area on the workstation surface, labeled HELP/EXIT, that the user needs to pick when asking for the help screen.

❸ This code calls CREATE_HELP. This subroutine creates the help screen segment and associates it with the GKS$K_WSCAT_OUTIN workstation. Section 7.2 explains this subroutine in detail.

❹ This code establishes attributes needed to write a message to the user inside of a fill area. Chapter 6, Requesting Input, uses this type of message board.

❺ This code creates a small rectangular segment labeled HELP/EXIT. The call to GKS$INQ_TEXT_EXTENT provides the rectangular dimensions of the text extent rectangle for the string HELP/EXIT. By increasing the dimensions slightly, the rectangular, hollow fill area outlines the text without obscuring it.

If there is a need to see the help screen, the user picks this segment.

❻ This call to the INIT_DEVICES subroutine initializes all of the logical input devices needed for this application. Section 7.3 explains this subroutine in detail.

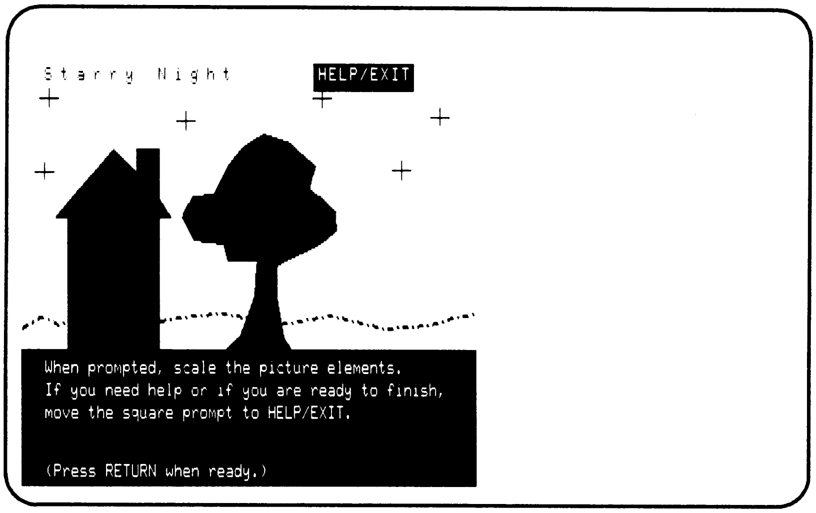
❼ This code establishes whether the user can pick specified segments (the detectability attribute). Only the help screen segment is undetectable.

❽ This code writes the text of the initial message to the message board.

❾ This code calls the subroutine GET_VALUES. This subroutine asks for input from the user, and the user can end the input process by picking the HELP/EXIT segment. Section 7.4 explains this subroutine in detail.

Figure 7–12 illustrates the surface of the workstation when the application creates the message board. Figure 7–13 illustrates the surface as soon as the application calls GET_VALUES (allowing the user to enter input).

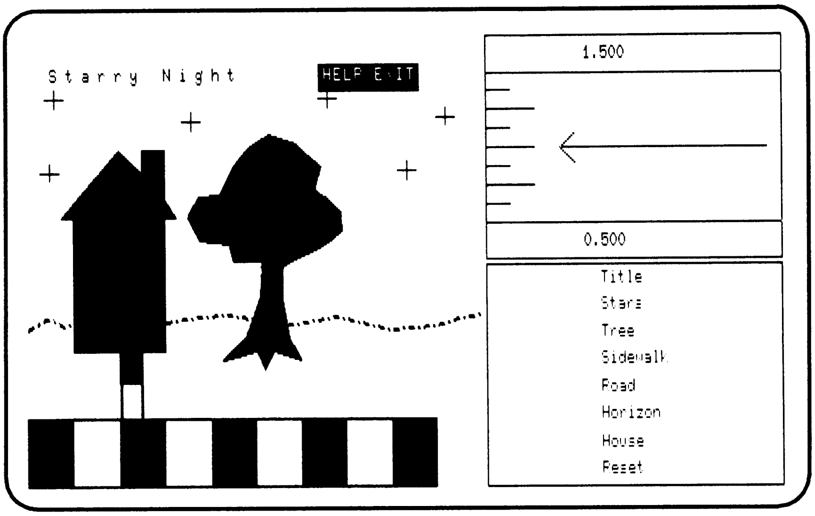
Figure 7–12: The Message Board—VT241



When prompted, scale the picture elements.
If you need help or if you are ready to finish,
move the square prompt to HELP/EXIT.

(Press RETURN when ready.)

Figure 7–13: The Initial Input Device Prompts—VT241



7.2.3 Creating the Help Segment

Before the application in Example 7–1 attempts to initialize the logical input devices, it calls the subroutine CREATE_HELP. This subroutine creates a segment containing the text of a help screen, and accomplishes this task so that the user does not view the help screen unless needed. The CREATE_HELP subroutine is as follows.

```
          .
          .
          .
C     ***********************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )
          .
          .
          .
C     Create the HELP segment.
      CALL CREATE_HELP( WS_ID, WISS, HELP )
          .
          .
          .
      RETURN
      END

C     ***********************************************************
C     This subroutine creates a HELP screen...
      SUBROUTINE CREATE_HELP( WS_ID, WISS, HELP )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WISS, HELP, BACKGROUND

      DATA BACKGROUND / 1 /

C     Only create the help screen on WISS.
❶     CALL GKS$DEACTIVATE_WS( WS_ID )
      CALL GKS$ACTIVATE_WS( WISS )

C     Set the necessary attributes.
      CALL GKS$SET_TEXT_HEIGHT( 0.033 )
      CALL GKS$SET_TEXT_SPACING( -0.4 )
      CALL GKS$SET_TEXT_COLOR_INDEX( BACKGROUND )

C     Create the help screen.
❷     CALL GKS$CREATE_SEG( HELP )
      CALL GKS$TEXT( 0.05, 0.9,
     * 'DEVICES---One device chooses a picture item, one' )
      CALL GKS$TEXT( 0.1, 0.85,
     * 'changes the current scaling value, and one either' )
      CALL GKS$TEXT( 0.1, 0.80,
     * 'stops further scaling of a specified element, or' )
      CALL GKS$TEXT( 0.1, 0.75,
     * 'it resets all elements to their original scaling' )
      CALL GKS$TEXT( 0.1, 0.70,
     * 'and enables subsequent scaling.')
      CALL GKS$TEXT( 0.05, 0.65,
     * 'CYCLING INPUT DEVICES---If you have a numeric')
      CALL GKS$TEXT( 0.1, 0.60,
     * 'keypad, you can use the two keys, in the upper' )
      CALL GKS$TEXT( 0.1, 0.55,
     * 'left corner, to turn devices on and off.' )
      CALL GKS$TEXT( 0.1, 0.50,
     * 'Otherwise, all devices move synchronously.' )
```

```
         CALL GKS$TEXT( 0.05, 0.45,
        * 'MOVING THE PROMPTS---Use whatever your device' )
         CALL GKS$TEXT( 0.1, 0.40,
        * 'normally uses to move a cursor on the surface.' )
         CALL GKS$TEXT( 0.1, 0.35,
        * 'This can include arrow keys, a mouse, a puck,' )
         CALL GKS$TEXT( 0.1, 0.30,
        * 'or a joy disk.' )
         CALL GKS$TEXT( 0.05, 0.25,
        * 'ENTERING VALUES---To enter values, you need to ' )
         CALL GKS$TEXT( 0.1, 0.20,
        * 'use whatever your device normally uses, such as ' )
         CALL GKS$TEXT( 0.1, 0.15,
        * 'the RETURN key, mouse button, or puck button; to' )
         CALL GKS$TEXT( 0.1, 0.10,
        * 'pick an element, all you have to do is align the ' )
         CALL GKS$TEXT( 0.1, 0.05,
        * 'rectangular prompt with the element you choose.' )
         CALL GKS$TEXT( 0.1, 0.001,
        * 'Do you want to QUIT or CONTINUE?' )

         CALL GKS$CLOSE_SEG()

    C    Associate the help screen and reactivate the workstation.
❸        CALL GKS$SET_SEG_VISIBILITY( HELP, GKS$K_INVISIBLE )
❹        CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, HELP )
❺        CALL GKS$DEACTIVATE_WS( WISS )
         CALL GKS$ACTIVATE_WS( WS_ID )

         RETURN
         END
             .
             .
             .
```

The following numbers correspond to the numbers in the previous example:

❶ When you deactivate WS_ID, generated output does not appear on the workstation surface. By activating WISS, you can place the help screen text in a stored segment.

❷ This code creates the text of the help screen. The text explains to the user how to use the logical input devices used in this application.

❸ By making the segment invisible, it will not appear on the workstation surface once you associate the help screen segment with WS_ID.

❹ This code associates the help screen segment, stored in WISS, with the open workstation (WS_ID).

❺ Once you deactivate WISS, you cannot store segments on WISS. By activating WS_ID, you enable further generation of output on the workstation surface.

7.3 Initializing the Logical Input Devices

This section describes the INIT_DEVICES subroutine in Example 7–1. However, this section does not repeat basic information about input device initialization already presented in Chapter 6, Requesting Input.

When using sample and event mode, you need to be careful when initializing devices. Since you cannot initialize a device whose prompt is currently present on the surface of the workstation, you need to make sure that a device is set to request mode (the DEC GKS default mode) before you call one of the GKS$INIT_class functions.

The first two sections explain the initial values of the three logical input devices used in Example 7–1. The last section describes precautions you need to take when defining echo areas of concurrently active input devices.

7.3.1 Initializing the Pick Device

The following code is the initial section of the INIT_DEVICES subroutine in Example 7–1, which includes the pick device initialization.

```
              .
              .
              .
C     ************************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )
              .
              .
              .
C     Initialize all input devices.
      CALL INIT_DEVICES( WS_ID, WS_TYPE )
              .
              .
              .
      RETURN
      END
C     ************************************************************
C     This subroutine initializes all input devices...
      SUBROUTINE INIT_DEVICES( WS_ID, WS_TYPE )
```

```
        IMPLICIT NONE
        INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
        INTEGER WS_ID, WS_TYPE, ERROR_STATUS, DUMMY_INTEGER,
       * NUM_PICK_DEVICES, DEVICE_NUM, INPUT_MODE, ECHO_FLAG,
       * INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
       * PROMPT_ECHO_TYPE, DATA_RECORD, RECORD_BUFFER_LENGTH,
❶      * RECORD_SIZE, PICK_DATA_RECORD( 10 ),
       * CHOICE_DATA_RECORD( 3 ), NUM_VAL_DEVICES, NUM_CHOICES,
       * NUM_CHOICE_DEVICES, SIZES( 10 ), ADDRESSES( 10 ),
       * LIST_PROMPT_TYPES( 10 ), PROMPT_RETURN_SIZE, PROMPT_FLAG,
       * INITIAL_CHOICE, INCR, AREA_FLAG, VAL_RECORD_BUFFER_LENGTH,
       * VAL_PROMPT_ECHO_TYPE
        REAL CHOICE_ECHO_AREA( 4 ), VAL_ECHO_AREA( 4 ),
       * ECHO_AREA( 4 ), DUMMY_ARRAY( 4 ), VAL_DATA_RECORD( 2 ),
       * VALUE, UPPER_LIMIT, LOWER_LIMIT, MAX_COORD, DISPLAY_X,
       * DISPLAY_Y

        CHARACTER*80 DEFAULT_STRINGS( 2 )

        DATA DEVICE_NUM / 1 /
            .
            .
            .
C       Initialize the pick device.
C       Make sure that the device supports pick input...
        CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
       * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
       * DUMMY_INTEGER, NUM_PICK_DEVICES, DUMMY_INTEGER )

        IF ( NUM_PICK_DEVICES .EQ. 0 ) THEN
             WRITE(6,*) 'The workstation does not support'
             WRITE(6,*) 'pick input.'
             STOP
        ENDIF

C       Give the data record the size of your data record buffer and
C       inquire about the realized pick values.
        RECORD_BUFFER_LENGTH = 40
        CALL GKS$INQ_PICK_STATE( WS_ID, DEVICE_NUM,
       * GKS$K_VALUE_REALIZED, ERROR_STATUS, INPUT_MODE,
       * ECHO_FLAG, INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
       * PROMPT_ECHO_TYPE, ECHO_AREA, PICK_DATA_RECORD,
       * RECORD_BUFFER_LENGTH, RECORD_SIZE )

C       Make sure that the data record was not truncated...
        IF ( RECORD_SIZE .LT. RECORD_BUFFER_LENGTH ) THEN
             WRITE(6,*) 'The data record was truncated.'
             WRITE(6,*) 'Declare a larger buffer.'
             STOP
        ENDIF
C       Make sure that the pick aperture is not placed on any segment.
❷       INITIAL_STATUS = GKS$K_STATUS_NOPICK

C       Make sure that the device is in request mode (the DEC GKS default).
        CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
       * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
```

```
C     Initialize the device...
      CALL GKS$INIT_PICK( WS_ID, DEVICE_NUM,
     * INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
     * PROMPT_ECHO_TYPE, ECHO_AREA, PICK_DATA_RECORD,
     * RECORD_BUFFER_LENGTH )
           .
           .
           .
      RETURN
      END
           .
           .
           .
```

The following numbers correspond to the numbers in the previous example:

❶ When you declare and define arguments to the GKS$INIT_class functions, you should use caution when naming variables. If you do not, you may pass the wrong variable to an initializing function. For instance, in this code; the data record buffers are named PICK_DATA_RECORD, and CHOICE_DATA_RECORD. There is no confusion as to which buffer you pass to which initializing function.

❷ This code sets the initial input status to GKS$K_STATUS_NOPICK and uses the remaining default pick values to intialize the device. Section 7.4.1 discusses the function of this pick device.

7.3.2 Initializing the Choice and Valuator Devices

The following code is the portion of subroutine INIT_DEVICES that initializes the choice and valuator devices:

```
           .
           .
           .
C     ************************************************************
C     This subroutine initializes all input devices...
      SUBROUTINE INIT_DEVICES( WS_ID, WS_TYPE )
           .
           .
           .
C     First element in the data record is the number of choices.
      EQUIVALENCE( CHOICE_DATA_RECORD( 1 ), NUM_CHOICES )

C     According to the standard, the elements in the data record are
C     the upper and lower limits for all prompt and echo types.
      EQUIVALENCE( VAL_DATA_RECORD( 1 ), LOWER_LIMIT )
      EQUIVALENCE( VAL_DATA_RECORD( 2 ), UPPER_LIMIT )
           .
           .
           .
```

```
C     Initialize the choice and valuator devices...
C     Make sure that the device supports choice input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NUM_CHOICE_DEVICES, DUMMY_INTEGER, DUMMY_INTEGER )

      IF ( NUM_CHOICE_DEVICES .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'choice input.'
           STOP
      ENDIF

C     Make sure that the device supports valuator input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, NUM_VAL_DEVICES,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER )

      IF ( NUM_VAL_DEVICES .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'valuator input.'
           STOP
      ENDIF

C     Obtain the default valuator values...
      VAL_RECORD_BUFFER_LENGTH = 8
      CALL GKS$INQ_VALUATOR_STATE( WS_ID, DEVICE_NUM,
     * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, VALUE,
     * VAL_PROMPT_ECHO_TYPE, VAL_ECHO_AREA, VAL_DATA_RECORD,
     * VAL_RECORD_BUFFER_LENGTH, RECORD_SIZE )

C     Establish the size of the choice record buffer: 12 bytes.
      RECORD_BUFFER_LENGTH = 12

C     The second element in the choice data record for prompt and echo type 1
C     is the pointer to the array containing sizes of each choice character
C     string. You need to initialize the pointer so that the array can be
C     initialized.
      CHOICE_DATA_RECORD( 2 ) = %LOC( SIZES( 1 ) )

C     The third element in the VT241 choice data record is the pointer to the
C     array containing the pointers to the strings to be used. You need
C     to initialize the pointer so that the array can be initialized.
      CHOICE_DATA_RECORD( 3 ) = %LOC( ADDRESSES( 1 ) )
      ADDRESSES( 1 ) = %LOC( DEFAULT_STRINGS( 1 ) )
      ADDRESSES( 2 ) = %LOC( DEFAULT_STRINGS( 2 ) )

C     Initialize NUM_CHOICES to 10.
      NUM_CHOICES = 10

C     Obtain the available prompt and echo types...
      CALL GKS$INQ_DEF_CHOICE_DATA( WS_TYPE, DEVICE_NUM,
     * ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
     * %DESCR( LIST_PROMPT_TYPES), CHOICE_ECHO_AREA,
     * CHOICE_DATA_RECORD, PROMPT_RETURN_SIZE,
     * RECORD_BUFFER_LENGTH, RECORD_SIZE )

C     Obtain the remaining default input values...
      CALL GKS$INQ_CHOICE_STATE( WS_ID, DEVICE_NUM,
     * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, INITIAL_STATUS,
     * INITIAL_CHOICE, PROMPT_ECHO_TYPE, CHOICE_ECHO_AREA,
     * CHOICE_DATA_RECORD, RECORD_BUFFER_LENGTH, RECORD_SIZE )
```

```
          DO 400 INCR = 1, PROMPT_RETURN_SIZE, 1
          IF ( LIST_PROMPT_TYPES( INCR ) .EQ. 3 ) THEN
               PROMPT_FLAG = 1
          ENDIF
    400   CONTINUE

    C     If the workstation does not support prompt and echo type 3...
          IF ( PROMPT_FLAG .EQ. 0 ) THEN
               WRITE(6,*) 'The workstation does not support'
               WRITE(6,*) 'choice prompt and echo type 3.'
               STOP
          ENDIF

❶   C     Make sure that the two echo areas don't conflict...
              .
              .
              .
    C     Initialize the choice device...
          PROMPT_ECHO_TYPE = 3
          INITIAL_CHOICE = 1
          NUM_CHOICES = 8
          INITIAL_STATUS = GKS$K_STATUS_NOCHOICE

    C     Establish sizes of prompt strings...
          SIZES( 1 ) = 5
          SIZES( 2 ) = 5
          SIZES( 3 ) = 4
          SIZES( 4 ) = 8
          SIZES( 5 ) = 4
          SIZES( 6 ) = 7
          SIZES( 7 ) = 5
          SIZES( 8 ) = 5

    C     Establish locations of prompt strings...
❷         ADDRESSES( 1 ) = %LOC( 'Title' )
          ADDRESSES( 2 ) = %LOC( 'Stars' )
          ADDRESSES( 3 ) = %LOC( 'Tree' )
          ADDRESSES( 4 ) = %LOC( 'Sidewalk' )
          ADDRESSES( 5 ) = %LOC( 'Road' )
          ADDRESSES( 6 ) = %LOC( 'Horizon' )
          ADDRESSES( 7 ) = %LOC( 'House' )
          ADDRESSES( 8 ) = %LOC( 'Reset' )

    C     Make sure that the device is in request mode (the DEC GKS default).
          CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
         * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

          CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
         * INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
         * CHOICE_ECHO_AREA, CHOICE_DATA_RECORD,
         * RECORD_BUFFER_LENGTH )

    C     Initialize the valuator device...
❸         VALUE = 1.0
          UPPER_LIMIT = 1.5
          LOWER_LIMIT = 0.5

    C     Make sure that the device is in request mode (the DEC GKS default).
          CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
         * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
```

```
      CALL GKS$INIT_VALUATOR( WS_ID, DEVICE_NUM,
     * VALUE, VAL_PROMPT_ECHO_TYPE, VAL_ECHO_AREA,
     * VAL_DATA_RECORD, VAL_RECORD_BUFFER_LENGTH )

      RETURN
      END
          .
          .
          .
```

The following numbers correspond to the numbers in the previous example:

❶ The code removed from this section establishes echo areas for the choice and valuator devices that do not overlap on the workstation surface. (Section 7.3.3 discusses this code in detail.) Using some of the DEC GKS supported workstations, if you attempt to define echo areas for these types of devices, you generate an error. Using the VAXstations, you do not generate an error, but the user would have to pop and push the overlapping windows (that contain each input device) in order to view an entire device while entering input.

❷ This code establishes eight titles for the items in the choice menu. There is a title for each of the segments in the Starry Night picture (the house, the tree, and so forth), and there is a title for Reset. Section 7.4.4 discusses how the choices in this menu affect the picture.

❸ This code establishes the limits of the valuator device. Section 7.4.4 discusses how these values affect the picture.

7.3.3 Avoiding Overlapping Echo Areas

As mentioned, the user needs to be able to see the entire prompt of certain logical input devices (valuator, choice, and string) in order to easily enter input. If you activate several of these devices concurrently and if you specify overlapping echo areas, you can generate an error message on some DEC GKS supported workstations. Even if you do not generate an error on certain types of workstations (for instance, the VAXstations), you require that the user perform additonal tasks to view the entire prompt of a given input device. (Using a VAXstation, the user has to pop and push the windows containing the input devices to view the entire prompt of an underlying device.)

To avoid this situation, the subroutine INIT_DEVICES calculates the echo areas of the valuator and choice devices so that the areas do not overlap on the workstation surface. The portion of INIT_DEVICES that performs this task is as follows.

```
          .
          .
          .
   C   Make sure that the two echo areas don't conflict...
❶      IF ((( CHOICE_ECHO_AREA( 1 ) .EQ. VAL_ECHO_AREA( 1 )  ) .OR.
      *       ( CHOICE_ECHO_AREA( 2 ) .EQ. VAL_ECHO_AREA( 2 )  )) .OR.
      *    (( CHOICE_ECHO_AREA( 3 ) .EQ. VAL_ECHO_AREA( 3 )  ) .OR.
      *    ( CHOICE_ECHO_AREA( 4 ) .EQ. VAL_ECHO_AREA( 4 )  ))) THEN

❷         CALL GKS$INQ_MAX_DS_SIZE( WS_TYPE, ERROR_STATUS,
      *    DUMMY_INTEGER, DISPLAY_X, DISPLAY_Y, DUMMY_INTEGER,
      *    DUMMY_INTEGER )

❸         MAX_COORD = MAX( DISPLAY_X, DISPLAY_Y )

❹         IF ( DISPLAY_X .NE. DISPLAY_Y ) THEN
❺         IF (( DISPLAY_X / MAX_COORD ) .EQ. 1.0 ) THEN

❻             CHOICE_ECHO_AREA( 1 ) = DISPLAY_X -
      *                         ( DISPLAY_X - DISPLAY_Y )
               CHOICE_ECHO_AREA( 2 ) = DISPLAY_X
               CHOICE_ECHO_AREA( 3 ) = 0.0
               CHOICE_ECHO_AREA( 4 ) = DISPLAY_Y / 2.02
               VAL_ECHO_AREA( 1 ) = DISPLAY_X -
      *                         ( DISPLAY_X - DISPLAY_Y )
               VAL_ECHO_AREA( 2 ) = DISPLAY_X
               VAL_ECHO_AREA( 3 ) = DISPLAY_Y / 1.98
               VAL_ECHO_AREA( 4 ) = DISPLAY_Y

   C           Make sure the pick area does not conflict...
❼             DO WHILE ( ECHO_AREA( 2 ) .GE. VAL_ECHO_AREA( 1 ) )
                  VAL_ECHO_AREA( 1 ) = VAL_ECHO_AREA( 1 ) +
      *                                ( VAL_ECHO_AREA( 1 ) / 100 )
                  CHOICE_ECHO_AREA( 1 ) = CHOICE_ECHO_AREA( 1 ) +
      *                                ( CHOICE_ECHO_AREA( 1 ) / 100 )

               ENDDO
           ELSE

❽             CHOICE_ECHO_AREA( 1 ) = 0.0
               CHOICE_ECHO_AREA( 2 ) = DISPLAY_X / 2.02
               CHOICE_ECHO_AREA( 3 ) = DISPLAY_Y -
      *                         ( DISPLAY_Y - DISPLAY_X )
               CHOICE_ECHO_AREA( 4 ) = DISPLAY_Y
               VAL_ECHO_AREA( 1 ) = DISPLAY_X / 1.98
               VAL_ECHO_AREA( 2 ) = DISPLAY_X
               VAL_ECHO_AREA( 3 ) = DISPLAY_Y -
      *                         ( DISPLAY_Y - DISPLAY_X )
               VAL_ECHO_AREA( 4 ) = DISPLAY_Y

   C           Make sure the pick area does not conflict...
               DO WHILE ( ECHO_AREA( 4 ) .GE. VAL_ECHO_AREA( 3 ) )
                  VAL_ECHO_AREA( 1 ) = VAL_ECHO_AREA( 1 ) +
      *                                ( VAL_ECHO_AREA( 1 ) / 100 )
                  CHOICE_ECHO_AREA( 1 ) = CHOICE_ECHO_AREA( 1 ) +
      *                                ( CHOICE_ECHO_AREA( 1 ) / 100 )

               ENDDO

           ENDIF    !  MAX_COORD equals DISPLAY_X or DISPLAY_Y
❾         ELSE     !  ELSE, if the surface is square...
```

```
              WRITE(6,*) 'The workstation surface is square.'
              WRITE(6,*) 'Any echo area I pick will cover'
              WRITE(6,*) 'part of the picture.  You need to'
              WRITE(6,*) 'alter program transformations.'
              STOP

           ENDIF    !  If the surface is square
        ENDIF   ! If the echo areas conflict.
            .
            .
            .
```

The following numbers correspond to the numbers in the previous example:

❶ This IF clause checks to see if the default echo areas for the choice and valuator devices overlap.

❷ If the areas overlap, then this code obtains the maximum X and Y values of the workstation surface (display size).

❸ This code determines whether the workstation surface is wider than tall (maximum X value larger than maximum Y value), or, if it is taller than wide (Y larger than X).

❹ This IF clause checks to see if the workstation surface is not square.

❺ This IF clause checks to see if the surface is wider than tall (X is larger than Y).

❻ This code determines the area on the workstation surface that is not used to contain the picture, divides it in half, and defines the echo areas of the devices.

Remember that by default, DEC GKS uses the largest square area on the workstation surface, beginning in the lower left corner, to present the picture. If the surface is rectangular, and if the surface is wider than tall, then this leaves extra space on the right side of the surface on which to prompt the user for input.

❼ This code makes sure that the choice and valuator devices do not conflict with the pick echo area.

❽ This code determines the area on the workstation surface that is not used to contain the picture, divides it in half, and defines the echo areas of the devices. Since this portion of the IF statement executes if the workstation surface is taller than wide, then this code uses the extra space on the top of the surface on which to prompt the user for input.

❾ If the workstation surface is square, there is no way to position the choice and valuator devices without covering a portion of the picture. If this happens, this code tells the programmer the problem and makes the suggestion to alter the workstation transformations so both the picture and the input devices can fit on the surface at one time. Since this requires additional programming, this code uses the STOP command to halt program execution.

7.4 Accepting Input

Example 7–1 allows the user to enter input using pick, choice, and valuator devices that are active concurrently. In essence, this example performs the same task as the example in Chapter 6, Requesting Input, allowing the user to scale a segment. However, Example 7–1 also allows the user to do the following:

- Scale more than one segment in the Starry Night picture.
- Scale segments more than once without having to be prompted by the application.
- Use a help screen interactively.
- Stop subsequent scaling of a segment.
- Reset the Starry Night picture, if dissatisfied with prior scaling attempts.

This program accomplishes this task in the following manner:

- By placing the pick device in sample mode.
- By placing the choice and valuator devices in event mode.
- By using an additional choice device in request mode, to ask if the user is finished scaling the picture.

The following sections describe each component of the GET_VALUES subroutine in Example 7–1.

7.4.1 Sampling Pick Input

The application in Example 7–1 activates a pick logical input device, in sample mode, that allows the user to indicate the segment to be scaled. The following code illustrates the portion of the subroutine GET_VALUES that samples the pick device.

```
                .
                .
                .
C     ************************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )
                .
                .
                .
C     Get the input values.
      CALL GET_VALUES( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON, HELP, HELP_BOX, WS_TYPE)
                .
                .
                .
      RETURN
      END

C     ************************************************************
C     This subroutine obtains input values...
      SUBROUTINE GET_VALUES( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON, HELP, HELP_BOX, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, HELP, HELP_BOX, DEVICE_NUM, FINISHED_FLAG,
     * PICKED_SEGMENT, PICK_INPUT_STATUS, INPUT_STATUS,
     * PICK_ID, CLASS, ERROR_STATUS, INPUT_CHOICE, RESET,
     * MORE_EVENTS_FLAG, DUMMY_INTEGER, NEW_FRAME_FLAG, WS_TYPE,
     * CURRENT_SEGMENT, INITIAL_STATUS, PROMPT_ECHO_TYPE,
     * PICK_DATA_RECORD( 10 ), RECORD_BUFFER_LENGTH, REPEAT_FLAG,
     * HELP_FLAG, VALUE_FLAG, LOCKED_SEGMENT, INCR
      REAL IDENTITY( 6 ), TITLE_XFORM_MATRIX( 6 ),
     * STARS_XFORM_MATRIX( 6 ), HOUSE_XFORM_MATRIX( 6 ),
     * TREE_XFORM_MATRIX( 6 ), SIDE_XFORM_MATRIX( 6 ),
     * ROAD_XFORM_MATRIX( 6 ), HORIZON_XFORM_MATRIX( 6 ), VALUE,
     * FIXED_X, FIXED_Y, CURRENT_VALUE, ECHO_AREA( 4 )

      DATA DEVICE_NUM / 1 /, FINISHED_FLAG / 0 /, RESET / 8 /

  C   Place the devices in the proper input mode.
❶     CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_SAMPLE, GKS$K_ECHO )
      CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )
      CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )
```

```
   C     Create an identity matrix and initial transformation matrixes.
❷        CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, IDENTITY )
         CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, TITLE_XFORM_MATRIX )
         CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, STARS_XFORM_MATRIX )
         CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
         CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
         CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, SIDE_XFORM_MATRIX )
         CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, ROAD_XFORM_MATRIX )
         CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
        * 1.0, 1.0, GKS$K_COORDINATES_NDC, HORIZON_XFORM_MATRIX )

❸        HELP_FLAG = 0
         REPEAT_FLAG = 0
         VALUE = 1.0
         VALUE_FLAG = 0
         CURRENT_VALUE = 1.0
         LOCKED_SEGMENT = 0
         DO WHILE ( FINISHED_FLAG .NE. 1 )

❹        CALL GKS$SAMPLE_PICK( WS_ID, DEVICE_NUM,
        * PICK_INPUT_STATUS, PICKED_SEGMENT, PICK_ID )

❺        IF (( LOCKED_SEGMENT .NE. 0 ) .AND.
        *    ( PICKED_SEGMENT .NE. LOCKED_SEGMENT )) THEN
               LOCKED_SEGMENT = 0
         ENDIF

❻        IF (( PICKED_SEGMENT .NE. HELP_BOX ) .AND.
        *    ( HELP_FLAG .EQ. 1 )) THEN
              HELP_FLAG = 0
         ENDIF

❼        IF ( HELP_FLAG .EQ. 1 ) THEN
               PICKED_SEGMENT = TITLE
               PICK_INPUT_STATUS = GKS$K_STATUS_NOPICK
         ENDIF

❽        IF (( PICKED_SEGMENT .EQ. HELP_BOX ) .AND.
        *    ( HELP_FLAG .NE. 1 )) THEN

              CALL GET_HELP( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
        *                    HOUSE, HORIZON, HELP, HELP_BOX,
        *                    FINISHED_FLAG, WS_TYPE )

❾             HELP_FLAG = 1

❿        ELSE
             .
             .
             .
         ENDIF  ! If segment equals help.
```

```
  C     Set the current segment, current value, and entered value flag...
⓫       CURRENT_SEGMENT = PICKED_SEGMENT
        CURRENT_VALUE = VALUE
        VALUE_FLAG = 0

        ENDDO

        RETURN
        END
            .
            .
            .
```

The following numbers correspond to the numbers in the previous example:

❶ This code sets the logical input devices to the proper input operating modes. The pick device is in sample mode; the valuator and choice devices are in event mode.

❷ This code establishes the initial transformation matrixes for each of the segments and creates a single identity matrix. The application can use the identity matrix to return any segment to its original size, shape, and relative position within the picture.

❸ This application uses a series of flags as it loops, continually sampling the currently picked segment. This code initializes these flags, as follows:

- The HELP_FLAG signifies whether the user had just used the help screen.
- The REPEAT_FLAG signifies whether you would be repeating a scaling value for the same segment without having moved the pick prompt.
- The VALUE_FLAG signifies whether there has been a change in the current real value used for scaling.
- The LOCKED_SEGMENT signifies which segment is currently set to GKS$K_UNDETECTABLE (cannot be picked, stopping further scaling).
- The FINISHED_FLAG signifies whether the user wishes to stop entering any further input.

❹ This code samples the current measure of the pick device.

❺ If the user has moved the pick prompt to a new segment, and if the user had previously specified a segment to lock, then this code resets LOCKED_SEGMENT to zero (0). This application uses LOCKED_SEGMENT elsewhere in the program to set the segment's detectablity to GKS$K_UNDETECTABLE, stopping further scaling of that segment.

❻ If the user had requested the help screen, and if the user has moved the pick prompt off of the HELP_BOX segment, then this code resets the help flag. When the user requests the help screen, this flag is set to the value 1. The value is not reset until the user moves the prompt off of the HELP_BOX segment. This code avoids an infinite loop when the user requests help.

7 This code temporarily assigns a value to PICKED_SEGMENT as a signal that the user already viewed the help screen. In this way, HELP_FLAG can be reset to zero (0) the next time through the loop.

8 This code determines whether the currently picked segment is the help screen segment (HELP_BOX). If it is, then this program calls the subroutine GET_HELP. Section 7.4.2 describes this subroutine in detail.

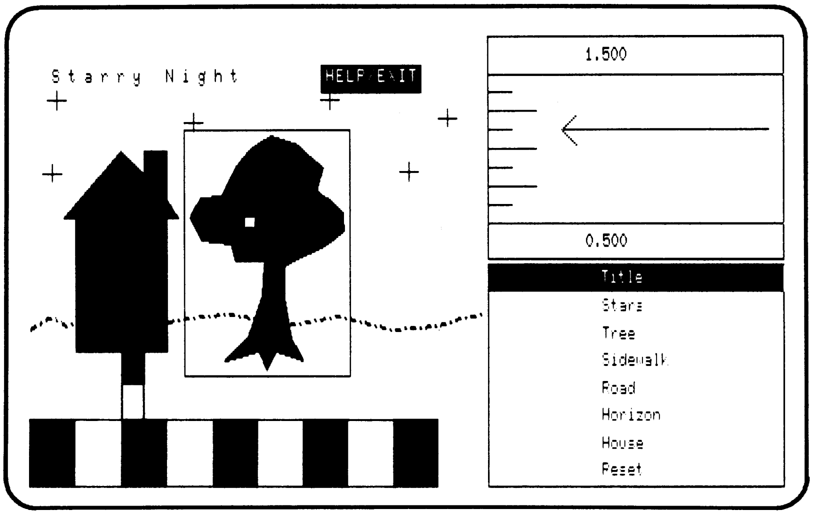
9 If the user requires the help screen, set HELP_FLAG to 1.

10 If the user does not require help, then, in this ELSE clause, this application checks the event queue to see if the user specified new valuator or choice input. Section 7.4.4 describes this portion of the code in detail.

11 This code resets flags, and establishes the current choice and valuator values. The application compares the current input with new input to determine changes in subsequently specified input values.

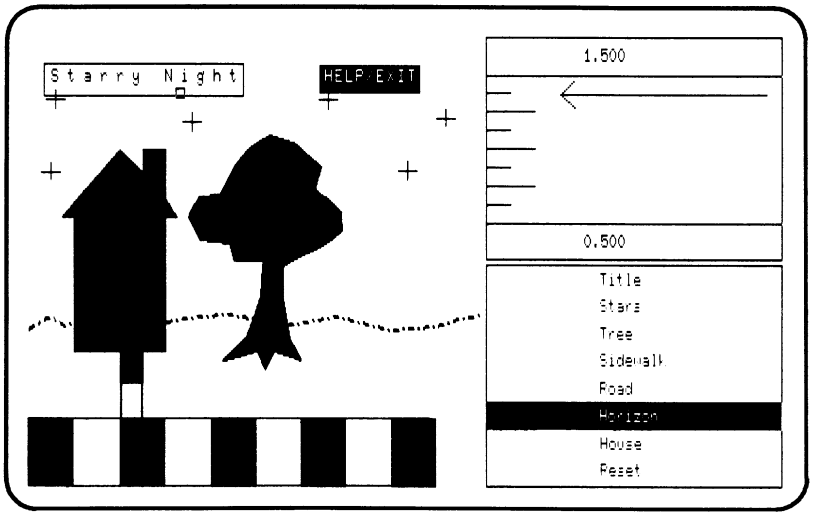
Figures 7–14 through 7–16 illustrate the effect of sampling different segments. In Figures 7–14 and 7–15, using the VT241, the measures of the choice and valuator devices change along with the pick device (pressing the arrow keys alters the measures of all of these active devices). After pressing the PF1 key until only the pick prompt is active, Figure 7–16 illustrates the difference in appearance; notice that the valuator arrow is gone and that none of the choices are highlighted.

Figure 7–14: Picking Segments in Sample Mode—VT241



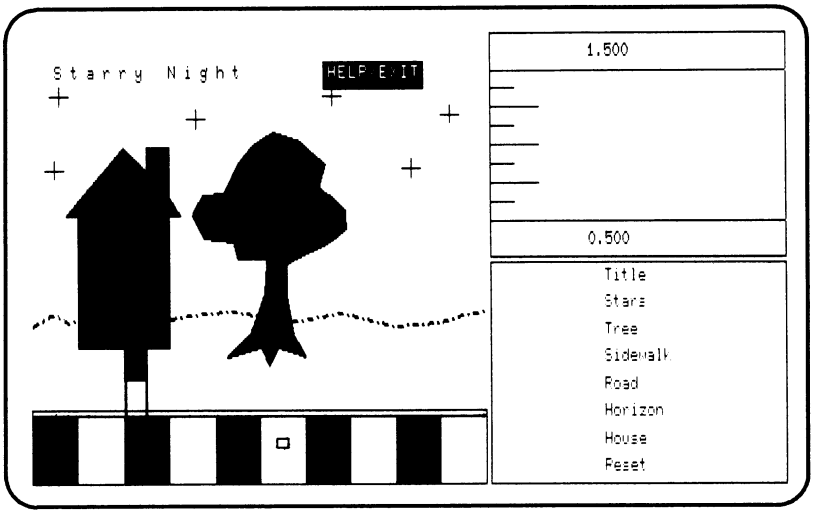
ZK-5953-HC

Figure 7–15: Picking Segments in Sample Mode—VT241



ZK-5954-HC

Figure 7–16: Picking Segments in Sample Mode—VT241



ZK-5955-HC

7.4.2 Using the Help Screen

When the user picks the HELP_BOX segment (the rectangle on the surface labeled HELP/EXIT), this application places a help screen on the surface of the workstation. This section describes the GET_HELP subroutine in Example 7–1.

```
         .
         .
         .
C       ************************************************************
C       This subroutine obtains input values...
        SUBROUTINE GET_VALUES( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
      * HOUSE, HORIZON, HELP, HELP_BOX, WS_TYPE )
         .
         .
         .
        IF (( PICKED_SEGMENT .EQ. HELP_BOX ) .AND.
      *     ( HELP_FLAG .NE. 1 )) THEN
```

```
          CALL GET_HELP( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     *                   HOUSE, HORIZON, HELP, HELP_BOX,
     *                   FINISHED_FLAG, WS_TYPE )
            .
            .
            .
C    ************************************************************
C     This subroutine makes the help screen visible...
      SUBROUTINE GET_HELP( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON, HELP, HELP_BOX, FINISHED_FLAG, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, HELP, HELP_BOX, DEVICE_NUM, FINISHED_FLAG,
     * NEW_FRAME_FLAG, DUMMY_INTEGER, ERROR_STATUS, INPUT_MODE,
     * ECHO_FLAG, INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
     * CHOICE_DATA_RECORD( 3 ), RECORD_BUFFER_LENGTH, RECORD_SIZE,
     * TEMP_DATA_RECORD( 3 ), TEMP_RBL, TEMP_RECORD_SIZE,
     * LIST_PROMPT_TYPES, PROMPT_RETURN_SIZE, TEMP_INITIAL_STATUS,
     * TEMP_INITIAL_CHOICE, NUM_CHOICES, SIZES( 10 ),
     * ADDRESSES( 10 ), TEMP_SIZES( 2 ), TEMP_ADDRESSES( 2 ),
     * INPUT_STATUS, CHOICE, WS_TYPE, CONTINUE

      CHARACTER*80 CURRENT_STRINGS( 10 ),
     * TEMP_CURRENT_STRINGS( 2 )

❶     REAL ECHO_AREA( 4 ), TEMP_ECHO_AREA( 4 )

      DATA DEVICE_NUM / 1 /, CONTINUE / 1 /

      EQUIVALENCE( CHOICE_DATA_RECORD( 1 ), NUM_CHOICES )

C     Reset visibility of all segments and deactivate the input prompts.
❷     CALL GKS$SET_SEG_VISIBILITY( TITLE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( STARS, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( TREE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( SIDE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( ROAD, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HOUSE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HELP_BOX, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HELP, GKS$K_VISIBLE )
❸     CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
      CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
      CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
❹
            .
            .
            .
C     Check to see whether the picture on the screen is out of date.
❺     CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NEW_FRAME_FLAG )
```

```
  C     Release deferred output. Regenerate if necessary.
        IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
        ENDIF

  C     Ask the user to quit or to continue...
❻       CALL GKS$REQUEST_CHOICE( WS_ID, DEVICE_NUM, INPUT_STATUS,
       * CHOICE )

  C     Reset the visibility of the segments.
❼       CALL GKS$SET_SEG_VISIBILITY( TITLE, GKS$K_VISIBLE )
        CALL GKS$SET_SEG_VISIBILITY( STARS, GKS$K_VISIBLE )
        CALL GKS$SET_SEG_VISIBILITY( TREE, GKS$K_VISIBLE )
        CALL GKS$SET_SEG_VISIBILITY( SIDE, GKS$K_VISIBLE )
        CALL GKS$SET_SEG_VISIBILITY( ROAD, GKS$K_VISIBLE )
        CALL GKS$SET_SEG_VISIBILITY( HOUSE, GKS$K_VISIBLE )
        CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_VISIBLE )
        IF ( CHOICE .EQ. CONTINUE ) THEN
        CALL GKS$SET_SEG_VISIBILITY( HELP_BOX, GKS$K_VISIBLE )
        ENDIF
        CALL GKS$SET_SEG_VISIBILITY( HELP, GKS$K_INVISIBLE )

  C     Check to see whether the picture on the screen is out of date.
        CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
       * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
       * NEW_FRAME_FLAG )

  C     Release deferred output. Regenerate if necessary.
        IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
             CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
        ENDIF

  C     Set values depending on the user's choice.
❽       IF ( CHOICE .EQ. CONTINUE ) THEN
             FINISHED_FLAG = 0
  C     Reset the choice device with its previous values...
           .
           .
           .
  C     Reactivate the input devices...
             CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
       *       GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )
             CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
       *       GKS$K_INPUT_MODE_SAMPLE, GKS$K_ECHO )
             CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
       *       GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )

        ELSE
             FINISHED_FLAG = 1
             CALL GKS$DELETE_SEG( HELP_BOX )
        ENDIF

        RETURN
        END
```

The following numbers correspond to the numbers in the previous example:

❶ This subroutine makes Starry Night invisible, deactivates the logical input devices, and displays a help screen. When the help screen is visible to the user, this subroutine uses another choice device, in request mode, to ask the user to either continue scaling or exit from the program. Since this subroutine activates a second choice device, it defines duplicate variables (such as ECHO_AREA and TEMP_ECHO_AREA) to store the current choice values and to present the new Continue/Exit menu.

❷ This code changes the visibility of the segments so that only the help screen is visible.

❸ This code places all of the current logical input devices in request mode so that the device handler removes their prompts from the workstation surface. At this point, you can call one of the GKS$INIT_class functions to reinitialize the device.

❹ The missing code that belongs here stores the current choice values (House/Tree . . .) and places the Continue/Exit menu on the workstation surface. Section 7.4.3 discusses this code in detail.

❺ This code updates the surface of the workstation if necessary. At this point in the program the new visibility attributes of the segments take effect; only the help screen is visible on the workstation surface.

❻ This call to GKS$REQUEST_CHOICE places the Continue/Exit choice menu on the workstation surface. The application pauses until the user makes a decision.

❼ This code resets the visibility attributes of the segments so that Starry Night is once again visible on the workstation surface.

❽ This code reestablishes the old choice values (House/Tree . . .) by reinitializing the choice device, reactivates the remaining logical input devices, and sets FINISHED_FLAG to the appropriate value.

Figure 7–17 illustrates the workstation surface as the user moves the prompt to the HELP_BOX segment. As soon as the pick aperture touches the box, the help screen becomes visible. Figure 7–18 illustrates the help screen and the Continue/Exit menu.

Figure 7–17: Picking the HELP/EXIT Segment—VT241

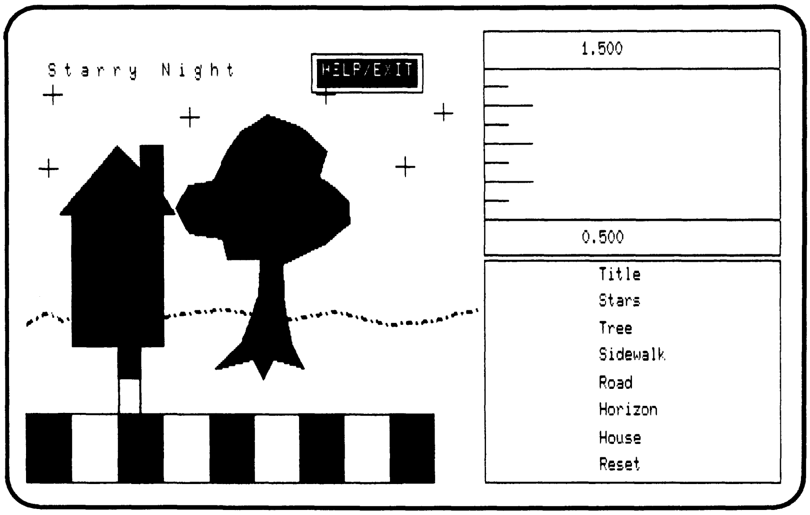
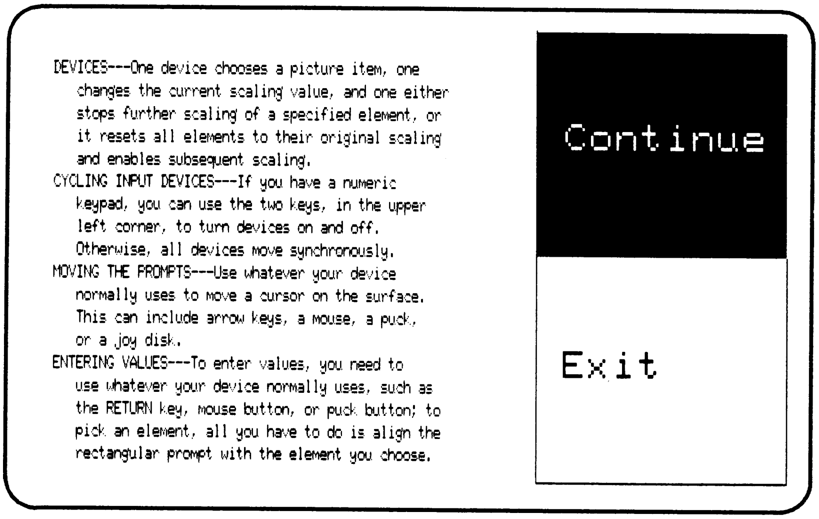


Figure 7–18: Displaying a Help Screen—VT241


DEVICES---One device chooses a picture item, one
   changes the current scaling value, and one either
   stops further scaling of a specified element, or
   it resets all elements to their original scaling
   and enables subsequent scaling.
CYCLING INPUT DEVICES---If you have a numeric
   keypad, you can use the two keys, in the upper
   left corner, to turn devices on and off.
   Otherwise, all devices move synchronously.
MOVING THE PROMPTS---Use whatever your device
   normally uses to move a cursor on the surface.
   This can include arrow keys, a mouse, a puck,
   or a joy disk.
ENTERING VALUES---To enter values, you need to
   use whatever your device normally uses, such as
   the RETURN key, mouse button, or puck button; to
   pick an element, all you have to do is align the
   rectangular prompt with the element you choose.

ZK-5957-HC

7.4.3 Storing Current Input Values

The application in Example 7–1 requires the use of a choice input device to obtain input from the user that affects subsequent execution of the program. Using the Continue/Exit choice menu, the application cannot continue to process until it knows whether the user wants to continue scaling or to exit from the program.

The use of such a menu presents the problem of using two logical input devices of the same class in the same application. If your application requires this, you must choose one of the solutions to the problem.

- Use two distinct logical input device numbers for each device.
- Use the same device number, and reinitialize the device each time you need a change.

If you use two distinct logical input device numbers, you may have two devices that look the same to the user, but that measure and trigger differently. For instance, using the VT241, choice device number 1 requires that the user press the arrow keys to change the measure and the RETURN key to trigger. Choice device number 2 requires that the user press either the arrow keys or one of the numeric keypad keys to both change the measure and trigger the device (the user does not need to press RETURN). Consequently, if you use two distinct logical input device numbers for devices of the same class, you need to be sure that the user knows the differences in the ways in which the two devices are measured and triggered.

The application in Example 7–1 uses a single choice device (numbered 1), and reinitializes the device when it needs a different choice menu. For this purpose, DEC GKS provides you with the following input inquiry functions:

- Ones that write the default input values to its arguments. (GKS$INQ_DEF_CHOICE_DATA, GKS$INQ_DEF_LOCATOR_DATA, GKS$INQ_DEF_PICK_DATA, GKS$INQ_DEF_STRING_DATA, GKS$INQ_DEF_STROKE_DATA, and GKS$INQ_DEF_VALUATOR_DATA)
- Ones that write the current input values to its arguments. (GKSINQ_CHOICE_STATE, GKSINQ_LOCATOR_STATE, GKSINQ_PICK_STATE, GKSINQ_STRING_STATE, GKS$INQ_STROKE_STATE, and GKS$INQ_VALUATOR_STATE)

When reinitializing a single logical input device, you accomplish this by performing the following tasks:

- By storing the current values of the device using one of the GKS$INQ_class_STATE functions.
- By obtaining the default values of the device using one of the GKS$INQ_DEF_class_DATA functions.
- By reinitializing the device using different values.

Once you are finished using one device, you can reinitialize the device using the values that you had stored in buffers, thus reestablishing the values of the old device.

The following code from Example 7-1 illustrates this process:

```
            .
            .
            .
  C      ************************************************************
  C      This subroutine makes the help screen visible...
         SUBROUTINE GET_HELP( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
        * HOUSE, HORIZON, HELP, HELP_BOX, FINISHED_FLAG, WS_TYPE )
            .
            .
            .
❶        CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
        * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
         CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
        * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
         CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
        * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
            .
            .
            .
  C      Store the current choice values...
❷        CHOICE_DATA_RECORD( 1 ) = 10
         CHOICE_DATA_RECORD( 2 ) = %LOC( SIZES( 1 ) )
         SIZES( 1 ) = 80
         SIZES( 2 ) = 80
         SIZES( 3 ) = 80
         SIZES( 4 ) = 80
         SIZES( 5 ) = 80
         SIZES( 6 ) = 80
         SIZES( 7 ) = 80
         SIZES( 8 ) = 80
         SIZES( 9 ) = 80
         SIZES( 10 ) = 80
         CHOICE_DATA_RECORD( 3 ) = %LOC( ADDRESSES( 1 ) )
         ADDRESSES( 1 ) = %LOC( CURRENT_STRINGS( 1 ) )
         ADDRESSES( 2 ) = %LOC( CURRENT_STRINGS( 2 ) )
         ADDRESSES( 3 ) = %LOC( CURRENT_STRINGS( 3 ) )
         ADDRESSES( 4 ) = %LOC( CURRENT_STRINGS( 4 ) )
         ADDRESSES( 5 ) = %LOC( CURRENT_STRINGS( 5 ) )
         ADDRESSES( 6 ) = %LOC( CURRENT_STRINGS( 6 ) )
         ADDRESSES( 7 ) = %LOC( CURRENT_STRINGS( 7 ) )
         ADDRESSES( 8 ) = %LOC( CURRENT_STRINGS( 8 ) )
         ADDRESSES( 9 ) = %LOC( CURRENT_STRINGS( 9 ) )
         ADDRESSES( 10 ) = %LOC( CURRENT_STRINGS( 10 ) )
  C      Save the current choice input initialization values...
❸        RECORD_BUFFER_LENGTH = 12
         CALL GKS$INQ_CHOICE_STATE( WS_ID, DEVICE_NUM,
        * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, INITIAL_STATUS,
        * INITIAL_CHOICE, PROMPT_ECHO_TYPE, ECHO_AREA,
        * CHOICE_DATA_RECORD, RECORD_BUFFER_LENGTH, RECORD_SIZE )

  C      Obtain the default values and put them in temporary buffers.
❹        TEMP_DATA_RECORD( 1 ) = 2
         TEMP_DATA_RECORD( 2 ) = %LOC( TEMP_SIZES( 1 ) )
         TEMP_DATA_RECORD( 3 ) = %LOC( TEMP_ADDRESSES( 1 ) )
         TEMP_ADDRESSES( 1 ) = %LOC( TEMP_CURRENT_STRINGS( 1 ) )
         TEMP_ADDRESSES( 2 ) = %LOC( TEMP_CURRENT_STRINGS( 2 ) )
  C     Inquire the default values...
❺       TEMP_RBL = 12
        CALL GKS$INQ_DEF_CHOICE_DATA( WS_TYPE, DEVICE_NUM,
       * ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
       * %DESCR( LIST_PROMPT_TYPES), TEMP_ECHO_AREA,
       * TEMP_DATA_RECORD, PROMPT_RETURN_SIZE, TEMP_RBL,
       * TEMP_RECORD_SIZE )

  C     Set temporary values...
        TEMP_INITIAL_CHOICE = 1
        TEMP_INITIAL_STATUS = GKS$K_STATUS_OK
        TEMP_SIZES( 1 ) = 8
        TEMP_SIZES( 2 ) = 4
❻       TEMP_ADDRESSES( 1 ) = %LOC( 'Continue' )
        TEMP_ADDRESSES( 2 ) = %LOC( 'Exit' )

  C     Reinitialize the choice device...
        CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
       * TEMP_INITIAL_STATUS, TEMP_INITIAL_CHOICE,
       * PROMPT_ECHO_TYPE, TEMP_ECHO_AREA, TEMP_DATA_RECORD,
       * TEMP_RBL )
            .
            .
            .
  C     Ask the user to quit or to continue...
        CALL GKS$REQUEST_CHOICE( WS_ID, DEVICE_NUM, INPUT_STATUS,
       * CHOICE )
            .
            .
            .
  C     Set values depending on the user's choice.
        IF ( CHOICE .EQ. CONTINUE ) THEN
             FINISHED_FLAG = 0
  C     Reset the choice device with its previous values...

❼            CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
       *       INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
       *       ECHO_AREA, CHOICE_DATA_RECORD, RECORD_BUFFER_LENGTH )

  C     Reactivate the input devices...
             CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
       *       GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )
             CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
       *       GKS$K_INPUT_MODE_SAMPLE, GKS$K_ECHO )
             CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
       *       GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )

            .
            .
            .
        RETURN
        END
```

The following numbers correspond to the numbers in the previous example:

❶ Deactivate the devices by removing their prompts from the workstation surface. At this point, you can reinitialize the devices by calling one of the GKS$INIT_class functions.

❷ This code establishes the sizes and locations of the buffers that will hold the current choice data. This application assumes that it does not know how many choices are in the current menu, so it establishes ten buffers. Also assuming that the sizes of the choice labels are unknown, the application defines string buffers that can hold labels up to 80 characters in length.

❸ This call to GKS$INQ_CHOICE_STATE writes the current choice values (whose labels are House, Tree, and so forth), exactly as the user left it, to its arguments. The application will use these values later to reinitialize the choice device when it needs the previously used menu again.

❹ This code establishes temporary buffers for use only within the GET_HELP subroutine. These temporary buffers will hold the values used for the Continue/Exit menu.

❺ The call to GKS$INQ_DEF_CHOICE_DATA obtains the default values for choice device number 1.

❻ The new labels are Continue and Exit.

❼ If the user chooses to continue, this code reinitializes the choice device with the stored values of the House/Tree menu.

7.4.4 Checking for Generated Events

The remaining portion of the GET_VALUES subroutine in Example 7–1 checks the event queue for input. The effects of this input depend on the types and order of the input values. The following portion of GET_VALUES shows how to obtain input from the event queue:

```
              .
              .
              .
  C     ************************************************************
  C     This subroutine obtains input values...
        SUBROUTINE GET_VALUES( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
       * HOUSE, HORIZON, HELP, HELP_BOX, WS_TYPE )
              .
              .
              .
        IF (( PICKED_SEGMENT .EQ. HELP_BOX ) .AND.
       *    ( HELP_FLAG .NE. 1 )) THEN
              .
              .
              .
        ELSE

  C     Check the event queue.
❶       CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )

  C     Check for queue overflow.
❷       CALL GKS$INQ_INPUT_QUEUE_OVERFLOW( ERROR_STATUS, WS_ID,
       * CLASS, DEVICE_NUM )
  C      If the queue has overflowed...
❸        IF ( ERROR_STATUS .EQ. 0 ) THEN
         CALL GKS$FLUSH_DEVICE_EVENTS( WS_ID,
        * GKS$K_INPUT_CLASS_VALUATOR, DEVICE_NUM )
         CALL GKS$FLUSH_DEVICE_EVENTS( WS_ID,
        * GKS$K_INPUT_CLASS_CHOICE, DEVICE_NUM )
         CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )
         ENDIF

  100    CONTINUE
❹        IF ( CLASS .EQ. GKS$K_INPUT_CLASS_CHOICE ) THEN
              CALL GKS$GET_CHOICE( INPUT_STATUS, INPUT_CHOICE )

              IF ( INPUT_STATUS .NE. GKS$K_STATUS_NOCHOICE ) THEN
              IF ( INPUT_CHOICE .NE. RESET ) THEN
                   CALL GKS$SET_SEG_DETECTABILITY( INPUT_CHOICE,
        *                                     GKS$K_UNDETECTABLE )
  C      Don't let the user scale the segment any more.
                   LOCKED_SEGMENT = PICKED_SEGMENT
              ELSE
❺                  CALL GKS$SET_SEG_XFORM( TITLE, IDENTITY )
                   DO 200 INCR = 1, 6, 1
                   TITLE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
  200              CONTINUE
                   CALL GKS$SET_SEG_XFORM( STARS, IDENTITY )
                   DO 300 INCR = 1, 6, 1
                   STARS_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
  300              CONTINUE
                   CALL GKS$SET_SEG_XFORM( HOUSE, IDENTITY )
                   DO 400 INCR = 1, 6, 1
                   HOUSE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
  400              CONTINUE
                   CALL GKS$SET_SEG_XFORM( TREE, IDENTITY )
                   DO 500 INCR = 1, 6, 1
                   TREE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
  500              CONTINUE
                   CALL GKS$SET_SEG_XFORM( SIDE, IDENTITY )
                   DO 600 INCR = 1, 6, 1
                   SIDE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
  600              CONTINUE
                   CALL GKS$SET_SEG_XFORM( ROAD, IDENTITY )
                   DO 700 INCR = 1, 6, 1
                   ROAD_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
  700              CONTINUE
                   CALL GKS$SET_SEG_XFORM( HORIZON, IDENTITY )
                   DO 800 INCR = 1, 6, 1
                   HORIZON_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
  800              CONTINUE
```

```
                 CALL GKS$SET_SEG_DETECTABILITY( TITLE,
     *                                           GKS$K_DETECTABLE )
                 CALL GKS$SET_SEG_DETECTABILITY( STARS,
     *                                           GKS$K_DETECTABLE )
                 CALL GKS$SET_SEG_DETECTABILITY( HOUSE,
     *                                           GKS$K_DETECTABLE )
                 CALL GKS$SET_SEG_DETECTABILITY( TREE,
     *                                           GKS$K_DETECTABLE )
                 CALL GKS$SET_SEG_DETECTABILITY( SIDE,
     *                                           GKS$K_DETECTABLE )
                 CALL GKS$SET_SEG_DETECTABILITY( ROAD,
     *                                           GKS$K_DETECTABLE )
                 CALL GKS$SET_SEG_DETECTABILITY( HORIZON,
     *                                           GKS$K_DETECTABLE )
C     Check to see whether the picture on the screen is out of date.
❻       CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     *      DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     *      NEW_FRAME_FLAG )

C     Release deferred output. Regenerate if necessary.
         IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
            CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
         ENDIF

         ENDIF  !  If choice does not equal reset.
         ENDIF  !  If input status does not equal no choice.
      ENDIF       !  If input class equals choice.

❼     IF ( CLASS .EQ. GKS$K_INPUT_CLASS_VALUATOR ) THEN
           CALL GKS$GET_VALUATOR( VALUE )
           VALUE_FLAG = 1
      ENDIF

C     Check for simultaneously entered events...
❽     CALL GKS$INQ_MORE_SIMUL_EVENTS( ERROR_STATUS,
     * MORE_EVENTS_FLAG )

C     If there are more simultaneous events, take them from the queue...
      IF ( MORE_EVENTS_FLAG .EQ. GKS$K_MORE_EVENTS) THEN
         CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )
         GOTO 100
      ENDIF

❾     IF ( PICK_INPUT_STATUS .EQ. GKS$K_STATUS_NOPICK ) THEN
           REPEAT_FLAG = 1
      ELSEIF ( VALUE .EQ. 1.0 ) THEN
           REPEAT_FLAG = 1
      ELSEIF (( VALUE_FLAG .EQ. 0 ) .AND.
     *        ( PICKED_SEGMENT .EQ. CURRENT_SEGMENT )) THEN
           REPEAT_FLAG = 1
      ELSEIF ( PICKED_SEGMENT .EQ. LOCKED_SEGMENT ) THEN
           REPEAT_FLAG = 1
      ENDIF
```

```
C     Establish fixed points for segments depending on picked segment.
      IF ( REPEAT_FLAG .EQ. 0 ) THEN
⓾     IF ( PICKED_SEGMENT .EQ. TITLE ) THEN
         FIXED_X = 0.3
         FIXED_Y = 0.925
         CALL GKS$ACCUM_XFORM_MATRIX( TITLE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, TITLE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( TITLE, TITLE_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. STARS ) THEN
         FIXED_X = 0.5
         FIXED_Y = 0.8
         CALL GKS$ACCUM_XFORM_MATRIX( STARS_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, STARS_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( STARS, STARS_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. TREE ) THEN
         FIXED_X = 0.52
         FIXED_Y = 0.51
         CALL GKS$ACCUM_XFORM_MATRIX( TREE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. SIDE ) THEN
         FIXED_X = 0.225
         FIXED_Y = 0.22
         CALL GKS$ACCUM_XFORM_MATRIX( SIDE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, SIDE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( SIDE, SIDE_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. ROAD ) THEN
         FIXED_X = 0.5
         FIXED_Y = 0.075
         CALL GKS$ACCUM_XFORM_MATRIX( ROAD_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, ROAD_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( ROAD, ROAD_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ.  HORIZON ) THEN
         FIXED_X = 0.1
         FIXED_Y = 0.35
         CALL GKS$ACCUM_XFORM_MATRIX( HORIZON_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, HORIZON_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( HORIZON, HORIZON_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. HOUSE ) THEN
         FIXED_X = 0.2
         FIXED_Y = 0.5
         CALL GKS$ACCUM_XFORM_MATRIX( HOUSE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( HOUSE, HOUSE_XFORM_MATRIX )
      ENDIF   !  Scaling.

C     Check to see whether the picture on the screen is out of date.
      CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NEW_FRAME_FLAG )
```

```
C     Release deferred output. Regenerate if necessary.
      IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

      ENDIF   !  If we aren't repeating a transformation...

C     Reset the repeating transformations flag.
      REPEAT_FLAG = 0

      ENDIF  ! If segment equals help.

C     Set the current segment, current value, and entered value flag...
      CURRENT_SEGMENT = PICKED_SEGMENT
      CURRENT_VALUE = VALUE
      VALUE_FLAG = 0

      ENDDO

      RETURN
      END
```

The following numbers correspond to the numbers in the previous example:

❶ The call to GKS$AWAIT_EVENT checks the event input queue immediately (0.0 seconds). If there was at least one event on the queue, the call removes the oldest report, places it in the *current event report* in the DEC GKS state list, and writes either GKS$K_INPUT_CLASS_VALUATOR or GKS$K_INPUT_CLASS_CHOICE to its CLASS argument (depending on which device generated the event). If there were no reports on the queue at the time of the call, GKS$AWAIT_EVENT writes GKS$K_INPUT_CLASS_NONE to its CLASS argument.

❷ The call to GKS$INQ_INPUT_QUEUE_OVERFLOW returns zero (0) to its ERROR_STATUS argument if the queue has overflowed and if information about the overflow is available.

❸ If the overflow occurred, this code flushes all reports generated by choice and valuator devices (the only possible reports that the user can generate using this application) and calls GKS$AWAIT_EVENT to reset the CLASS argument (which will probably be GKS$K_INPUT_CLASS_NONE).

❹ If GKS$AWAIT_EVENT wrote GKS$K_INPUT_CLASS_CHOICE to its CLASS argument, then this code calls GKS$GET_CHOICE to obtain the input. If the choice is anything other than Reset, then this code locks the segment by setting its attribute to be GKS$K_UNDETECTABLE. This action disables further scaling of the segment.

❺ If the user chose Reset, then this code sets all of the segment transformations equal to the identity transformation (reestablishing the segment's original size), and sets all of the segment's attributes to GKS$K_UNDETECTABLE.

❻ This code updates the workstation surface if necessary.

7 If the CLASS argument to GKS$AWAIT_EVENT is GKS$K_INPUT_CLASS_VALUATOR, then this code calls GKS$GET_VALUATOR to obtain the value and then resets the flag that specifies that there is a newly specified value.

8 If there exists a simultaneously generated report, this code calls GKS$AWAIT_EVENT to remove that report from the queue, and then redirects control to line 100. At line 100, the application processes the input in a manner dependent on whether the current report was generated by a choice or valuator device.

9 This code checks current values to determine whether there is a need to scale a segment. If the user has not specified a new segment or a new scaling value, then the application does not attempt to scale. This avoids resetting the same scaling value for the same segment twice in a row. (If there is no need to adjust scaling, the application sets REPEAT_FLAG to the value 1.)

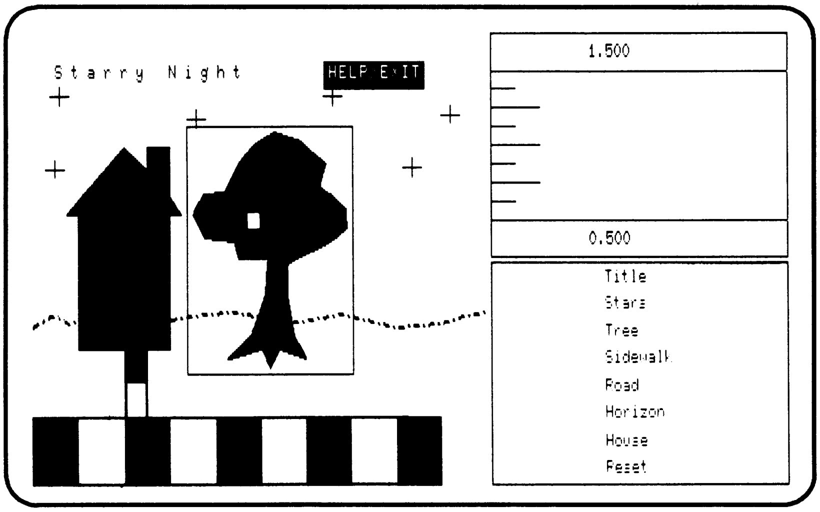
10 This code scales the appropriate segment.

Figures 7–19 to 7–28 illustrate the effects of input possibly generated by the user. The following list describes each figure separately:

- Figure 7–19 shows the workstation surface after the user cycles to the pick device and then picks the tree.
- Figure 7–20 shows the workstation surface after the user cycles to the valuator device and alters its measure.
- Figure 7–21 shows the workstation surface after the user triggers the valuator device. Figure 7–22 shows the surface when the user triggers the valuator device a second time.
- Figure 7–23 shows the workstation surface after the user cycles to the choice device, triggers the device (entering the choice Tree), and then cycles to the pick device. Notice how the extent rectangle of the tree is not visible. The application stops further scaling of this segment; its attribute is GKS$K_UNDETECTABLE.
- Figure 7–24 shows the workstation surface after the user picks the road.
- Figure 7–25 shows the workstation surface after the user cycles to the choice device and triggers on the "Reset" choice. All segments regain their original shape and their attributes are reset to GKS$K_DETECTABLE.
- Figure 7–26 shows the workstation surface after the user cycles to the pick device and picks the house.

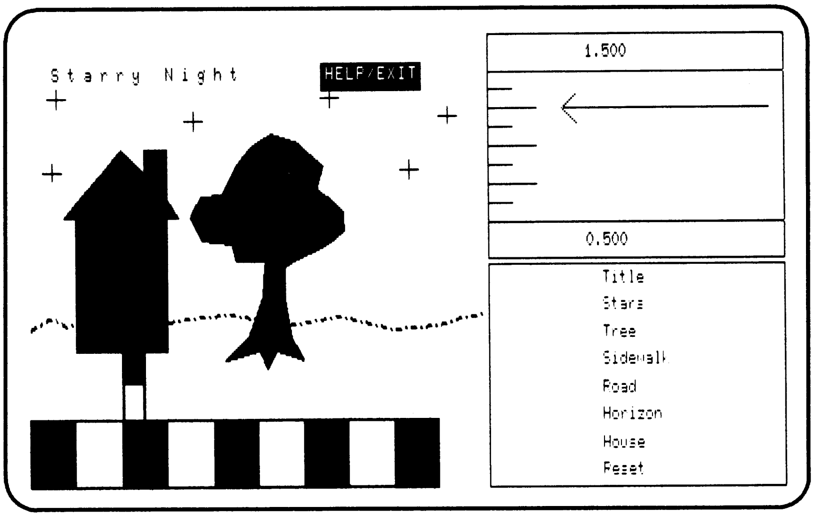
- Figure 7–27 shows the workstation surface after the user picks the help box and chooses to stop execution of the program.
- Figure 7–28 shows the workstation surface after the user triggers on the Exit choice.

Figure 7–19: Picking the Tree—VT241



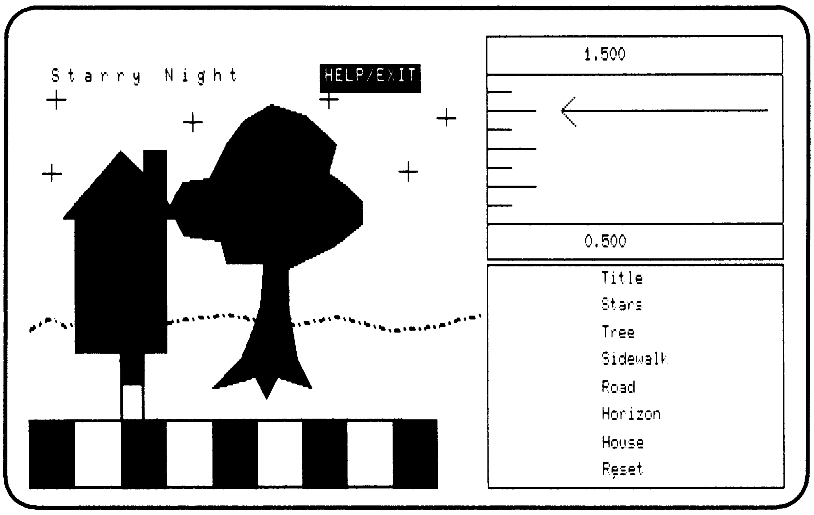
ZK-5958-HC

Figure 7–20: Choosing a Value—VT241



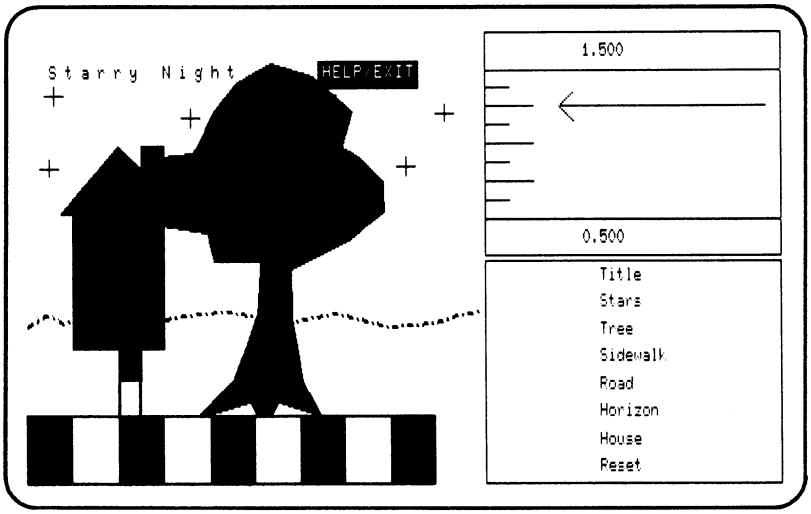
ZK-5959-HC

Figure 7–21: Triggering the Device—VT241



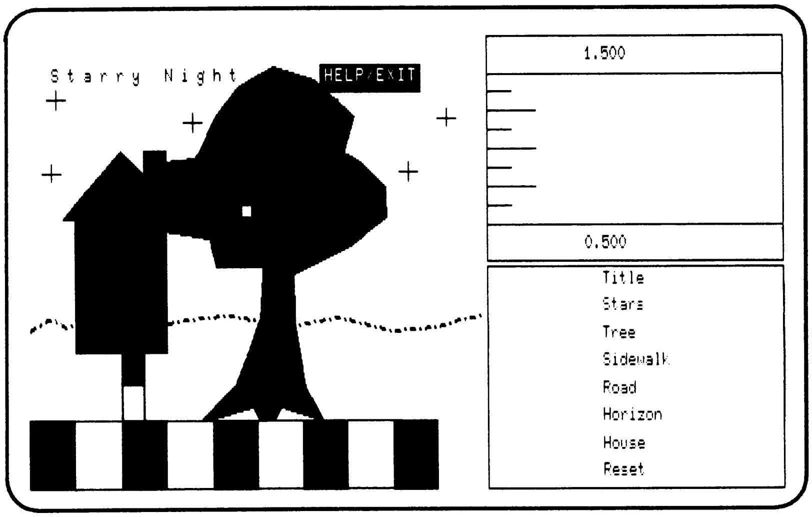
ZK-5960-HC

Figure 7–22: Triggering a Second Time—VT241



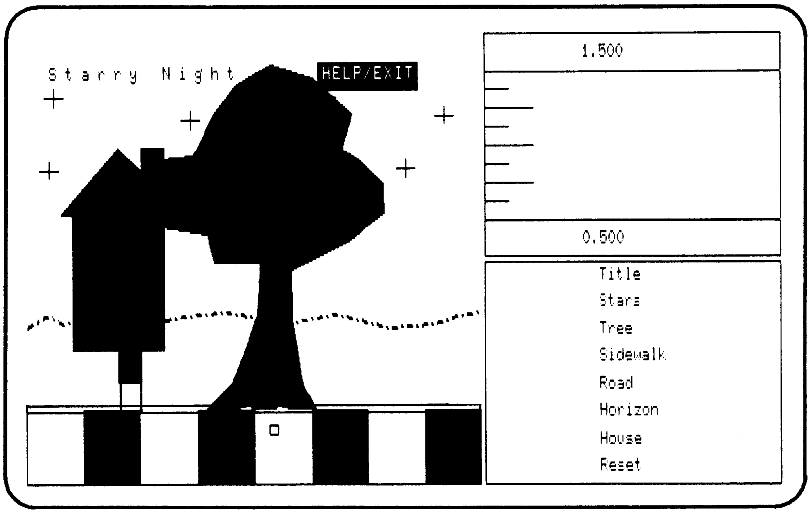
ZK-5961-HC

Figure 7–23: Stopping Scaling of the Tree—VT241



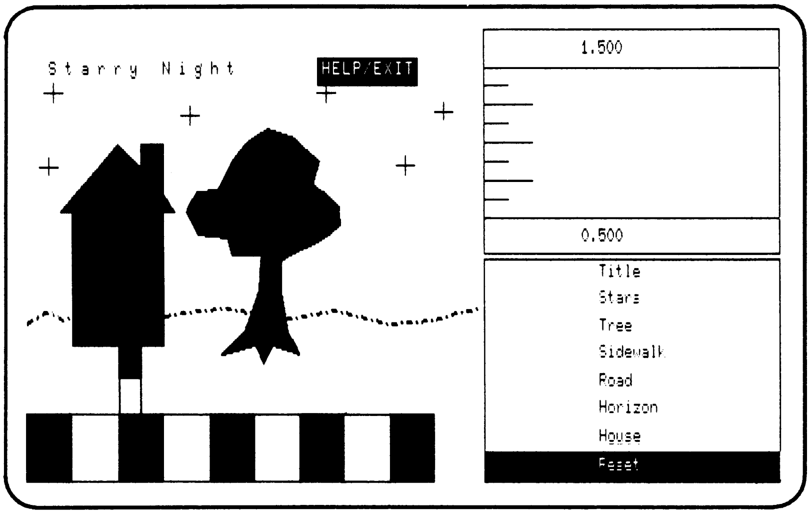
ZK-5962-HC

Figure 7–24: Picking the Road—VT241



ZK-5963-HC

Figure 7–25: Choosing to Reset the Picture—VT241



ZK-5964-HC

Figure 7–26: Picking the House—VT241

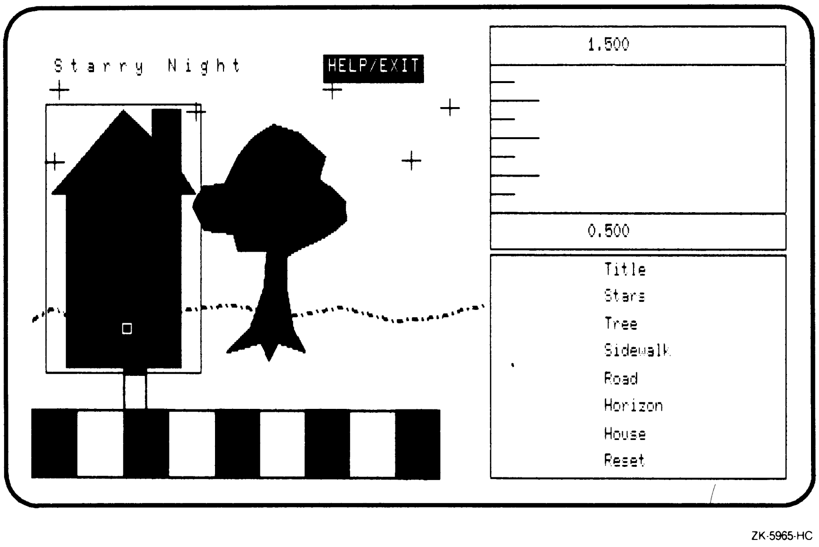
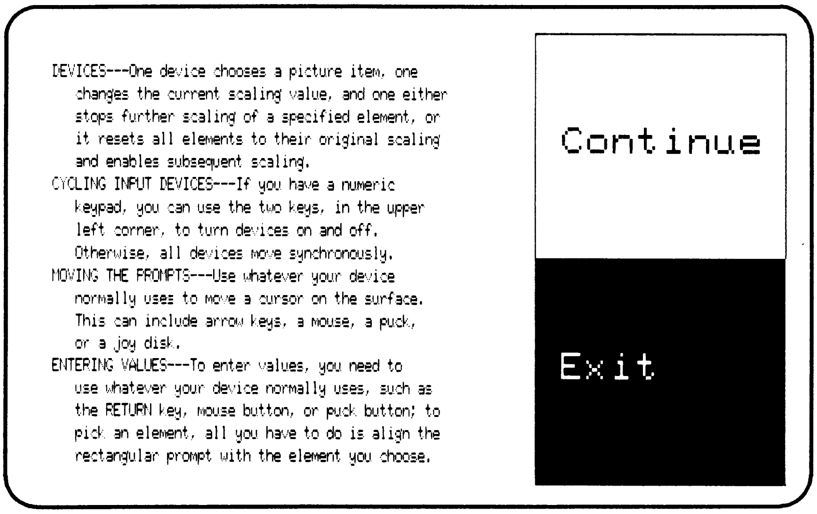


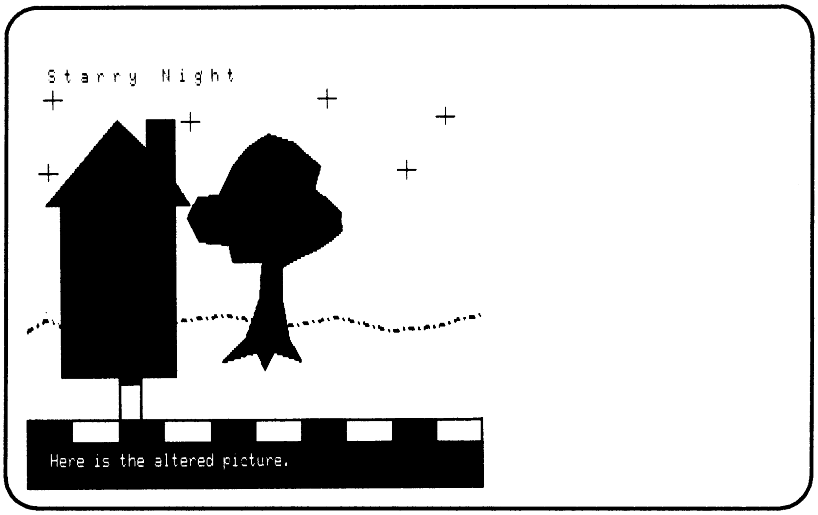
Figure 7–27: Choosing to Exit from the Program—VT241



DEVICES---One device chooses a picture item, one
   changes the current scaling value, and one either
   stops further scaling of a specified element, or
   it resets all elements to their original scaling
   and enables subsequent scaling.
CYCLING INPUT DEVICES---If you have a numeric
   keypad, you can use the two keys, in the upper
   left corner, to turn devices on and off.
   Otherwise, all devices move synchronously.
MOVING THE PROMPTS---Use whatever your device
   normally uses to move a cursor on the surface.
   This can include arrow keys, a mouse, a puck,
   or a joy disk.
ENTERING VALUES---To enter values, you need to
   use whatever your device normally uses, such as
   the RETURN key, mouse button, or puck button; to
   pick an element, all you have to do is align the
   rectangular prompt with the element you choose.

ZK-5966-HC

Figure 7–28: The Final Picture—VT241



ZK-5967-HC

7.5 Program Example Used in this Chapter

Example 7–1 presents all of the changes that you need to make to Example 3–2 in order to follow the code examples in this chapter.

Example 7–1: Using the DEC GKS Asynchronous Input Functions

```
      IMPLICIT NONE
      INTEGER WS_ID, WISS, HOUSE, TREE, HORIZON, STARS, TITLE,
     * SIDE, ROAD

      DATA WS_ID / 1 /, WISS / 2 /, TITLE / 1 /, STARS / 2 /,
     * TREE / 3 /, SIDE / 4 /, ROAD / 5 /, HORIZON / 6 /,
     * HOUSE / 7 /

      CALL SET_UP( WS_ID, WISS )
      CALL DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
      CALL CLEAN_UP( WS_ID, WISS )

      END

C     ***********************************************************
C     Set up the DEC GKS and the workstation environments...
      SUBROUTINE SET_UP( WS_ID, WISS )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, ERROR_STATUS, CATEGORY, INQUIRY_OKAY,
     * DUMMY_INTEGER, DEF_MODE, REGEN_FLAG, WS_TYPE, WISS,
     * GKS_LEVEL

      CHARACTER*80  DUMMY_STRING

      DATA INQUIRY_OKAY / 0 /

      CALL GKS$OPEN_GKS( 'temp.txt' )

      CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
     * ERROR_STATUS, CATEGORY )

C     Make sure that the workstation type is valid.
      IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
     *   (( CATEGORY .EQ. GKS$K_WSCAT_MI ) .OR.
     *   (( CATEGORY .EQ. GKS$K_WSCAT_MO ) .OR.
     *   (( CATEGORY .EQ. GKS$K_WSCAT_INPUT ) .OR.
     *   ( CATEGORY .EQ. GKS$K_WSCAT_WISS ))))) THEN
              WRITE(6,*)
     *        'The specified workstation type is invalid.'
              WRITE(6,*) 'Error status:', ERROR_STATUS
              STOP
      ENDIF

      CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT,
     * GKS$K_WSTYPE_DEFAULT )
      CALL GKS$ACTIVATE_WS( WS_ID )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Make sure that WISS is supported.
      CALL GKS$INQ_LEVEL( ERROR_STATUS, GKS_LEVEL )

      IF (( ERROR_STATUS .NE. INQUIRY_OKAY ) .OR.
     *   ( GKS_LEVEL .LT. GKS$K_LEVEL_2A )) THEN
             WRITE(6,*)
     *       'This level of GKS does not support WISS.'
             WRITE(6,*) 'Error status:', ERROR_STATUS
             STOP
      ENDIF

C     Open WISS so that you can store the help information.
      CALL GKS$OPEN_WS( WISS, GKS$K_CONID_DEFAULT,
     * GKS$K_WSTYPE_WISS )
         .
         .
         .
      RETURN
      END

C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZON )
         .
         .
         .
C     Ask the user for input...
      CALL GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS, TREE,
     * SIDE, ROAD, HOUSE, HORIZON )

      RETURN
      END

C     ************************************************************
C     Clean up the DEC GKS and the workstation environments...
      SUBROUTINE CLEAN_UP( WS_ID, WISS )

      INTEGER WS_ID

      CALL GKS$DEACTIVATE_WS( WS_ID )
      CALL GKS$CLOSE_WS( WS_ID )
      CALL GKS$CLOSE_WS( WISS )
      CALL GKS$CLOSE_GKS()

      RETURN
      END

C     ************************************************************
C     From this point forward, all code is additional code that you
C     need to add to the "Starry Night" program.
C     ************************************************************
```

(continued on next page)

Example 7-1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     ************************************************************
C     Coordinate user input...
      SUBROUTINE GO_FOR_INPUT( WS_ID, WS_TYPE, TITLE, STARS,
     * TREE, SIDE, ROAD, HOUSE, HORIZON )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, CATEGORY, ERROR_STATUS, FILL_PTS, FOREGROUND,
     * BACKGROUND, UNITY, HELP, HELP_BOX, WS_TYPE, WISS
      REAL FILL_X( 5 ), FILL_Y( 5 ), TEXT_EXTENT_X( 4 ),
     * TEXT_EXTENT_Y( 4 ), DUMMY_REAL( 4 )

      DATA FILL_PTS / 5 /, FOREGROUND / 1 /, BACKGROUND / 0 /,
     * UNITY / 0 /, HELP / 8 /, HELP_BOX / 9 /, WISS / 2 /

      DATA FILL_X / 0.0, 1.0, 1.0, 0.0, 0.0 /
      DATA FILL_Y / 0.0, 0.0, 0.3, 0.3, 0.0 /

      CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT,
     * ERROR_STATUS, CATEGORY )

      IF ( CATEGORY .NE. GKS$K_WSCAT_OUTIN ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'both input and output.'
           STOP
      ENDIF

C     Make sure that you are using the unity transformation...
      CALL GKS$SELECT_XFORM( UNITY )

C     Fill an area on which to send the user a message...
      CALL GKS$SET_FILL_COLOR_INDEX( FOREGROUND )
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )

C     Create the HELP segment.
      CALL CREATE_HELP( WS_ID, WISS, HELP )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Set the proper text attributes.
      CALL GKS$SET_TEXT_HEIGHT( 0.04 )
      CALL GKS$SET_TEXT_COLOR_INDEX( BACKGROUND )
      CALL GKS$SET_TEXT_SPACING( -0.3 )

C     Create the HELP/EXIT segments.

      CALL GKS$CREATE_SEG( HELP_BOX )
      CALL GKS$INQ_TEXT_EXTENT( WS_ID, 0.65, 0.9, 'HELP/EXIT',
     * ERROR_STATUS, DUMMY_REAL, DUMMY_REAL, TEXT_EXTENT_X,
     * TEXT_EXTENT_Y )
      TEXT_EXTENT_X( 1 ) = TEXT_EXTENT_X( 1 ) - 0.01
      TEXT_EXTENT_X( 4 ) = TEXT_EXTENT_X( 4 ) - 0.01
      TEXT_EXTENT_X( 2 ) = TEXT_EXTENT_X( 2 ) + 0.01
      TEXT_EXTENT_X( 3 ) = TEXT_EXTENT_X( 3 ) + 0.01
      TEXT_EXTENT_Y( 1 ) = TEXT_EXTENT_Y( 1 ) - 0.01
      TEXT_EXTENT_Y( 2 ) = TEXT_EXTENT_Y( 2 ) - 0.01
      TEXT_EXTENT_Y( 3 ) = TEXT_EXTENT_Y( 3 ) + 0.01
      TEXT_EXTENT_Y( 4 ) = TEXT_EXTENT_Y( 4 ) + 0.01
      CALL GKS$FILL_AREA( 4, TEXT_EXTENT_X, TEXT_EXTENT_Y )
      CALL GKS$TEXT( 0.65, 0.9,
     * 'HELP/EXIT' )
      CALL GKS$CLOSE_SEG()

C     Initialize all input devices.
      CALL INIT_DEVICES( WS_ID, WS_TYPE )


C     Make sure that all of the segments are detectable...
      CALL GKS$SET_SEG_DETECTABILITY( TITLE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( STARS, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( TREE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( SIDE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( ROAD, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HOUSE, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HORIZON, GKS$K_DETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HELP, GKS$K_UNDETECTABLE )
      CALL GKS$SET_SEG_DETECTABILITY( HELP_BOX, GKS$K_DETECTABLE )

C     Reset the attribute values and the message board.
      CALL GKS$SET_TEXT_HEIGHT( 0.033 )
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )

      CALL GKS$TEXT( 0.05, 0.25,
     * 'When prompted, scale the picture elements.' )
      CALL GKS$TEXT( 0.05, 0.20,
     * 'If you need help or if you are ready to finish,' )
      CALL GKS$TEXT( 0.05, 0.15,
     * 'move the square prompt to HELP/EXIT.' )
      CALL GKS$TEXT( 0.05, 0.02,
     * '(Press RETURN when ready.)' )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     The user presses RETURN when ready to pick...
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

C     Erase the message and redraw the segments...
      CALL GKS$REDRAW_SEG_ON_WS( WS_ID )

C     Get the input values.
      CALL GET_VALUES( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON, HELP, HELP_BOX, WS_TYPE)

C     Show the final picture...
      FILL_Y( 3 ) = 0.1
      FILL_Y( 4 ) = 0.1
      CALL GKS$FILL_AREA( FILL_PTS, FILL_X, FILL_Y )
      CALL GKS$TEXT( 0.05, 0.05,
     * 'Here is the altered picture.' )

C     Press RETURN when finished viewing the picture.
      CALL GKS$UPDATE_WS( WS_ID, GKS$K_POSTPONE_FLAG )
      READ(5,*)

      RETURN
      END

C     ************************************************************
C     This subroutine creates a help screen...
      SUBROUTINE CREATE_HELP( WS_ID, WISS, HELP )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WISS, HELP, BACKGROUND

      DATA BACKGROUND / 1 /

C     Only create the help screen on WISS.
      CALL GKS$DEACTIVATE_WS( WS_ID )
      CALL GKS$ACTIVATE_WS( WISS )

C     Set the necessary attributes.
      CALL GKS$SET_TEXT_HEIGHT( 0.033 )
      CALL GKS$SET_TEXT_SPACING( -0.4 )
      CALL GKS$SET_TEXT_COLOR_INDEX( BACKGROUND )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Create the help screen.
      CALL GKS$CREATE_SEG( HELP )
      CALL GKS$TEXT( 0.05, 0.9,
     * 'DEVICES---One device chooses a picture item, one' )
      CALL GKS$TEXT( 0.1, 0.85,
     * 'changes the current scaling value, and one either' )
      CALL GKS$TEXT( 0.1, 0.80,
     * 'stops further scaling of a specified element, or' )
      CALL GKS$TEXT( 0.1, 0.75,
     * 'it resets all elements to their original scaling' )
      CALL GKS$TEXT( 0.1, 0.70,
     * 'and enables subsequent scaling.')

      CALL GKS$TEXT( 0.05, 0.65,
     * 'CYCLING INPUT DEVICES---If you have a numeric')
      CALL GKS$TEXT( 0.1, 0.60,
     * 'keypad, you can use the two keys, in the upper' )
      CALL GKS$TEXT( 0.1, 0.55,
     * 'left corner, to turn devices on and off.' )
      CALL GKS$TEXT( 0.1, 0.50,
     * 'Otherwise, all devices move synchronously.' )
      CALL GKS$TEXT( 0.05, 0.45,
     * 'MOVING THE PROMPTS---Use whatever your device' )
      CALL GKS$TEXT( 0.1, 0.40,
     * 'normally uses to move a cursor on the surface.' )
      CALL GKS$TEXT( 0.1, 0.35,
     * 'This can include arrow keys, a mouse, a puck,' )
      CALL GKS$TEXT( 0.1, 0.30,
     * 'or a joy disk.' )
      CALL GKS$TEXT( 0.05, 0.25,
     * 'ENTERING VALUES---To enter values, you need to ' )
      CALL GKS$TEXT( 0.1, 0.20,
     * 'use whatever your device normally uses, such as ' )
      CALL GKS$TEXT( 0.1, 0.15,
     * 'the RETURN key, mouse button, or puck button; to' )
      CALL GKS$TEXT( 0.1, 0.10,
     * 'pick an element, all you have to do is align the ' )
      CALL GKS$TEXT( 0.1, 0.05,
     * 'rectangular prompt with the element you choose.' )
      CALL GKS$TEXT( 0.1, 0.001,
     * 'Do you want to QUIT or CONTINUE?' )

      CALL GKS$CLOSE_SEG()
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Associate the help screen and reactivate the workstation.
      CALL GKS$SET_SEG_VISIBILITY( HELP, GKS$K_INVISIBLE )
      CALL GKS$ASSOC_SEG_WITH_WS( WS_ID, HELP )
      CALL GKS$DEACTIVATE_WS( WISS )
      CALL GKS$ACTIVATE_WS( WS_ID )

      RETURN
      END

C     ***********************************************************
C     This subroutine initializes all input devices...
      SUBROUTINE INIT_DEVICES( WS_ID, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, WS_TYPE, ERROR_STATUS, DUMMY_INTEGER,
     * NUM_PICK_DEVICES, DEVICE_NUM, INPUT_MODE, ECHO_FLAG,
     * INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
     * PROMPT_ECHO_TYPE, DATA_RECORD, RECORD_BUFFER_LENGTH,
     * RECORD_SIZE, PICK_DATA_RECORD( 10 ),
     * CHOICE_DATA_RECORD( 3 ), NUM_VAL_DEVICES, NUM_CHOICES,
     * NUM_CHOICE_DEVICES, SIZES( 10 ), ADDRESSES( 10 ),
     * LIST_PROMPT_TYPES( 10 ), PROMPT_RETURN_SIZE, PROMPT_FLAG,
     * INITIAL_CHOICE, INCR, AREA_FLAG, VAL_RECORD_BUFFER_LENGTH,
     * VAL_PROMPT_ECHO_TYPE
      REAL CHOICE_ECHO_AREA( 4 ), VAL_ECHO_AREA( 4 ),
     * ECHO_AREA( 4 ), DUMMY_ARRAY( 4 ), VAL_DATA_RECORD( 2 ),
     * VALUE, UPPER_LIMIT, LOWER_LIMIT, MAX_COORD, DISPLAY_X,
     * DISPLAY_Y

      CHARACTER*80 DEFAULT_STRINGS( 2 )

      DATA DEVICE_NUM / 1 /

C     First element in the data record is the number of choices.
      EQUIVALENCE( CHOICE_DATA_RECORD( 1 ), NUM_CHOICES )

C     According to the standard, the elements in the data record are
C     the upper and lower limits for all prompt and echo types.
      EQUIVALENCE( VAL_DATA_RECORD( 1 ), LOWER_LIMIT )
      EQUIVALENCE( VAL_DATA_RECORD( 2 ), UPPER_LIMIT )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Initialize the pick device.
C     Make sure that the device supports pick input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * DUMMY_INTEGER, NUM_PICK_DEVICES, DUMMY_INTEGER )

      IF ( NUM_PICK_DEVICES .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'pick input.'
           STOP
      ENDIF

C     Give the data record the size of your data record buffer and
C     inquire about the realized pick values.
      RECORD_BUFFER_LENGTH = 40
      CALL GKS$INQ_PICK_STATE( WS_ID, DEVICE_NUM,
     * GKS$K_VALUE_REALIZED, ERROR_STATUS, INPUT_MODE,
     * ECHO_FLAG, INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
     * PROMPT_ECHO_TYPE, ECHO_AREA, PICK_DATA_RECORD,
     * RECORD_BUFFER_LENGTH, RECORD_SIZE )

C     Make sure that the data record was not truncated...
      IF ( RECORD_SIZE .LT. RECORD_BUFFER_LENGTH ) THEN
           WRITE(6,*) 'The data record was truncated.'
           WRITE(6,*) 'Declare a larger buffer.'
           STOP
      ENDIF
C     Make sure that the pick aperture is not placed on any segment.
      INITIAL_STATUS = GKS$K_STATUS_NOPICK

C     Make sure that the device is in request mode (the DEC GKS default).
      CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

C     Initialize the device...
      CALL GKS$INIT_PICK( WS_ID, DEVICE_NUM,
     * INITIAL_STATUS, PICKED_SEGMENT, PICK_ID,
     * PROMPT_ECHO_TYPE, ECHO_AREA, PICK_DATA_RECORD,
     * RECORD_BUFFER_LENGTH )

C     Initialize the choice and valuator devices...
C     Make sure that the device supports choice input...
      CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NUM_CHOICE_DEVICES, DUMMY_INTEGER, DUMMY_INTEGER )

      IF ( NUM_CHOICE_DEVICES .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'choice input.'
           STOP
      ENDIF
```

(continued on next page)

```
C      Make sure that the device supports valuator input...
       CALL GKS$INQ_INPUT_DEV( WS_TYPE, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, NUM_VAL_DEVICES,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER )

       IF ( NUM_VAL_DEVICES .EQ. 0 ) THEN
            WRITE(6,*) 'The workstation does not support'
            WRITE(6,*) 'valuator input.'
            STOP
       ENDIF

C      Obtain the default valuator values...
       VAL_RECORD_BUFFER_LENGTH = 8
       CALL GKS$INQ_VALUATOR_STATE( WS_ID, DEVICE_NUM,
     * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, VALUE,
     * VAL_PROMPT_ECHO_TYPE, VAL_ECHO_AREA, VAL_DATA_RECORD,
     * VAL_RECORD_BUFFER_LENGTH, RECORD_SIZE )

C      Establish the size of the choice record buffer: 12 bytes.
       RECORD_BUFFER_LENGTH = 12

C      The second element in the choice data record for prompt and echo type 1
C      is the pointer to the array containing sizes of each choice character
C      string. You need to initialize the pointer so that the array can be
C      initialized.
       CHOICE_DATA_RECORD( 2 ) = %LOC( SIZES( 1 ) )

C      The third element in the VT241 choice data record is the pointer to the
C      array containing the pointers to the strings to be used. You need
C      to initialize the pointer so that the array can be initialized.
       CHOICE_DATA_RECORD( 3 ) = %LOC( ADDRESSES( 1 ) )
       ADDRESSES( 1 ) = %LOC( DEFAULT_STRINGS( 1 ) )
       ADDRESSES( 2 ) = %LOC( DEFAULT_STRINGS( 2 ) )

C      There are ten choices.
       NUM_CHOICES = 10

C      Obtain the available prompt and echo types...
       CALL GKS$INQ_DEF_CHOICE_DATA( WS_TYPE, DEVICE_NUM,
     * ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
     * %DESCR( LIST_PROMPT_TYPES), CHOICE_ECHO_AREA,
     * CHOICE_DATA_RECORD, PROMPT_RETURN_SIZE,
     * RECORD_BUFFER_LENGTH, RECORD_SIZE )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Obtain the remaining default input values...
      CALL GKS$INQ_CHOICE_STATE( WS_ID, DEVICE_NUM,
     * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, INITIAL_STATUS,
     * INITIAL_CHOICE, PROMPT_ECHO_TYPE, CHOICE_ECHO_AREA,
     * CHOICE_DATA_RECORD, RECORD_BUFFER_LENGTH, RECORD_SIZE )

      DO 400 INCR = 1, PROMPT_RETURN_SIZE, 1
      IF ( LIST_PROMPT_TYPES( INCR ) .EQ. 3 ) THEN
           PROMPT_FLAG = 1
      ENDIF
400   CONTINUE

C     If the workstation does not support prompt and echo type 3...
      IF ( PROMPT_FLAG .EQ. 0 ) THEN
           WRITE(6,*) 'The workstation does not support'
           WRITE(6,*) 'choice prompt and echo type 3.'
           STOP
      ENDIF

C     Make sure that the two echo areas don't conflict...
      IF ((( CHOICE_ECHO_AREA( 1 ) .EQ. VAL_ECHO_AREA( 1 )  ) .OR.
     *       ( CHOICE_ECHO_AREA( 2 ) .EQ. VAL_ECHO_AREA( 2 )  )) .OR.
     *    (( CHOICE_ECHO_AREA( 3 ) .EQ. VAL_ECHO_AREA( 3 )  ) .OR.
     *    ( CHOICE_ECHO_AREA( 4 ) .EQ. VAL_ECHO_AREA( 4 )  ))) THEN

         CALL GKS$INQ_MAX_DS_SIZE( WS_TYPE, ERROR_STATUS,
     *    DUMMY_INTEGER, DISPLAY_X, DISPLAY_Y, DUMMY_INTEGER,
     *    DUMMY_INTEGER )

         MAX_COORD = MAX( DISPLAY_X, DISPLAY_Y )

         IF ( DISPLAY_X .NE. DISPLAY_Y ) THEN
         IF (( DISPLAY_X / MAX_COORD ) .EQ. 1.0 ) THEN

             CHOICE_ECHO_AREA( 1 ) = DISPLAY_X -
     *                      ( DISPLAY_X - DISPLAY_Y )
             CHOICE_ECHO_AREA( 2 ) = DISPLAY_X
             CHOICE_ECHO_AREA( 3 ) = 0.0
             CHOICE_ECHO_AREA( 4 ) = DISPLAY_Y / 2.02
             VAL_ECHO_AREA( 1 ) = DISPLAY_X -
     *                      ( DISPLAY_X - DISPLAY_Y )
             VAL_ECHO_AREA( 2 ) = DISPLAY_X
             VAL_ECHO_AREA( 3 ) = DISPLAY_Y / 1.98
             VAL_ECHO_AREA( 4 ) = DISPLAY_Y
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C          Make sure the pick area does not conflict...
           DO WHILE ( ECHO_AREA( 2 ) .GE. VAL_ECHO_AREA( 1 ) )
              VAL_ECHO_AREA( 1 ) = VAL_ECHO_AREA( 1 ) +
     *                          ( VAL_ECHO_AREA( 1 ) / 100 )
              CHOICE_ECHO_AREA( 1 ) = CHOICE_ECHO_AREA( 1 ) +
     *                          ( CHOICE_ECHO_AREA( 1 ) / 100 )

           ENDDO

        ELSE

           CHOICE_ECHO_AREA( 1 ) = 0.0
           CHOICE_ECHO_AREA( 2 ) = DISPLAY_X / 2.02
           CHOICE_ECHO_AREA( 3 ) = DISPLAY_Y -
     *                     ( DISPLAY_Y - DISPLAY_X )
           CHOICE_ECHO_AREA( 4 ) = DISPLAY_Y
           VAL_ECHO_AREA( 1 ) = DISPLAY_X / 1.98
           VAL_ECHO_AREA( 2 ) = DISPLAY_X
           VAL_ECHO_AREA( 3 ) = DISPLAY_Y -
     *                     ( DISPLAY_Y - DISPLAY_X )
           VAL_ECHO_AREA( 4 ) = DISPLAY_Y

C          Make sure the pick area does not conflict...
           DO WHILE ( ECHO_AREA( 4 ) .GE. VAL_ECHO_AREA( 3 ) )
              VAL_ECHO_AREA( 1 ) = VAL_ECHO_AREA( 1 ) +
     *                          ( VAL_ECHO_AREA( 1 ) / 100 )
              CHOICE_ECHO_AREA( 1 ) = CHOICE_ECHO_AREA( 1 ) +
     *                          ( CHOICE_ECHO_AREA( 1 ) / 100 )

           ENDDO

        ENDIF    !  MAX_COORD equals DISPLAY_X or DISPLAY_Y
        ELSE     !  ELSE, if the surface is square...

          WRITE(6,*) 'The workstation surface is square.'
          WRITE(6,*) 'Any echo area I pick will cover'
          WRITE(6,*) 'part of the picture.  You need to'
          WRITE(6,*) 'alter program transformations.'
          STOP

        ENDIF    !  If the surface is square
      ENDIF   ! If the echo areas conflict.

C     Initialize the choice device...
      PROMPT_ECHO_TYPE = 3
      INITIAL_CHOICE = 1
      NUM_CHOICES = 8
      INITIAL_STATUS = GKS$K_STATUS_NOCHOICE
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Establish sizes of prompt strings...
      SIZES( 1 ) = 5
      SIZES( 2 ) = 5
      SIZES( 3 ) = 4
      SIZES( 4 ) = 8
      SIZES( 5 ) = 4
      SIZES( 6 ) = 7
      SIZES( 7 ) = 5
      SIZES( 8 ) = 5
C     Establish locations of prompt strings...
      ADDRESSES( 1 ) = %LOC( 'Title' )
      ADDRESSES( 2 ) = %LOC( 'Stars' )
      ADDRESSES( 3 ) = %LOC( 'Tree' )
      ADDRESSES( 4 ) = %LOC( 'Sidewalk' )
      ADDRESSES( 5 ) = %LOC( 'Road' )
      ADDRESSES( 6 ) = %LOC( 'Horizon' )
      ADDRESSES( 7 ) = %LOC( 'House' )
      ADDRESSES( 8 ) = %LOC( 'Reset' )

C     Make sure that the device is in request mode (the DEC GKS default).
      CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

      CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
     * INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
     * CHOICE_ECHO_AREA, CHOICE_DATA_RECORD,
     * RECORD_BUFFER_LENGTH )

C     Initialize the valuator device...
      VALUE = 1.0
      UPPER_LIMIT = 1.5
      LOWER_LIMIT = 0.5

C     Make sure that the device is in request mode (the DEC GKS default).
      CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )

      CALL GKS$INIT_VALUATOR( WS_ID, DEVICE_NUM,
     * VALUE, VAL_PROMPT_ECHO_TYPE, VAL_ECHO_AREA,
     * VAL_DATA_RECORD, VAL_RECORD_BUFFER_LENGTH )

      RETURN
      END
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     *************************************************************
C     This subroutine obtains input values...
      SUBROUTINE GET_VALUES( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON, HELP, HELP_BOX, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, HELP, HELP_BOX, DEVICE_NUM, FINISHED_FLAG,
     * PICKED_SEGMENT, PICK_INPUT_STATUS, INPUT_STATUS,
     * PICK_ID, CLASS, ERROR_STATUS, INPUT_CHOICE, RESET,
     * MORE_EVENTS_FLAG, DUMMY_INTEGER, NEW_FRAME_FLAG, WS_TYPE,
     * CURRENT_SEGMENT, INITIAL_STATUS, PROMPT_ECHO_TYPE,
     * PICK_DATA_RECORD( 10 ), RECORD_BUFFER_LENGTH, REPEAT_FLAG,
     * HELP_FLAG, VALUE_FLAG, LOCKED_SEGMENT, INCR
      REAL IDENTITY( 6 ), TITLE_XFORM_MATRIX( 6 ),
     * STARS_XFORM_MATRIX( 6 ), HOUSE_XFORM_MATRIX( 6 ),
     * TREE_XFORM_MATRIX( 6 ), SIDE_XFORM_MATRIX( 6 ),
     * ROAD_XFORM_MATRIX( 6 ), HORIZON_XFORM_MATRIX( 6 ), VALUE,
     * FIXED_X, FIXED_Y, CURRENT_VALUE, ECHO_AREA( 4 )

      DATA DEVICE_NUM / 1 /, FINISHED_FLAG / 0 /, RESET / 8 /

C     Place the devices in the proper input mode.
      CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_SAMPLE, GKS$K_ECHO )
      CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )
      CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )

C     Create an identity matrix and initial transformation matrixes.
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, IDENTITY )
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, TITLE_XFORM_MATRIX )
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, STARS_XFORM_MATRIX )
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, SIDE_XFORM_MATRIX )
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, ROAD_XFORM_MATRIX )
      CALL GKS$EVAL_XFORM_MATRIX( 0.0, 0.0, 0.0, 0.0, 0.0,
     * 1.0, 1.0, GKS$K_COORDINATES_NDC, HORIZON_XFORM_MATRIX )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
      HELP_FLAG = 0
      REPEAT_FLAG = 0
      VALUE = 1.0
      VALUE_FLAG = 0
      CURRENT_VALUE = 1.0
      LOCKED_SEGMENT = 0
      DO WHILE ( FINISHED_FLAG .NE. 1 )

      CALL GKS$SAMPLE_PICK( WS_ID, DEVICE_NUM,
     * PICK_INPUT_STATUS, PICKED_SEGMENT, PICK_ID )

      IF (( LOCKED_SEGMENT .NE. 0 ) .AND.
     *    ( PICKED_SEGMENT .NE. LOCKED_SEGMENT )) THEN
            LOCKED_SEGMENT = 0
      ENDIF

      IF (( PICKED_SEGMENT .NE. HELP_BOX ) .AND.
     *    ( HELP_FLAG .EQ. 1 )) THEN
           HELP_FLAG = 0
      ENDIF

      IF ( HELP_FLAG .EQ. 1 ) THEN
            PICKED_SEGMENT = TITLE
            PICK_INPUT_STATUS = GKS$K_STATUS_NOPICK
      ENDIF

      IF (( PICKED_SEGMENT .EQ. HELP_BOX ) .AND.
     *    ( HELP_FLAG .NE. 1 )) THEN

           CALL GET_HELP( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     *                    HOUSE, HORIZON, HELP, HELP_BOX,
     *                    FINISHED_FLAG, WS_TYPE )

           HELP_FLAG = 1

      ELSE

C     Check the event queue.
      CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )

C     Check for queue overflow.
      CALL GKS$INQ_INPUT_QUEUE_OVERFLOW( ERROR_STATUS, WS_ID,
     * CLASS, DEVICE_NUM )

C     If the queue has overflowed...
      IF ( ERROR_STATUS .EQ. 0 ) THEN
      CALL GKS$FLUSH_DEVICE_EVENTS( WS_ID,
     * GKS$K_INPUT_CLASS_VALUATOR, DEVICE_NUM )
      CALL GKS$FLUSH_DEVICE_EVENTS( WS_ID,
     * GKS$K_INPUT_CLASS_CHOICE, DEVICE_NUM )
      CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )
      ENDIF

100   CONTINUE
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
      IF ( CLASS .EQ. GKS$K_INPUT_CLASS_CHOICE ) THEN
          CALL GKS$GET_CHOICE( INPUT_STATUS, INPUT_CHOICE )

          IF ( INPUT_STATUS .NE. GKS$K_STATUS_NOCHOICE ) THEN
          IF ( INPUT_CHOICE .NE. RESET ) THEN
              CALL GKS$SET_SEG_DETECTABILITY( INPUT_CHOICE,
     *                                        GKS$K_UNDETECTABLE )
C     Don't let the user scale the segment any more.
              LOCKED_SEGMENT = PICKED_SEGMENT
          ELSE

              CALL GKS$SET_SEG_XFORM( TITLE, IDENTITY )
              DO 200 INCR = 1, 6, 1
              TITLE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
200           CONTINUE
              CALL GKS$SET_SEG_XFORM( STARS, IDENTITY )
              DO 300 INCR = 1, 6, 1
              STARS_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
300           CONTINUE
              CALL GKS$SET_SEG_XFORM( HOUSE, IDENTITY )
              DO 400 INCR = 1, 6, 1
              HOUSE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
400           CONTINUE
              CALL GKS$SET_SEG_XFORM( TREE, IDENTITY )
              DO 500 INCR = 1, 6, 1
              TREE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
500           CONTINUE
              CALL GKS$SET_SEG_XFORM( SIDE, IDENTITY )
              DO 600 INCR = 1, 6, 1
              SIDE_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
600           CONTINUE
              CALL GKS$SET_SEG_XFORM( ROAD, IDENTITY )
              DO 700 INCR = 1, 6, 1
              ROAD_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
700           CONTINUE
              CALL GKS$SET_SEG_XFORM( HORIZON, IDENTITY )
              DO 800 INCR = 1, 6, 1
              HORIZON_XFORM_MATRIX( INCR ) = IDENTITY( INCR )
800           CONTINUE
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
           CALL GKS$SET_SEG_DETECTABILITY( TITLE,
     *                                     GKS$K_DETECTABLE )
           CALL GKS$SET_SEG_DETECTABILITY( STARS,
     *                                     GKS$K_DETECTABLE )
           CALL GKS$SET_SEG_DETECTABILITY( HOUSE,
     *                                     GKS$K_DETECTABLE )
           CALL GKS$SET_SEG_DETECTABILITY( TREE,
     *                                     GKS$K_DETECTABLE )
           CALL GKS$SET_SEG_DETECTABILITY( SIDE,
     *                                     GKS$K_DETECTABLE )
           CALL GKS$SET_SEG_DETECTABILITY( ROAD,
     *                                     GKS$K_DETECTABLE )
           CALL GKS$SET_SEG_DETECTABILITY( HORIZON,
     *                                     GKS$K_DETECTABLE )
C     Check to see whether the picture on the screen is out of date.
        CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     *      DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     *      NEW_FRAME_FLAG )

C     Release deferred output. Regenerate if necessary.
         IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
            CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
         ENDIF

         ENDIF  !  If choice does not equal reset.
         ENDIF  !  If input status does not equal no choice.
      ENDIF       !  If input class equals choice.

      IF ( CLASS .EQ. GKS$K_INPUT_CLASS_VALUATOR ) THEN
           CALL GKS$GET_VALUATOR( VALUE )
           VALUE_FLAG = 1
      ENDIF

C     Check for simultaneously entered events...
      CALL GKS$INQ_MORE_SIMUL_EVENTS( ERROR_STATUS,
     * MORE_EVENTS_FLAG )

C     If there are more simultaneous events, take them from the queue...
      IF ( MORE_EVENTS_FLAG .EQ. GKS$K_MORE_EVENTS) THEN
         CALL GKS$AWAIT_EVENT( 0.0, WS_ID, CLASS, DEVICE_NUM )
         GOTO 100
      ENDIF
```

(continued on next page)

```
      IF ( PICK_INPUT_STATUS .EQ. GKS$K_STATUS_NOPICK ) THEN
           REPEAT_FLAG = 1
      ELSEIF ( VALUE .EQ. 1.0 ) THEN
           REPEAT_FLAG = 1
      ELSEIF (( VALUE_FLAG .EQ. 0 ) .AND.
     *        ( PICKED_SEGMENT .EQ. CURRENT_SEGMENT )) THEN
           REPEAT_FLAG = 1
      ELSEIF ( PICKED_SEGMENT .EQ. LOCKED_SEGMENT ) THEN
           REPEAT_FLAG = 1
      ENDIF

C     Establish fixed points for segments depending on picked segment.
      IF ( REPEAT_FLAG .EQ. 0 ) THEN
      IF ( PICKED_SEGMENT .EQ. TITLE ) THEN
         FIXED_X = 0.3
         FIXED_Y = 0.925
         CALL GKS$ACCUM_XFORM_MATRIX( TITLE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, TITLE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( TITLE, TITLE_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. STARS ) THEN
         FIXED_X = 0.5
         FIXED_Y = 0.8
         CALL GKS$ACCUM_XFORM_MATRIX( STARS_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, STARS_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( STARS, STARS_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. TREE ) THEN
         FIXED_X = 0.52
         FIXED_Y = 0.51
         CALL GKS$ACCUM_XFORM_MATRIX( TREE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, TREE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( TREE, TREE_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. SIDE ) THEN
         FIXED_X = 0.225
         FIXED_Y = 0.22
         CALL GKS$ACCUM_XFORM_MATRIX( SIDE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, SIDE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( SIDE, SIDE_XFORM_MATRIX )

      ELSEIF ( PICKED_SEGMENT .EQ. ROAD ) THEN
         FIXED_X = 0.5
         FIXED_Y = 0.075
         CALL GKS$ACCUM_XFORM_MATRIX( ROAD_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, ROAD_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( ROAD, ROAD_XFORM_MATRIX )
```

(continued on next page)

```
      ELSEIF ( PICKED_SEGMENT .EQ.  HORIZON ) THEN
         FIXED_X = 0.1
         FIXED_Y = 0.35
         CALL GKS$ACCUM_XFORM_MATRIX( HORIZON_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, HORIZON_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( HORIZON, HORIZON_XFORM_MATRIX )
      ELSEIF ( PICKED_SEGMENT .EQ. HOUSE ) THEN
         FIXED_X = 0.2
         FIXED_Y = 0.5
         CALL GKS$ACCUM_XFORM_MATRIX( HOUSE_XFORM_MATRIX,
     *        FIXED_X, FIXED_Y, 0.0, 0.0, 0.0, VALUE, VALUE,
     *        GKS$K_COORDINATES_NDC, HOUSE_XFORM_MATRIX )
         CALL GKS$SET_SEG_XFORM( HOUSE, HOUSE_XFORM_MATRIX )
      ENDIF   !  Scaling.

C     Check to see whether the picture on the screen is out of date.
      CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NEW_FRAME_FLAG )

C     Release deferred output. Regenerate if necessary.
      IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
          CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

      ENDIF   !  If we aren't repeating a transformation...

C     Reset the repeating transformations flag.
      REPEAT_FLAG = 0

      ENDIF  ! If segment equals help.

C     Set the current segment, current value, and entered value flag...
      CURRENT_SEGMENT = PICKED_SEGMENT
      CURRENT_VALUE = VALUE
      VALUE_FLAG = 0

      ENDDO

      RETURN
      END
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     **********************************************************
C     This subroutine makes the help screen visible...
      SUBROUTINE GET_HELP( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
     * HOUSE, HORIZON, HELP, HELP_BOX, FINISHED_FLAG, WS_TYPE )

      IMPLICIT NONE
      INCLUDE 'SYS$LIBRARY:GKSDEFS.FOR'
      INTEGER WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZON, HELP, HELP_BOX, DEVICE_NUM, FINISHED_FLAG,
     * NEW_FRAME_FLAG, DUMMY_INTEGER, ERROR_STATUS, INPUT_MODE,
     * ECHO_FLAG, INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
     * CHOICE_DATA_RECORD( 3 ), RECORD_BUFFER_LENGTH, RECORD_SIZE,
     * TEMP_DATA_RECORD( 3 ), TEMP_RBL, TEMP_RECORD_SIZE,
     * LIST_PROMPT_TYPES, PROMPT_RETURN_SIZE, TEMP_INITIAL_STATUS,
     * TEMP_INITIAL_CHOICE, NUM_CHOICES, SIZES( 10 ),
     * ADDRESSES( 10 ), TEMP_SIZES( 2 ), TEMP_ADDRESSES( 2 ),
     * INPUT_STATUS, CHOICE, WS_TYPE, CONTINUE

      CHARACTER*80 CURRENT_STRINGS( 10 ),
     * TEMP_CURRENT_STRINGS( 2 )

      REAL ECHO_AREA( 4 ), TEMP_ECHO_AREA( 4 )

      DATA DEVICE_NUM / 1 /, CONTINUE / 1 /

      EQUIVALENCE( CHOICE_DATA_RECORD( 1 ), NUM_CHOICES )

C     Reset visibility of all segments and deactivate the input prompts.
      CALL GKS$SET_SEG_VISIBILITY( TITLE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( STARS, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( TREE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( SIDE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( ROAD, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HOUSE, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HELP_BOX, GKS$K_INVISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HELP, GKS$K_VISIBLE )

      CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
      CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
      CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
     * GKS$K_INPUT_MODE_REQUEST, GKS$K_ECHO )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Store the current choice values...
      CHOICE_DATA_RECORD( 1 ) = 10
      CHOICE_DATA_RECORD( 2 ) = %LOC( SIZES( 1 ) )
      SIZES( 1 ) = 80
      SIZES( 2 ) = 80
      SIZES( 3 ) = 80
      SIZES( 4 ) = 80
      SIZES( 5 ) = 80
      SIZES( 6 ) = 80
      SIZES( 7 ) = 80
      SIZES( 8 ) = 80
      SIZES( 9 ) = 80
      SIZES( 10 ) = 80
      CHOICE_DATA_RECORD( 3 ) = %LOC( ADDRESSES( 1 ) )
      ADDRESSES( 1 ) = %LOC( CURRENT_STRINGS( 1 ) )
      ADDRESSES( 2 ) = %LOC( CURRENT_STRINGS( 2 ) )
      ADDRESSES( 3 ) = %LOC( CURRENT_STRINGS( 3 ) )
      ADDRESSES( 4 ) = %LOC( CURRENT_STRINGS( 4 ) )
      ADDRESSES( 5 ) = %LOC( CURRENT_STRINGS( 5 ) )
      ADDRESSES( 6 ) = %LOC( CURRENT_STRINGS( 6 ) )
      ADDRESSES( 7 ) = %LOC( CURRENT_STRINGS( 7 ) )
      ADDRESSES( 8 ) = %LOC( CURRENT_STRINGS( 8 ) )
      ADDRESSES( 9 ) = %LOC( CURRENT_STRINGS( 9 ) )
      ADDRESSES( 10 ) = %LOC( CURRENT_STRINGS( 10 ) )
C     Save the current choice input initialization values...
      RECORD_BUFFER_LENGTH = 12
      CALL GKS$INQ_CHOICE_STATE( WS_ID, DEVICE_NUM,
     * ERROR_STATUS, INPUT_MODE, ECHO_FLAG, INITIAL_STATUS,
     * INITIAL_CHOICE, PROMPT_ECHO_TYPE, ECHO_AREA,
     * CHOICE_DATA_RECORD, RECORD_BUFFER_LENGTH, RECORD_SIZE )

C     Obtain the default values and put them in temporary buffers.
      TEMP_DATA_RECORD( 1 ) = 2
      TEMP_DATA_RECORD( 2 ) = %LOC( TEMP_SIZES( 1 ) )
      TEMP_DATA_RECORD( 3 ) = %LOC( TEMP_ADDRESSES( 1 ) )
      TEMP_ADDRESSES( 1 ) = %LOC( TEMP_CURRENT_STRINGS( 1 ) )
      TEMP_ADDRESSES( 2 ) = %LOC( TEMP_CURRENT_STRINGS( 2 ) )

C     Inquire the default values...
      TEMP_RBL = 12
      CALL GKS$INQ_DEF_CHOICE_DATA( WS_TYPE, DEVICE_NUM,
     * ERROR_STATUS, DUMMY_INTEGER, DUMMY_INTEGER,
     * %DESCR( LIST_PROMPT_TYPES), TEMP_ECHO_AREA,
     * TEMP_DATA_RECORD, PROMPT_RETURN_SIZE, TEMP_RBL,
     * TEMP_RECORD_SIZE )
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Set temporary values...
      TEMP_INITIAL_CHOICE = 1
      TEMP_INITIAL_STATUS = GKS$K_STATUS_OK
      TEMP_SIZES( 1 ) = 8
      TEMP_SIZES( 2 ) = 4
      TEMP_ADDRESSES( 1 ) = %LOC( 'Continue' )
      TEMP_ADDRESSES( 2 ) = %LOC( 'Exit' )

C     Reinitialize the choice device...
      CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
     * TEMP_INITIAL_STATUS, TEMP_INITIAL_CHOICE,
     * PROMPT_ECHO_TYPE, TEMP_ECHO_AREA, TEMP_DATA_RECORD,
     * TEMP_RBL )

C     Check to see whether the picture on the screen is out of date.
      CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NEW_FRAME_FLAG )

C     Release deferred output. Regenerate if necessary.
      IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF

C     Ask the user to quit or to continue...
      CALL GKS$REQUEST_CHOICE( WS_ID, DEVICE_NUM, INPUT_STATUS,
     * CHOICE )

C     Reset the visibility of the segments.
      CALL GKS$SET_SEG_VISIBILITY( TITLE, GKS$K_VISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( STARS, GKS$K_VISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( TREE, GKS$K_VISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( SIDE, GKS$K_VISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( ROAD, GKS$K_VISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HOUSE, GKS$K_VISIBLE )
      CALL GKS$SET_SEG_VISIBILITY( HORIZON, GKS$K_VISIBLE )
      IF ( CHOICE .EQ. CONTINUE ) THEN
      CALL GKS$SET_SEG_VISIBILITY( HELP_BOX, GKS$K_VISIBLE )
      ENDIF
      CALL GKS$SET_SEG_VISIBILITY( HELP, GKS$K_INVISIBLE )

C     Check to see whether the picture on the screen is out of date.
      CALL GKS$INQ_WS_DEFER_AND_UPDATE( WS_ID, ERROR_STATUS,
     * DUMMY_INTEGER, DUMMY_INTEGER, DUMMY_INTEGER,
     * NEW_FRAME_FLAG )

C     Release deferred output. Regenerate if necessary.
      IF ( NEW_FRAME_FLAG .EQ. GKS$K_NEWFRAME_NECESSARY ) THEN
           CALL GKS$UPDATE_WS( WS_ID, GKS$K_PERFORM_FLAG )
      ENDIF
```

(continued on next page)

Example 7–1 (Cont.): Using the DEC GKS Asynchronous Input Functions

```
C     Set values depending on the user's choice.
      IF ( CHOICE .EQ. CONTINUE ) THEN
           FINISHED_FLAG = 0

C     Reset the choice device with its previous values...
           CALL GKS$INIT_CHOICE( WS_ID, DEVICE_NUM,
     *       INITIAL_STATUS, INITIAL_CHOICE, PROMPT_ECHO_TYPE,
     *       ECHO_AREA, CHOICE_DATA_RECORD, RECORD_BUFFER_LENGTH )

C     Reactivate the input devices...
           CALL GKS$SET_CHOICE_MODE( WS_ID, DEVICE_NUM,
     *       GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )
           CALL GKS$SET_PICK_MODE( WS_ID, DEVICE_NUM,
     *       GKS$K_INPUT_MODE_SAMPLE, GKS$K_ECHO )
           CALL GKS$SET_VALUATOR_MODE( WS_ID, DEVICE_NUM,
     *       GKS$K_INPUT_MODE_EVENT, GKS$K_ECHO )

      ELSE
           FINISHED_FLAG = 1
           CALL GKS$DELETE_SEG( HELP_BOX )
      ENDIF

      RETURN
      END
```

Appendix A
DEC GKS Glossary

This appendix is a list of definitions of some of the DEC GKS terminology used in the DEC GKS documentation.

active devices

Logical input devices whose prompts currently appear on the workstation surface. To deactivate a logical input device, you remove its prompt from the workstation surface by placing the device in request mode.

aperture

A pick logical input cursor that a user positions on a segment to be picked.

aspect ratio

A ratio of Y to X used to describe the shape of a rectangle in a particular coordinate system. The aspect ratio of the rectangular region affects the shapes of the output primitives contained in the region.

asynchronous input

A method of interaction between an application and its user. DEC GKS sample and event asynchronous input modes allow the user to enter input values while the application continues to execute. See also *synchronous input*.

attribute

A particular property that applies to an output primitive (such as character height) or to a segment (such as highlighting).

attribute source flag (ASF)

A setting that tells DEC GKS to use either individual or bundled attribute settings when generating an output primitive.

background color
The color of a blank workstation surface.

baseline
A horizontal line within a character body which, for many character definitions, has the appearance of being the lower limit of the character shape. A descender passes below this line. All baselines in a font are in the same position in the character bodies.

bound attributes
Attribute values that DEC GKS assigns to a primitive at the time of output generation. Bound attribute values cannot be changed.

break
A device-dependent method the user has of terminating the input process without changing the measure of a logical input device. See also *measure*.

bundle index
An integer value that specifies a single entry in a bundle table for a particular output primitive. A bundle table entry contains settings for each nongeometric attribute of an output primitive.

bundle table
A workstation-dependent table associated with a particular output primitive. Entries in the table specify all the workstation-dependent aspects of a primitive. Bundle tables exist for the output primitives polyline, polymarker, text and fill area. See also *attribute*.

caplines
A horizontal line within a character body, which, for many character definitions, has the appearance of being the upper limit of the character shape. An ascender may pass above this line and in some languages an additional mark (for example, an accent) over the character may be defined above this line. All caplines in a font are in the same position in the character bodies.

cell array
An output primitive consisting of a rectangular grid of equal size rectangular cells, each having a single color or shade.

centerline
A vertical line bisecting the character body.

character body

A rectangle used by a font designer to define a character shape. All character bodies in a font have the same height.

choice device

A logical input device providing a nonnegative integer defining one of a set of alternatives. An example of a choice prompt is a menu with a single highlighted choice.

clipping

Removing parts of output primitives that extend outside a window or viewport.

color table

A workstation-dependent table, in which the entries associate red, green and blue intensity values to color indexes. Color indexes are integer values that indicate individual color table entries.

current event report

The oldest event report, which is removed from the event input queue and placed into the DEC GKS state list by a call to GKS$AWAIT_EVENT (as long as the queue contains at least one report). See also *event input queue* and *report*.

cycling

A process by which a user activates the prompt of only one logical input device and deactivates all other prompts, activating each prompt in succession in some device-specific order. See also *active devices*.

deferred output

A process of delaying the transmission of output primitives to the surface of a workstation.

device coordinate system

A physical device's coordinate system used by a DEC GKS workstation. Device coordinate units are device dependent, and are expressed either in meters or some device-specific unit of measure. You use all or part of the device coordinate system to present graphical pictures.

device dependent

A property that is unique to a particular device (terminal, plotter, workstation, and so forth). For instance, the device coordinate system range is device dependent; its minimum and maximum X and Y values differ from device to device.

device independent

A property that remains consistent no matter which device you use (terminal, plotter, workstation, and so forth). For instance, text height, since it is expressed in world coordinates, is a device independent attribute; a single value can apply to any workstation you use.

display surface

See *workstation surface.*

dynamic changes

Attribute or transformation changes that DEC GKS can implement without having to redraw the entire picture on the workstation surface.

echo

Visual indication on the workstation surface of the current logical input device measure. See also *prompt and echo types.*

escape

A function used to access implementation or device-dependent features other than output generation (Generalized Drawing Primitives access device-dependent output features).

event mode

An asynchronous input operating mode that allows the user to trigger input devices, placing event reports on the event input queue. When the application chooses, it removes the oldest report from the queue, places it in the current event report, and processes the information. See also *asynchronous input* and *report.*

event input queue

A time-ordered queue on which a device handler places an input report, which the input process generates each time the user triggers a device in event mode. See also *report* and *trigger.*

event queue overflow

A condition that occurs when the queue cannot accept subsequently generated reports. In order to allow the user to generate further reports, you need to empty the event input queue. See also *report.*

fill area

An output primitive consisting of a polygon which may be hollow or may be filled with a uniform color, a pattern, or a hatch style.

fill area bundle table

A table associating specific values, for all fill area nongeometric attributes, with a fill area bundle index. This table contains entries consisting of interior style, style index, and color index. See also *index*.

foreground colors

Colors that a workstation uses to represent output primitives.

generalized drawing primitive (GDP)

An output primitive used to address special geometrical workstation capabilities, such as a curve drawing. For example, a workstation can support a GDP that draws circles.

geometric attributes

Primitive attributes that are device-dependent attributes. For instance, character height, character path, and pattern size are geometric attributes.

GKS level

A value from both the output levels (m, 0, 1, 2) and the input levels (a, b, c) that together define the minimal functional capabilities provided by a specific GKS implementation.

GKS Metafile (GKSM)

A standard metafile structure used by DEC GKS. See also *metafile*.

graphics handler

A device-dependent part of a DEC GKS implementation that supports a physical device. The DEC GKS kernel uses graphics handlers to perform the device-dependent tasks involved with output generation and input requests.

halfline

A horizontal line between the capline and the baseline within the character body, on which a horizontal string of characters in a font would appear centrally placed in the vertical direction. All halflines in a font are in the same position in the character bodies.

hatch

One possible method of filling the interior of a fill area primitive. DEC GKS fills the interior with an arrangement of one or more sets of parallel lines. When generating hatches, DEC GKS overlays the hatch so that portions of the underlying primitives are still visible.

highlighting

A device-independent way of emphasizing a segment by modifying its appearance on the workstation surface. For example, an implementation of GKS can highlight a segment by causing all segment primitives to blink on and off.

identity transformation

A default segment transformation (number 0) that makes no changes to the segment as stored on the NDC plane.

implicit regeneration

A process of clearing the workstation surface and redrawing only the stored segments. DEC GKS performs implicit regenerations if you request an attribute or transformation change that cannot be implemented dynamically. If an implicit regeneration occurs, you lose all primitives not stored in segments. See also *segments.*

index

An integer value that specifies a single entry in a bundle table. See also *bundle table.*

input class

A set of input devices that returns the same DEC GKS data type. The input classes are locator, stroke, valuator, choice, pick, and string.

interaction

A request for input from an application user.

inquiry function

A function that returns default or current values contained in DEC GKS data structures. Calls to these functions have no effect on the DEC GKS operating state or on the currently generated picture.

kernel

A part of a GKS implementation that performs device-independent tasks. To perform device-dependent tasks, the DEC GKS kernel calls a specified graphics handler.

locator device

A logical input device that accepts a device coordinate point and returns the corresponding world coordinate points. See also *world coordinate system.*

lock

A disabling of an active logical input device prompt so that its measure cannot be changed. See also *active devices, event mode,* and *measure.*

logical input device

An abstraction of one or more physical devices that delivers logical input values to the program. Logical input device classes are locator, stroke, valuator, choice, pick and string.

mapping

A process of transferring the contents of a window to the interior of a viewport. Mapping in a normalization transformation establishes a one-to-one correspondence between the points in both the window and the viewport. Mapping in a workstation transformation establishes a one-to-one correspondence between the points in the window, and the points in the section of the viewport that maintains the aspect ratio of the picture. See also *aspect ratio.*

marker

A symbol with a specified appearance that you use to identify a particular location.

measure

A current value of a logical input device.

metafile

An audit of a DEC GKS picture generation session. Metafiles are used for long-term graphical data storage. You can use a metafile to reproduce a picture generated by another application program.

MI

An abbreviation for the DEC GKS metafile input workstation category.

MO

An abbreviation for the DEC GKS metafile output workstation category.

nominal sizes

A default line width or marker size as determined by the graphics handler. DEC GKS adjusts these sizes by multiplying the nominal size times the current scale factor value.

nongeometric attributes

Primitive attributes that are device-independent. For instance, line type, marker size scale factor, and fill area interior style are nongeometric attributes.

normalization transformation

A process of mapping a window in world coordinate space into a viewport in normalized device coordinate space. You use normalization transformations to compose a device-independent picture.

normalized device coordinate (NDC) system

A device-independent coordinate system. You use the NDC coordinate system as a pasteboard on which to compose a complete graphical picture.

occludes

Overlaps.

operating modes

Synchronous and asynchronous methods of input. The three input operating modes are request, sample, and event mode.

operating state

An ability to access a given number of DEC GKS data structures depending on the previously called control functions. The current DEC GKS operating state determines which DEC GKS functions you can or cannot call at a given point in an application program.

output primitive

See *primitive.*

overflow

See *event queue overflow.*

pattern

A pattern is a cell array that alternates its cells in a sequence of colors or shades. Patterns always overwrite any underlying primitives.

physical device

A tool used to generate graphical output or accept graphical input. Terminals, plotters, printers, and workstations are examples of physical devices.

pick device

A logical input device that accepts a device coordinate point and returns the name of the segment that contains the picked primitive. DEC GKS segment names are integer values. See also *segment.*

pick identifier

An output attribute that allows you to name output primitives within a segment. At the time of primitive generation, DEC GKS assigns the current pick identifier integer value to the primitive. During pick input, the pick logical input device returns both the name of the segment that contains the picked primitives, and that primitive's pick identifier.

picture

A collection of output primitives or segments displayed at any one time on a workstation surface.

pixel

The smallest element of a display surface that can be independently assigned a color or intensity.

polyline

An output primitive consisting of a set of connected lines.

polyline bundle table

A table associating specific values, for all polyline nongeometric attributes, with a polyline bundle index. This table contains entries consisting of line type, line width scale factor, and polyline color index. See also *index.*

polymarker

An output primitive consisting of a set of locations indicated by a symbol.

polymarker bundle table

A table associating specific values, for all polymarker nongeometric attributes, with a polymarker bundle index. This table contains entries consisting of marker type, marker size scale factor, and polymarker color index. See also *index.*

primitive

An element that you use to construct a graphical picture. The output primitives are polyline, polymarker, text, fill area, cell array, and generalized drawing primitive.

primitive attribute

See *attribute.*

prompt

A visual indication on the workstation surface of the current value of a logical input device.

prompt and echo types

Different visual indications used by devices of a logical input class. For example, a locator logical input device can prompt a user by placing cross hairs, a tracking plus sign, or a cross on the workstation surface.

queue

See *event input queue.*

raster graphics

Computer graphics in which a display image is composed of an array of pixels arranged in rows and columns.

raster units

Number of pixel rows and columns on a physical device.

report

A data structure that the input process creates and places on the queue when a user triggers a device in event mode. The data structure contains a workstation identifier, a device number, an input class specification, a simultaneous event flag, and input data. See also *event mode, simultaneous event,* and *trigger.*

representation

The attribute or RGB settings for a single bundle table entry.

request mode

A synchronous input operating mode that allows the user to trigger a logical input device once, returning its current measure. See also *synchronous input* and *trigger.*

rotation

Turning all or part of a segment around a fixed point axis.

sample mode

An asynchronous input operating mode that allows the application program to read the current measure of a logical input device; the user can only control the current measure and cannot trigger the device. See also *asynchronous input* and *measure.*

scaling

Enlarging or reducing all or part of a segment toward or away from a fixed point axis.

segment

A collection of output primitives that can be manipulated as a unit.

segment attributes

Properties that apply to segments. Segment attributes are visibility, highlighting, detectability, segment priority, and segment transformation. See also *attributes*.

segment priority

A segment attribute used to determine which of several overlapping segments takes precedence.

segment transformation

A number specifying an associated matrix that expresses values for segment scaling, rotation, and translation.

simultaneous events

Several event reports generated by a trigger that affects several devices active at the same time. The device handler places the simultaneously generated events on the queue in some device-specific order. See also *event mode* and *report*.

string device

A logical input device that accepts and returns a character string.

stroke device

A logical input device that accepts a set of device coordinate points and returns the set of corresponding world coordinate points. See also *world coordinate system*.

synchronous input

A method of interaction between an application and its user. DEC GKS request mode causes the application to pause while the user alters the measure of a device and triggers input. After the user triggers the device, the device handler removes the device's prompt from the workstation surface and application execution resumes. See also *asynchronous input*.

text

An output primitive consisting of a character string.

text bundle table

A table associating specific values, for all text nongeometric attributes, with a text bundle index. This table contains entries consisting of text font and precision, character expansion factor, character spacing and text color index. See also *index*.

text font and precision

A text attribute having the components *font* and *precision*, which together determine the shape of the characters being output on a particular workstation. In addition, the precision describes the effectiveness of the other text attributes. In order of increasing permitted effectiveness, the precisions are string, character and stroke.

transformation

A mapping of primitives from one coordinate system's window to another coordinate system's viewport.

translation

Altering a segment's coordinate points so that the segment appears in a new position in the picture.

trigger

An indication from the user telling a logical input device to accept the current input value. An example of an input trigger is the pressing of the RETURN key.

unity transformation

A default normalization transformation (number 0) that uses the default normalization window and viewport.

update

To release deferred output and to implement all previously requested changes to the picture, if necessary.

valuator device

A logical input device that accepts and returns real values.

viewport

A defined rectangular area on a coordinate system into which DEC GKS maps primitives contained in a window.

window

A defined rectangular area on a coordinate system from which DEC GKS maps primitives to a viewport.

workstation

An abstract graphical device that includes a physical device and a software graphics handler that drives the physical device.

workstation dependent segment storage (WDSS)

Segment storage on a workstation other than a workstation of the category workstation independent segment storage. Segments cannot be transferred from WDSS to another workstation.

workstation handler

See *graphics handlers.*

workstation independent segment storage (WISS)

A special workstation type, where segments can be stored and later transferred to other workstations.

workstation surface

That portion of the device space corresponding to the area available for displaying images or for input working space (such as a digitizer).

workstation transformation

A transformation that maps the boundary and interior of a workstation window into the boundary and interior of a workstation viewport (part of display space), preserving aspect ratio. The effect of preserving aspect ratio is that the interior of the workstation window may not map to the whole of the workstation viewport. You use workstation transformations to control the amount of the picture shown on a given portion of a workstation surface.

world coordinate system

An imaginary device-independent Cartesian coordinate system that you use to plot output primitives. Once you plot a primitive in world coordinates, you must establish the proper normalization transformation, and then pass the world coordinate points to an output function.

Appendix B

Sample Programs

This appendix contains Example 3–2 written in all of the DEC GKS supported languages. Example 3–2 is the Starry Night program that is used as a base for every program example presented in this manual. For detailed information concerning this example, refer to Chapter 3, Writing Device-Independent Programs.

To use this program, you must have the appropriate definition file in the same directory as your source program. The definition file GKSDEFS.ext is located in the directory SYS$LIBRARY. For detailed information concerning the various definition files, refer to Chapter 1, Introduction to DEC GKS, in the *DEC GKS Reference Manual*.

B.1 FORTRAN Binding

Example B–1 presents the Starry Night program written in VAX FORTRAN using the DEC GKS FORTRAN binding. To link this program, you need to execute the following command on the DIGITAL Command Line:

```
$ LINK  object_module, SYS$LIBRARY:GKSFORBND/LIBRARY RETURN
```

Example B–1: FORTRAN Binding Sample Program

```
      INTEGER WKID, HOUSE, TREE, HORIZ, STARS, TITLE, SIDE, ROAD

      DATA WKID / 1 /, TITLE / 1 /, STARS / 2 /, TREE / 3 /,
     * SIDE / 4 /, ROAD / 5 /, HORIZ / 6 /, HOUSE / 7 /

      EXTERNAL SETUP, DEPICT, TIDYUP

      CALL SETUP( WKID )
      CALL DEPICT( WKID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZ )
      CALL TIDYUP( WKID )

      END

C     ************************************************************
C     Set up the DEC GKS and the workstation environments...
      SUBROUTINE SETUP( WKID )

      INCLUDE 'SYS$LIBRARY:GKSDEFS.BND'
      INTEGER WKID, ERRIND, WKCAT, OKAY, IDUMMY, DEFMOD, REGMOD

      CHARACTER*80  SDUMMY

      DATA OKAY / 0 /

      OPEN ( UNIT=1, FILE='SYS$ERROR:', STATUS='NEW' )
      CALL GOPKS( 1 )

      CALL GQWKCA( GWSDEF, ERRIND, WKCAT )

C     Make sure that the workstation type is valid.
      IF (( ERRIND .NE. OKAY ) .OR.
     *   (( WKCAT .NE. GOUTPT ) .AND. (  WKCAT .NE. GOUTIN ))) THEN
             WRITE(6,*)
     *       'The specified workstation type is invalid.'
             WRITE(6,*) 'Error status:', ERRIND
             STOP
      ENDIF

      CALL GOPWK( WKID, GWCONID, GWSDEF )
      CALL GACWK( WKID )

C     Make sure that the deferral mode and regeneration flag are
C     properly set.
      CALL GQWKC( WKID, ERRIND, SDUMMY, WTYPE )


      CALL GQDDS( WTYPE, ERRIND, DEFMOD, REGMOD )

C     You can check the status of the inquiry function execution, as
C     follows:
      IF ( ERRIND .NE. OKAY ) THEN
             WRITE(6,*) 'The deferral inquiry caused an error.'
             WRITE(6,*) 'Error status:', ERRIND
             STOP
      ENDIF
```

(continued on next page)

```
C     Defer output as long as possible and suppress implicit
C     regenerations.
      IF (( DEFMOD .NE. GASTI ) .AND. ( REGMOD .NE. GSUPPD )) THEN
            CALL GSDS( WKID, GASTI, GSUPPD )
      ENDIF

      RETURN
      END

C     ************************************************************
C     Draw the picture, and place each primitive in a segment...
      SUBROUTINE DEPICT( WKID, TITLE, STARS, TREE, SIDE,
     * ROAD, HOUSE, HORIZ )

      INCLUDE 'SYS$LIBRARY:GKSDEFS.BND'

      INTEGER WKID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE,
     * HORIZ, NSTARS, NTREE, NHOUSE, NLAND, SOFFC,
     * SOFFR, SNUMC, SNUMR, SCOLOR( 1, 2 ), ROFFC, ROFFR,
     * RNUMC, RNUMR, RCOLOR( 10, 1 ),
     * SDIMX, SDIMY, LIGHT, DARK, ERRIND, IDUMMY, WTYPE,
     * COLA, NPCI, THREE, NBW, RDIMX, RDIMY, ELEMNT

      CHARACTER*80 SDUMMY

      REAL TEXTX, TEXTY, STARSX( 6 ),
     * STARSY( 6 ), TREEX( 29 ), TREEY( 29 ),
     * HOUSEX( 12 ), HOUSEY( 12 ), LANDX( 15 ),
     * LANDY( 15 ), SSTRTX, SSTRTY, SDIAGX,
     * SDIAGY, RSTRTX, RSTRTY, RDIAGX,
     * RDIAGY, LARGER, WIDER, RLWMAX, RDUMMY,
     * NOMLW, BWX( 9 ), BWY( 9 )

      DATA TEXTX / 0.05 /,
     * TEXTY / 0.9 /, NSTARS / 6 /, NTREE / 29 /, NHOUSE / 12 /,
     * NLAND / 15 /, SSTRTX / 0.2 /, SSTRTY / 0.3 /,
     * SDIAGX / 0.25 /, SDIAGY / 0.15 /,
     * SDIMX / 1 /, SDIMY / 2 /, SOFFC / 1 /, SOFFR / 1 /,
     * SNUMC / 1 /, SNUMR / 2 /, RSTRTX/ 0.0 /,
     * RSTRTY / 0.15 /, RDIAGX / 1.0 /, RDIAGY / 0.0 /,
     * RDIMX / 10 /, RDIMY / 1 /, ROFFC / 1 /, ROFFR / 1 /,
     * RNUMC / 10 /, RNUMR / 1 /, LIGHT / 2 /, DARK / 3 /,
     * LARGER / 0.04 /, WIDER / 3.0 /, THREE / 3 /, NBW / 9 /,
     * ELEMNT / 1 /

      DATA BWX / 0.0, 0.0, 0.2, 0.2, 0.25, 0.25, 1.0, 1.0, 0.0 /
      DATA BWY / 0.0, 0.15, 0.15, 0.3, 0.3, 0.15, 0.15, 0.0, 0.0 /

      DATA SCOLOR / 2, 3 /
      DATA RCOLOR / 2, 3, 2, 3, 2, 3, 2, 3, 2, 3 /
```

(continued on next page)

```
      DATA STARSX / 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 /
      DATA STARSY / 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 /

      DATA TREEX / 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
     * 0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
     * 0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
     * 0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
     * 0.515, 0.51, 0.495, 0.475, 0.425 /
      DATA TREEY / 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
     * 0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
     * 0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
     * 0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
     * 0.5, 0.425, 0.38, 0.33, 0.28 /

      DATA HOUSEX / 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
     * 0.25, 0.25, 0.2, 0.075, 0.1, 0.1 /
      DATA HOUSEY / 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
     * 0.75, 0.7, 0.75, 0.6, 0.6, 0.3 /

      DATA LANDX / 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
     * 0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 /
      DATA LANDY / 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
     * 0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
     * 0.385 /


      CALL GSCHH( LARGER )
      CALL GSMK( GPLUS )
      CALL GSFAIS( GSOLID )
      CALL GSLN( GLDASD )

C     Obtain the workstation type.
      CALL GQWKC( WKID, ERRIND, SDUMMY, WTYPE )

C     Make sure that you don't ask for a line wider than the
C     workstation's widest line.
      CALL GQPLF( WTYPE, ELEMNT, ERRIND, IDUMMY, IDUMMY, IDUMMY,
     *            NOMLW, RDUMMY, RLWMAX, IDUMMY )

      DO WHILE (( WIDER * NOMLW ) .GT. RLWMAX )
         WIDER = WIDER - 0.1
      ENDDO

      CALL GSLWSC( WIDER )
      CALL GCRSG( TITLE )
      CALL GTX( TEXTX, TEXTY, 'Starry Night' )
      CALL GCLSG()

      CALL GCRSG( STARS )
      CALL GPM( NSTARS, STARSX, STARSY )
      CALL GCLSG()
```

(continued on next page)

```
      CALL GCRSG( TREE )
      CALL GFA( NTREE, TREEX, TREEY )
      CALL GCLSG()

C     Check to see if you are working with a color workstation.
      CALL GQCF( WTYPE, ERRIND, IDUMMY, COLA, NPCI )

C     For all workstations with only 2 color indexes, use a fill area
C     instead of a cell array for the sidewalk and road.
      IF ( NPCI .LT. THREE ) THEN
           CALL GCRSG( SIDE )
           CALL GSFAIS( GHATCH )
           CALL GFA( NBW, BWX, BWY )
           CALL GSFAIS( GSOLID )
           CALL GCLSG()
      ELSE
           CALL GCRSG( SIDE )
           CALL GCA( SSTRTX, SSTRTY, SDIAGX, SDIAGY, SDIMX, SDIMY,
     *               SOFFC, SOFFR, SNUMC, SNUMR, SCOLOR )
           CALL GCLSG()
           CALL GCRSG( ROAD )
           CALL GCA( RSTRTX, RSTRTY, RDIAGX, RDIAGY, RDIMX, RDIMY,
     *               ROFFC, ROFFR, RNUMC, RNUMR, RCOLOR )
           CALL GCLSG()
      ENDIF

      CALL GCRSG( HORIZ )
      CALL GPL( NLAND, LANDX, LANDY )
      CALL GCLSG()

      CALL GCRSG( HOUSE )
C     Only change the color index if working with a workstation
C     with more than two color indexes.
      IF ( NPCI .GE. THREE ) THEN
           CALL GSFACI( DARK )
      ENDIF
      CALL GFA( NHOUSE, HOUSEX, HOUSEY )
      CALL GCLSG()

      RETURN
      END
```

(continued on next page)

Example B–1 (Cont.): FORTRAN Binding Sample Program

```
C       ************************************************************
C       Clean up the DEC GKS and the workstation environments...
        SUBROUTINE TIDYUP( WKID )

        INTEGER WKID

        CALL GUWK( WKID, GPERFO )
        READ(5,*)

        CALL GDAWK( WKID )
        CALL GCLWK( WKID )
        CALL GCLKS()

        RETURN
        END
```

B.2 C Binding

Example B–2 presents the Starry Night program written in the GKS C Binding.

Example B–2: C Binding Sample Program

```
#include    "gks.h"
#include    <stdio.h>

#define     MAX_STRING     80
#define     MAX_INT     50
#define     NUM_SIDE_COLORS 2
#define     NUM_ROW_COLORS  10
#define     NUM_STARS     6
#define     NUM_TREE     29
#define     NUM_HOUSE     12
#define     NUM_LAND     15
#define     NUM_BW     9


main ()
{
    Gint    ws_id,
            house,
            tree,
            horizon,
            stars,
            title,
            side,
            road;

    /* Initialize variables */
    ws_id = 1;
    title = 1;
    stars = 2;
    tree = 3;
    side = 4;
    road = 5;
    horizon = 6;
    house = 7;


    set_up (ws_id);

    draw_picture (ws_id, title, stars, tree, side, road, house, horizon);

    clean_up (ws_id);

} /* end main */
```

(continued on next page)

Example B–2 (Cont.): C Binding Sample Program

```
/*********************************************************************/
/*                                                                   */
/*  Set up the DEC GKS and the workstation environments...           */
/*                                                                   */
/*********************************************************************/
set_up (ws_id)
Gint ws_id;
{
    Gint    error_status = 0,
            ws_size = 0,
            bufsize = 0,
            fac_size = 0,
            inquire_okay = 0,
            index = 0,
            regen_flag = 0;

    Gint    ws_type = 0;
    Gwsct   ct;
    Gwscat  category;
    Gdefer  def;
    Gdefmode defmode;
    Girgmode regen_mode;
    Gwstype  type = 0;
    Gwsclass class;
    Glnbundl rep;
    Glnfac   fac;
    Gcofac   col_fac;
    Gconn    conid = 0;

    /* Initialize variables */
    inquire_okay = 0;
    def.defmode = GASAP;
    def.irgmode = GALLOWED;


    gopengks(stderr, GDEFAULT_MEM_SIZE);
    ginqwscategory(&type, &category, &error_status);

    /* Make sure that the workstation type is valid. */
    if ((error_status != inquire_okay) ||
 ((category != GOUTIN) &&
 (category != GMO))) {
 printf ("The specified workstation type is invalid\n");
 printf ("Error status: %d\n", error_status);
 return;
    }
    gopenws(ws_id, &conid, &type);
    gactivatews(ws_id);

    /* Make sure the deferral mode and regeneration flag are properly set */
    ginqwsconntype (ws_id, bufsize, &ws_size, &ct, &error_status);

    ginqdefdeferst (&ws_type, &def, &error_status);
```

(continued on next page)

```
    /* Check the status of the inquiry function execution */
    if (error_status != inquire_okay) {
 printf ("The deferral inquiry caused an error\n");
 printf ("Error status: %d\n", error_status);
 return;
    }

    /* Defer output as long as possible and suppress implicit regenerations */
    if ((def.defmode != GASTI) && (def.irgmode != GSUPPRESSED))
 gsetdeferst (ws_id, defmode, regen_mode);

} /* end set_up */


/*********************************************************************/
/*                                                                   */
/*  Draw the picture, and place each primitive in a segment...      */
/*                                                                   */
/*********************************************************************/
draw_picture (ws_id, title, stars, tree, side, road, house, horizon)
Gint ws_id,
     title,
     stars,
     tree,
     side,
     road,
     house,
     horizon;
{
     Gint   num_stars = 6,
            num_tree_pts = 29,
            num_house_pts = 12,
            num_land_pts = 15,
            side_colors [NUM_SIDE_COLORS] = {2, 3},
            road_colors [NUM_ROW_COLORS] = {2, 3, 2, 3, 2, 3, 2, 3, 2, 3},
            light,
            dark,
            error_status = 0,
            ws_type,
            color_flag,
            num_indexes,
            line_type,
            bufsize = 0,
            fac_size = 0,
            index = 2,
            three,
            bw_num_pts;
    Gwstype type = 0;
    Gwsclass class;
    Glnbundl rep;
    Glnfac   fac;
    Gcofac   col_fac;
```

(continued on next page)

```
Gpoint text_start,
       stars_values[NUM_STARS] =
       { {0.05,0.7},{0.06,0.86},{0.36,0.81},{0.66,0.86},{0.835,0.701},
         {0.92,0.82} },
       bw_values[NUM_BW] =
       { {0.0,0.0},{0.0,0.15},{0.2,0.15},{0.2,0.3},{0.25,0.3},{0.25,0.15},
         {1.0,0.15},{1.0,0.0},{0.0,0.0} },
       house_values[NUM_HOUSE] =
       { {0.1,0.3},{0.3,0.3},{0.3,0.6},{0.325,0.6},{0.3,0.64},{0.3,0.75},
         {0.25,0.75},{0.25,0.7},{0.2,0.75},{0.075,0.6},{0.1,0.6},
         {0.1,0.3} },
       land_values[NUM_LAND] =
       { {0.0,0.35},{0.04,0.375},{0.055,0.376},{0.08,0.36},{0.1,0.365},
         {0.3,0.366},{0.375,0.38},{0.44,0.385},{0.49,0.375},{0.56,0.36},
         {0.68,0.38},{0.8,0.35},{0.9,0.359},{0.95,0.375},{1.0,0.385} },
       tree_values[NUM_TREE] =
       { {0.425,0.28},{0.5,0.3},{0.52,0.26},{0.54,0.3},{0.6,0.28},
         {0.575,0.33},{0.56,0.42},{0.559,0.49},{0.64,0.53},{0.69,0.57},
         {0.689,0.61},{0.66,0.64},{0.63,0.66},{0.645,0.71},{0.59,0.76},
         {0.53,0.78},{0.48,0.75},{0.45,0.71},{0.42,0.65},{0.375,0.645},
         {0.35,0.6},{0.375,0.55},{0.44,0.54},{0.45,0.5},{0.515,0.5},
         {0.51,0.425},{0.495,0.38},{0.475,0.33},{0.425,0.28} };

Gfloat  larger,
        wider;

Grect   side_rectangle_coordinates, road_rectangle_coordinates;
Gidim   side_rectangle_dim, road_rectangle_dim;
Gchar   text_str;
```

(continued on next page)

```
/* Initialize variables */
text_start.x = 0.05;
text_start.y = 0.9;
side_rectangle_coordinates.ul.x = 0.2;
side_rectangle_coordinates.ul.y = 0.3;
side_rectangle_coordinates.lr.x = 0.25;
side_rectangle_coordinates.lr.y = 0.15;
side_rectangle_dim.x_dim = 1;
side_rectangle_dim.y_dim = 2;
road_rectangle_coordinates.ul.x= 0.0;
road_rectangle_coordinates.ul.y= 0.15;
road_rectangle_coordinates.lr.x= 1.0;
road_rectangle_coordinates.lr.y= 0.0;
road_rectangle_dim.x_dim= 10;
road_rectangle_dim.y_dim = 0.0;
light = 2;
dark = 3;
larger = 0.04;
wider = 3.0;
three = 3;
bw_num_pts = 9;
line_type = 1;
fac.nom_width = 0.0;
fac.max_width = 0.0;

gsetcharheight (larger);
gsetmarkertype (GMK_PLUS);
gsetfillintstyle(GSOLID);
gsetlinetype (GLN_DASHED, GLN_DOTTED);

/* Obtain the workstation type. */
ginqwsclass ( &type, &class, &error_status );

/* Make sure that you don't ask for a line wider than the */
/* workstation's widest line.        */
ginqpatfacil (&type, bufsize, &fac_size, &fac, &error_status);
while (wider * fac.nom_width > fac.max_width)
    wider -= 0.1;
gsetlinewidth(wider);

gcreateseg (title);
text_str = "STARRY NIGHT";
gtext(&text_start, "STARRY NIGHT");
gcloseseg();

gcreateseg(stars);
gpolymarker(num_stars,stars_values);
gcloseseg();

gcreateseg(tree);
gfillarea(num_tree_pts, tree_values);
gcloseseg();
```

(continued on next page)

```
    /* CHECK TO SEE IF YOU ARE WORKING WITH A COLOR WORKSTATION */
    ginqcolourfacil ( &type, bufsize, &fac_size, &col_fac, &error_status);

    /* For all monochrome workstations (not including the VT125/240 or */
    /* the monochrome VAXStations), use GKS$FILL_AREA instead of       */
    /* GKS$CELL_ARRAY for the sidewalk and road.        */
    if (col_fac.coavail != GCOLOUR) {
        gcreateseg(side);
        gsetfillintstyle (GHATCH);
        gfillarea(bw_num_pts, bw_values);
        gsetfillintstyle(GSOLID);
        gcloseseg();
    } else {
        gcreateseg(side);
        gcellarray (&side_rectangle_coordinates, &side_rectangle_dim,
                    &side_colors);
        gcloseseg();
        gcreateseg(road);
        gcellarray (&road_rectangle_coordinates, &road_rectangle_dim,
                    &road_colors);
        gcloseseg();
    }

    gcreateseg(horizon);
    gpolyline (num_land_pts, land_values);
    gcloseseg();

    gcreateseg(house);

    /* Only change the color index if working with a color workstation */
    /* (or a VT125/240 or a VAXstation).    */
    if (col_fac.coavail == GCOLOUR)
        gsetfillcolourind(dark);

    gfillarea(num_house_pts, house_values);
    gcloseseg();

} /* end draw_picture */

/*********************************************************************/
/*                                                                   */
/*  Clean up the DEC GKS and the workstation environments...         */
/*                                                                   */
/*********************************************************************/
clean_up (ws_id)
Gint ws_id;
{
      gupdatews(ws_id, GPERFORM);
      getchar ();
```

(continued on next page)

Example B–2 (Cont.): C Binding Sample Program

```
     gdeactivatews(ws_id);
     gclosews(ws_id);
     gclosegks();

} /* end clean_up */
```

B.3 VAX C

Example B–3 presents the Starry Night program written in VAX C.

Example B–3: VAX C Sample Program

```
#include    <gksdefs.h>
#include    <descrip.h>
#include    <stdio.h>

#define     MAX_STRING      80
#define     MAX_INT         50
#define     NUM_SIDE_COLORS 2
#define     NUM_ROW_COLORS  10
#define     NUM_STARS       6
#define     NUM_TREE        29
#define     NUM_HOUSE       12
#define     NUM_LAND        15
#define     NUM_BW          9

struct  dsc$descriptor_a2 {
    unsigned short      dsc$w_length;   /* length of an array element in bytes,
                                         or if dsc$b_dtype is DSC$K_DTYPE_V, bits,
                                         or if dsc$b_dtype is DSC$K_DTYPE_P, digits (4 bits
                                         each) */
    unsigned char       dsc$b_dtype;    /* data type code */
    unsigned char       dsc$b_class;    /* descriptor class code = DSC$K_CLASS_A */
    char                *dsc$a_pointer; /* address of first actual byte of data storage */
    char                dsc$b_scale;    /* scale multiplier to convert from internal to external
                                         form */
    unsigned char       dsc$b_digits;   /* number of decimal digits in internal representation */
    struct      {
        unsigned                  : 4;  /* reserved, must be zero */
        unsigned dsc$v_fl_redim   : 1;  /* if set, indicates the array can be redimensioned */
        unsigned dsc$v_fl_column  : 1;  /* if set, indicates column-major order (FORTRAN) */
        unsigned dsc$v_fl_coeff   : 1;  /* if set, indicates the multipliers block is present */
        unsigned dsc$v_fl_bounds  : 1;  /* if set, indicates the bounds block is present */
    } dsc$b_aflags;
    unsigned char       dsc$b_dimct;    /* number of dimensions */
    unsigned long       dsc$l_arsize;   /* total size of array in bytes,
                                         or if dsc$b_dtype is DSC$K_DTYPE_P, digits (4 bits each) */
    char        *dsc$a_a0;              /* add of ele w/ zero coefficients */
    long        dsc$l_m [ 2 ];          /* Addressing coefficients (multipliers) */
    struct {
        long    dsc$l_l;                /* Lower bound */
        long    dsc$l_u;                /* Upper bound */
    } dsc$bounds [2];
};

#define DESC_ARRAY(name, length, ptr) struct dsc$descriptor_a          \
    name = {4, DSC$K_DTYPE_L, DSC$K_CLASS_A, ptr, 0, 0, {0, 0, 0, 0, 0},\
           1, length * 4}

#define DESC_ARRAY_2(name, dim1, dim2, ptr) struct dsc$descriptor_a2    \
    name = {4, DSC$K_DTYPE_L, DSC$K_CLASS_A, ptr, 0, 0, {0, 0, 0, 1, 1},\
          2, dim1 * dim2 * 4, ptr, {dim1, dim2}, {0, dim1 - 1, 0, dim2 - 1}}
```

(continued on next page)

Example B–3 (Cont.): VAX C Sample Program

```
main ()
{
    int     ws_id,
            house,
            tree,
            horizon,
            stars,
            title,
            side,
            road;

    /* Initialize variables */
    ws_id = 1;
    title = 1;
    stars = 2;
    tree = 3;
    side = 4;
    road = 5;
    horizon = 6;
    house = 7;

    set_up (ws_id);

    draw_picture (ws_id, title, stars, tree, side, road, house, horizon);

    clean_up (ws_id);

} /* end main */

/**********************************************************************/
/*                                                                    */
/*  Set up the DEC GKS and the workstation environments...            */
/*                                                                    */
/**********************************************************************/
set_up (ws_id)
int ws_id;
{
    int     error_status,
            category,
            inquire_okay,
            dummy_integer,
            def_mode,
            regen_flag,
            ws_type;

    struct  dsc$descriptor    dummy_dsc;
    char    dummy_string [MAX_STRING];

    $DESCRIPTOR( error_file, "sys$error:");
```

(continued on next page)

```
    /* Initialize variables */
    inquire_okay = 0;
    dummy_dsc.dsc$a_pointer = dummy_string;
    dummy_dsc.dsc$w_length = (short) MAX_STRING;

    gks$open_gks (&error_file);

    gks$inq_ws_category (&GKS$K_WSTYPE_DEFAULT, &error_status, &category);

    /* Make sure that the workstation type is valid. */
    if ((error_status != inquire_okay) ||
        ((category != GKS$K_WSCAT_OUTIN) &&
        (category != GKS$K_WSCAT_MO))) {
        printf ("The specified workstation type is invalid\n");
        printf ("Error status: %d\n", error_status);
        return;
    }

    gks$open_ws (&ws_id, &GKS$K_CONID_DEFAULT, &GKS$K_WSTYPE_DEFAULT);
    gks$activate_ws (&ws_id);

    /* Make sure the deferral mode and regeneration flag are properly set */
    gks$inq_ws_type (&ws_id, &error_status, &dummy_dsc, &ws_type,
                    &dummy_integer);

    gks$inq_def_defer_state (&ws_type, &error_status, &def_mode,
                             &regen_flag);

    /* Check the status of the inquiry function execution */
    if (error_status != inquire_okay) {
        printf ("The deferral inquiry caused an error\n");
        printf ("Error status: %d\n", error_status);
        return;
    }

    /* Defer output as long as possible and suppress implicit regenerations */
    if ((def_mode != GKS$K_ASTI) && (regen_flag != GKS$K_IRG_SUPPRESSED))
        gks$set_defer_state (&ws_id, &GKS$K_ASTI, &GKS$K_IRG_SUPPRESSED);

} /* end set_up */
```

(continued on next page)

```
/*********************************************************************/
/*                                                                   */
/*  Draw the picture, and place each primitive in a segment...       */
/*                                                                   */
/*********************************************************************/
draw_picture (ws_id, title, stars, tree, side, road, house, horizon)
int ws_id,
    title,
    stars,
    tree,
    side,
    road,
    house,
    horizon;
{

    int     num_stars,
            num_tree_pts,
            num_house_pts,
            num_land_pts,
            side_off_col,
            side_off_row,
            side_num_col,
            side_num_row,
            side_colors [NUM_SIDE_COLORS] = {2, 3},
            road_off_col,
            road_off_row,
            road_num_col,
            road_num_row,
            road_colors [NUM_ROW_COLORS] = {2, 3, 2, 3, 2, 3, 2, 3, 2, 3},
            light,
            dark,
            error_status,
            dummy_integer,
            ws_type,
            dummy_int_array [MAX_INT],
            color_flag,
            num_indexes,
            three,
            bw_num_pts;
```

(continued on next page)

Example B–3 (Cont.): VAX C Sample Program

```
float   text_start_x,
        text_start_y,
        stars_x_values [NUM_STARS] = {0.05, 0.06, 0.36, 0.66, 0.835, 0.92},
        stars_y_values [NUM_STARS] = {0.7, 0.86, 0.81, 0.86, 0.701, 0.82},

        tree_x [NUM_TREE] = {0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
                            0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
                            0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
                            0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
                            0.515, 0.51, 0.495, 0.475, 0.425},
        tree_y [NUM_TREE] = {0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
                            0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
                            0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
                            0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
                            0.5, 0.425, 0.38, 0.33, 0.28},
        house_x [NUM_HOUSE] = {0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
                               0.25, 0.25, 0.2, 0.075, 0.1, 0.1},
        house_y [NUM_HOUSE] = {0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
                               0.75, 0.7, 0.75, 0.6, 0.6, 0.3},
        land_x [NUM_LAND] = {0.0, 0.04, 0.055, 0.08, 0.1, 0.3, 0.375,
                             0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95,
                             1.0},
        land_y [NUM_LAND] = {0.35, 0.375, 0.376, 0.36, 0.365, 0.366, 0.38,
                             0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
                             0.385},
        side_start_x,
        side_start_y,
        side_diag_x,
        side_diag_y,
        road_start_x,
        road_start_y,
        road_diag_x,
        road_diag_y,
        larger,
        wider,
        max_width,
        dummy_real,
        nom_width,
        bw_x_values [NUM_BW] = {0.0, 0.0, 0.2, 0.2, 0.25, 0.25,
                                1.0, 1.0, 0.0 },
        bw_y_values [NUM_BW] = {0.0, 0.15, 0.15, 0.3, 0.3, 0.15,
                                0.15, 0.0, 0.0 };
```

(continued on next page)

Example B–3 (Cont.): VAX C Sample Program

```
    DESC_ARRAY( integer_dsc, MAX_INT, dummy_int_array);
    DESC_ARRAY_2( side_integer_dsc2, 1, 2, side_colors);
    DESC_ARRAY_2( road_integer_dsc2, 10, 1, road_colors);
    $DESCRIPTOR(title_dsc, "Starry Night");
    struct  dsc$descriptor    dummy_dsc;
    char    dummy_string [MAX_STRING];

    /* Initialize variables */
    text_start_x = 0.05;
    text_start_y = 0.9;
    num_stars = 6;
    num_tree_pts = 29;
    num_house_pts = 12;
    num_land_pts = 15;
    side_start_x = 0.2;
    side_start_y = 0.3;
    side_diag_x = 0.25;
    side_diag_y = 0.15;
    side_off_col = 0;
    side_off_row = 0;
    side_num_col = 1;
    side_num_row = 2;
    road_start_x = 0.0;
    road_start_y = 0.15;
    road_diag_x = 1.0;
    road_diag_y = 0.0;
    road_off_col = 0;
    road_off_row = 0;
    road_num_col = 10;
    road_num_row = 1;
    light = 2;
    dark = 3;
    larger = 0.04;
    wider = 3.0;
    three = 3;
    bw_num_pts = 9;
    dummy_dsc.dsc$a_pointer = dummy_string;
    dummy_dsc.dsc$w_length = (short) MAX_STRING;

    gks$set_text_height (&larger);
    gks$set_pmark_type (&GKS$K_MARKERTYPE_PLUS);
    gks$set_fill_int_style (&GKS$K_INTSTYLE_SOLID);
    gks$set_pline_linetype (&GKS$K_LINETYPE_DASHED_DOTTED);
```

(continued on next page)

Example B–3 (Cont.): VAX C Sample Program

```
/* Obtain the workstation type. */
gks$inq_ws_type ( &ws_id, &error_status, &dummy_dsc, &ws_type,
                  &dummy_integer);

/* Make sure that you don't ask for a line wider than the */
/* workstation's widest line.                             */
gks$inq_pline_fac (&ws_type, &error_status, &dummy_integer,
        &integer_dsc, &dummy_integer, &nom_width,
        &dummy_real, &max_width, &dummy_integer, &dummy_integer);

while (wider * nom_width > max_width)
    wider -= 0.1;

gks$set_pline_linewidth (&wider);

gks$create_seg (&title);
gks$text (&text_start_x, &text_start_y, &title_dsc);
gks$close_seg ();

gks$create_seg (&stars);
gks$polymarker (&num_stars, &stars_x_values, &stars_y_values);
gks$close_seg ();

gks$create_seg (&tree);
gks$fill_area (&num_tree_pts, &tree_x, &tree_y);
gks$close_seg ();

/* CHECK TO SEE IF YOU ARE WORKING WITH A COLOR WORKSTATION */
gks$inq_color_fac ( &ws_type, &error_status, &dummy_integer, &color_flag,
                    &num_indexes);

/*  For all workstations that have less than 2 color indexes,      */
/*  use GKS$FILL_AREA instead of GKS$CELL_ARRAY for the            */
/*  sidewalk and road.                                             */
if (num_indexes < three ) {
    gks$create_seg (&side);
    gks$set_fill_int_style (&GKS$K_INTSTYLE_HATCH);
    gks$fill_area (&bw_num_pts, &bw_x_values, &bw_y_values);
    gks$set_fill_int_style (&GKS$K_INTSTYLE_SOLID);
    gks$close_seg ();
} else {
    gks$create_seg (&side);
    gks$cell_array (&side_start_x, &side_start_y,
                    &side_diag_x, &side_diag_y, &side_off_col,
                    &side_off_row, &side_num_col, &side_num_row,
                    &side_integer_dsc2);
```

(continued on next page)

```
        gks$close_seg ();
        gks$create_seg (&road);
        gks$cell_array (&road_start_x, &road_start_y,
                        &road_diag_x, &road_diag_y, &road_off_col,
                        &road_off_row, &road_num_col, &road_num_row,
                        &road_integer_dsc2);
        gks$close_seg();
    }

    gks$create_seg (&horizon);
    gks$polyline (&num_land_pts, &land_x, &land_y);
    gks$close_seg ();

    gks$create_seg (&house);

    /* Only change the color index if working with a workstation       */
    /* that has three or more color indexes                            */
    if (num_indexes >= three)
        gks$set_fill_color_index (&dark);

    gks$fill_area (&num_house_pts, &house_x, &house_y);
    gks$close_seg ();

} /* end draw_picture */

/*******************************************************************/
/*                                                                 */
/*  Clean up the DEC GKS and the workstation environments...       */
/*                                                                 */
/*******************************************************************/
clean_up (ws_id)
int ws_id;
{
      gks$update_ws (&ws_id, &GKS$K_PERFORM_FLAG);
      getchar ();

      gks$deactivate_ws (&ws_id);
      gks$close_ws (&ws_id);
      gks$close_gks ();

} /* end clean_up */
```

B.4 VAX Pascal

Example B-4 presents the Starry Night program written in VAX Pascal. To compile this program, you need to copy SYS$LIBRARY:GKSDEFS.PAS to your local directory and then execute the following command on the DIGITAL Command Line:

```
$ PASCAL/ENVIRONMENT  GKSDEFS.PAS RETURN
```

Example B–4: VAX Pascal Sample Program

```
[INHERIT ('GKSDEFS')]
PROGRAM STARRY_NIGHT( INPUT, OUTPUT );

CONST

    WS_ID   = 1;
    TITLE   = 1;
    STARS   = 2;
    TREE    = 3;
    SIDE    = 4;
    ROAD    = 5;
    HORIZON = 6;
    HOUSE   = 7;

VAR

    SETUP_OK : BOOLEAN;

{*****************************************************************}
      {Set up the DEC GKS and the workstation environments...}
FUNCTION SET_UP( WS_ID : INTEGER ) : BOOLEAN;

    CONST

        INQUIRY_OKAY = 0;

    VAR

        WS_TYPE : INTEGER;
        ERROR_STATUS, CATEGORY : INTEGER;
        DUMMY_INTEGER, DEF_MODE, REGEN_FLAG : INTEGER;
        DUMMY_STRING : VARYING[ 80 ] OF CHAR;

    BEGIN  { Function SET_UP }

        { Initialize the successful setup flag as true }
        SET_UP := TRUE;

        GKS$OPEN_GKS( 'SYS$ERROR:' );

        GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT, ERROR_STATUS, CATEGORY );

        { Make sure that the workstation type is valid. }
        IF (( ERROR_STATUS <> INQUIRY_OKAY ) OR
           (( CATEGORY <> GKS$K_WSCAT_OUTPUT ) AND
           ( CATEGORY <> GKS$K_WSCAT_OUTIN ))) THEN
            BEGIN
                 { Setup was not completed properly - invalid
                   workstation type. }
                 WRITELN( 'The specified workstation type is invalid.' );
                 WRITELN( 'Error status:', ERROR_STATUS );
                 SET_UP := FALSE;
            END
```

(continued on next page)

```
        { If workstation is valid, continue to set up workstation. }
        ELSE
            BEGIN
                GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_WSTYPE_DEFAULT );
                GKS$ACTIVATE_WS( WS_ID );

                { Make sure that the deferral mode and regeneration flag are
                  properly set. }
                GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
                    WS_TYPE, DUMMY_INTEGER );

                GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,
                    DEF_MODE, REGEN_FLAG );

                { You can check the status of the inquiry function execution,
                  as follows: }
                IF ( ERROR_STATUS <> INQUIRY_OKAY ) THEN
                    BEGIN
                        { Setup was not completed properly - invalid
                          deferral inquiry. }
                        WRITELN( 'The deferral inquiry caused an error.' );
                        WRITELN( 'Error status:', ERROR_STATUS );
                        SET_UP := FALSE;
                    END

                ELSE
                    { Defer output as long as possible and suppress implicit
                      regenerations. }
                    IF (( DEF_MODE <> GKS$K_ASTI ) AND
                        ( REGEN_FLAG <> GKS$K_IRG_SUPPRESSED )) THEN
                            GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI,
                            GKS$K_IRG_SUPPRESSED );

            END; { Original if }

    END;  { Function SET_UP }

{*****************************************************************}
    {Draw the picture, and place each primitive in a segment...}
PROCEDURE DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
    ROAD, HOUSE, HORIZON : INTEGER );

    CONST
```

(continued on next page)

```
        TEXT_START_X  = 0.05;
        TEXT_START_Y  = 0.9;
        NUM_STARS     = 6;
        NUM_TREE_PTS  = 29;
        NUM_HOUSE_PTS = 12;
        NUM_LAND_PTS  = 15;
        SIDE_START_X  = 0.2;
        SIDE_START_Y  = 0.3;
        SIDE_DIAG_X   = 0.25;
        SIDE_DIAG_Y   = 0.15;
        SIDE_OFF_COL  = 1;
        SIDE_OFF_ROW  = 1;
        SIDE_NUM_COL  = 1;
        SIDE_NUM_ROW  = 2;
        ROAD_START_X  = 0.0;
        ROAD_START_Y  = 0.15;
        ROAD_DIAG_X   = 1.0;
        ROAD_DIAG_Y   = 0.0;
        ROAD_OFF_COL  = 1;
        ROAD_OFF_ROW  = 1;
        ROAD_NUM_COL  = 10;
        ROAD_NUM_ROW  = 1;
        LIGHT         = 2;
        DARK          = 3;
        LARGER        = 0.04;
        THREE         = 3;
        BW_NUM_PTS    = 9;

    VAR

        ERROR_STATUS : INTEGER;
        DUMMY_INTEGER, WS_TYPE : INTEGER;
        COLOR_FLAG, NUM_INDEXES : INTEGER;
        MAX_WIDTH, DUMMY_REAL, NOM_WIDTH : REAL;
        WIDER : REAL;

        DUMMY_INT_ARRAY : [STATIC] ARRAY[ 1..50 ] OF INTEGER;
        DUMMY_STRING    : VARYING[ 80 ] OF CHAR;

        SIDE_COLORS     : [STATIC] ARRAY[ 1..1,  1..2 ] OF INTEGER :=
              ( ( 2, 3 ) );

        ROAD_COLORS     : [STATIC] ARRAY[ 1..10, 1..1 ] OF INTEGER:=
              ( ( 2 ), ( 3 ), ( 2 ), ( 3 ), ( 2 ),
                ( 3 ), ( 2 ), ( 3 ), ( 2 ), ( 3 ) );

        STARS_X_VALUES  : [STATIC] ARRAY[ 1..6  ] OF REAL :=
            ( 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 );

        STARS_Y_VALUES  : [STATIC] ARRAY[ 1..6  ] OF REAL :=
            ( 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 );
```

(continued on next page)

```
    TREE_X          : [STATIC] ARRAY[ 1..29 ] OF REAL :=
        ( 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
          0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
          0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
          0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
          0.515, 0.51, 0.495, 0.475, 0.425 );

    TREE_Y          : [STATIC] ARRAY[ 1..29 ] OF REAL :=
        ( 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
          0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
          0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
          0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
          0.5, 0.425, 0.38, 0.33, 0.28 );

    HOUSE_X         : [STATIC] ARRAY[ 1..12 ] OF REAL :=
          ( 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
            0.25, 0.25, 0.2, 0.075, 0.1, 0.1 );

    HOUSE_Y         : [STATIC] ARRAY[ 1..12 ] OF REAL :=
          ( 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
            0.75, 0.7, 0.75, 0.6, 0.6, 0.3 );


    LAND_X          : [STATIC] ARRAY[ 1..15 ] OF REAL :=
          ( 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
            0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 );

    LAND_Y          : [STATIC] ARRAY[ 1..15 ] OF REAL :=
          ( 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
            0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375, 0.385);

    BW_X_VALUES     : [STATIC] ARRAY[ 1..9  ] OF REAL :=
          ( 0.0, 0.0, 0.2, 0.2, 0.25, 0.25, 1.0, 1.0, 0.0 );
    BW_Y_VALUES     : [STATIC] ARRAY[ 1..9  ] OF REAL :=
          ( 0.0, 0.15, 0.15, 0.3, 0.3, 0.15, 0.15, 0.0, 0.0 );

BEGIN  { Procedure DRAW_PICTURE }

    WIDER := 3.0;
    GKS$SET_TEXT_HEIGHT( LARGER );
    GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS );
    GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID );
    GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED );
```

(continued on next page)

```
{ Obtain the workstation type. }
GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
    WS_TYPE, DUMMY_INTEGER );

{ Make sure that you don't ask for a line wider than the
  workstation's widest line. }
GKS$INQ_PLINE_FAC( WS_TYPE, ERROR_STATUS,
    DUMMY_INTEGER, DUMMY_INT_ARRAY, DUMMY_INTEGER,
    NOM_WIDTH, DUMMY_REAL, MAX_WIDTH, DUMMY_INTEGER,
    DUMMY_INTEGER );

WHILE (( WIDER * NOM_WIDTH ) > MAX_WIDTH ) DO
    WIDER := WIDER - 0.1;

GKS$SET_PLINE_LINEWIDTH( WIDER );

GKS$CREATE_SEG( TITLE );
GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' );
GKS$CLOSE_SEG;


GKS$CREATE_SEG( STARS );
GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES, STARS_Y_VALUES );
GKS$CLOSE_SEG;

GKS$CREATE_SEG( TREE );
GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y );
GKS$CLOSE_SEG;

{ Check to see if you are working with a color workstation. }
GKS$INQ_COLOR_FAC( WS_TYPE, ERROR_STATUS,
    DUMMY_INTEGER, COLOR_FLAG, NUM_INDEXES );
```

(continued on next page)

```
    { For all workstations that have less than three color indexes,
      use GKS$FILL_AREA instead of GKS$CELL_ARRAY for the
      sidewalk and road. }
    IF ( NUM_INDEXES < THREE ) THEN
        BEGIN
            GKS$CREATE_SEG( SIDE );
            GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_HATCH );
            GKS$FILL_AREA( BW_NUM_PTS, BW_X_VALUES, BW_Y_VALUES );
            GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID );
            GKS$CLOSE_SEG;
        END
    ELSE
        BEGIN
            GKS$CREATE_SEG( SIDE );
            GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
                SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL,
                SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
                SIDE_COLORS );
            GKS$CLOSE_SEG;
            GKS$CREATE_SEG( ROAD );
            GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
                ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL,
                ROAD_OFF_ROW, ROAD_NUM_COL, ROAD_NUM_ROW,
                ROAD_COLORS );
            GKS$CLOSE_SEG;
        END;

    GKS$CREATE_SEG( HORIZON );
    GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y );
    GKS$CLOSE_SEG;


    GKS$CREATE_SEG( HOUSE );
    { Only change the color index if working with a workstation
      with three or more color indexes.}
    IF ( NUM_INDEXES > THREE ) THEN
        GKS$SET_FILL_COLOR_INDEX( DARK );

    GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y );
    GKS$CLOSE_SEG;

END;  { Procedure DRAW_PICTURE }
```

(continued on next page)

Example B–4 (Cont.): VAX Pascal Sample Program

```
{******************************************************************}
    { Clean up the DEC GKS and the workstation environments... }
PROCEDURE CLEAN_UP( WS_ID : INTEGER );

    VAR C : CHAR;

    BEGIN  { Procedure CLEAN_UP }

        GKS$UPDATE_WS( WS_ID, 1 );
        READ( C );

        GKS$DEACTIVATE_WS( WS_ID );
        GKS$CLOSE_WS( WS_ID );
        GKS$CLOSE_GKS;

    END;  { Procedure CLEAN_UP }

{******************************************************************}
BEGIN { Main Program }

    { Set up workstation and return flag to show if workstation was set
      up properly. }
    SETUP_OK := SET_UP( WS_ID );

    { If the workstation was set up properly, execute the
      remainder of the program. }
    IF ( SETUP_OK ) THEN
        BEGIN
            DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE,
                ROAD, HOUSE, HORIZON );
            CLEAN_UP( WS_ID );
        END;

END.  { Main Program }
```

B.5 VAX Ada

Example B–5 presents the Starry Night program written in VAX™ Ada.®

The following DIGITAL Command Language commands assume that your source code file is called STARRY_NIGHT.ADA, and that your definition file, source code, and ACS library are all in the directory [DIRECTORY]. To run this program, you need to execute the following commands on the DCL Line:

™ VAX is a trademark of DIGITAL Equipment Corporation.

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

```
$ SET DEFAULT [DIRECTORY] RETURN
$ ACS CREATE LIBRARY [DIRECTORY] RETURN
$ ACS SET LIBRARY [DIRECTORY] RETURN
$ ADA  GKSDEFS.ADA RETURN
$ ADA  STARRY_NIGHT.ADA RETURN
$ ACS  LINK  STARRY_NIGHT RETURN
$ RUN  STARRY_NIGHT RETURN
```

Example B–5: VAX Ada Sample Program

```
    with gksdefs;   use gksdefs;
    with text_io;   use text_io;

    procedure starry_night is

        ws_id  : constant integer := 1;
        title  : constant integer := 1;
        stars  : constant integer := 2;
        tree   : constant integer := 3;
        side   : constant integer := 4;
        road   : constant integer := 5;
        horizon: constant integer := 6;
        house  : constant integer := 7;

        package INT_IO is new INTEGER_IO( NUM => INTEGER );
--      ************************************************************
--      Set up the DEC GKS and the workstation environments...
        procedure set_up ( ws_id : in integer ) is
            error_status, category, inquiry_okay : integer;
            dummy_integer, def_mode, regen_flag  : integer;
            status, ws_type : integer;
            dummy_string : string ( 1..80 );

            begin
            inquiry_okay := 0;

            gks_open_gks( status, "SYS$ERROR:" );
            gks_inq_ws_category( status, GKS_K_WSTYPE_DEFAULT,
                error_status, category );

--          Make sure that the workstation type is valid.
            if (( error_status /= inquiry_okay ) or
                (( category /= GKS_K_WSCAT_OUTPUT ) and
                ( category /= GKS_K_WSCAT_OUTIN ))) then
                put ( "The specified workstation type is invalid." );
                put ( "Error status:" );
                int_io.put (  error_status );
                return;
            end if;


            gks_open_ws( status, ws_id, GKS_K_CONID_DEFAULT,
                                        GKS_K_WSTYPE_DEFAULT );
            gks_activate_ws( status, ws_id );

--          Make sure that the deferral mode and regeneration flag are
--          properly set.
            gks_inq_ws_type ( status, ws_id, error_status, dummy_string,
                ws_type, dummy_integer );
            gks_inq_def_defer_state( status, ws_type, error_status, def_mode,
                regen_flag );
```

(continued on next page)

```
--          You can check the status of the inquiry function execution, as
--          follows:
            if ( error_status /= inquiry_okay ) then
                put( "The deferral inquiry caused an error." );
                put( "Error status:" );
                int_io.put( error_status );
                return;
            end if;

--          Defer output as long as possible and suppress implicit
--          regenerations.
            if (( def_mode /= GKS_K_ASTI ) and
                ( regen_flag /= GKS_K_IRG_SUPPRESSED )) then
                gks_set_defer_state( status, ws_id, GKS_K_ASTI,
                    GKS_K_IRG_SUPPRESSED );
            end if;

        end set_up;
--      ************************************************************
--      Draw the picture, and place each primitive in a segment...
        procedure draw_picture ( ws_id, title, stars, tree, side,
                road, house, horizon : in integer ) is

            num_stars     : constant integer := 6;
            num_tree_pts  : constant integer := 29;
            num_house_pts: constant integer := 12;
            num_land_pts  : constant integer := 15;
            side_off_col  : constant integer := 1;
            side_off_row  : constant integer := 1;
            side_num_col  : constant integer := 1;
            side_num_row  : constant integer := 2;
            road_off_col  : constant integer := 1;
            road_off_row  : constant integer := 1;
            road_num_col  : constant integer := 10;
            road_num_row  : constant integer := 1;


            light         : constant integer := 2;
            dark          : constant integer := 3;
            bw_num_pts    : constant integer := 9;
            error_status  : integer;
            inquire_okay  : integer := 0;
            ws_type       : integer;
            color_flag    : integer;
            num_indexes   : integer;
            dummy_string  : string ( 1..80 );
            status        : integer;
            dummy_int_array : INTEGER_ARRAY (1..50);
```

(continued on next page)

```
side_colors   : constant INTEGER_MATRIX_TYPE(1..1,1..2) :=
                ( 1 => ( 1 => 2, 2 => 3 ) );
road_colors   : constant INTEGER_MATRIX_TYPE(1..10,1..1) :=
                ( 1 => (1=>2), 2 => (1=>3), 3 => (1=>2),
                  4 => (1=>3), 5 => (1=>2), 6 => (1=>3),
                  7 => (1=>2), 8 => (1=>3), 9 => (1=>2),
                  10=> (1=>3) );

text_start_x : constant float := 0.05;
text_start_y : constant float := 0.9;
side_start_x : constant float := 0.2;
side_start_y : constant float := 0.3;
side_diag_x  : constant float := 0.25;
side_diag_y  : constant float := 0.15;
road_start_x : constant float := 0.0;
road_start_y : constant float := 0.15;
road_diag_x  : constant float := 1.0;
road_diag_y  : constant float := 0.0;

larger    : float := 0.04;
wider     : float := 3.0;
max_width : float;
dummy_real: float;
nom_width : float;
dummy_integer : integer;

stars_x_values : constant FLOAT_ARRAY(1..6) :=
                 ( 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 );
stars_y_values : constant FLOAT_ARRAY(1..6) :=
                 ( 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 );
tree_x         : constant FLOAT_ARRAY(1..29) :=
                 ( 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
                   0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
                   0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
                   0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
                   0.515, 0.51, 0.495, 0.475, 0.425 );


tree_y         : constant FLOAT_ARRAY(1..29) :=
                 ( 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
                   0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
                   0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
                   0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
                   0.5, 0.425, 0.38, 0.33, 0.28 );
house_x        : constant FLOAT_ARRAY(1..12) :=
                 ( 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
                   0.25, 0.25, 0.2, 0.075, 0.1, 0.1 );
house_y        : constant FLOAT_ARRAY(1..12) :=
                 ( 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
                   0.75, 0.7, 0.75, 0.6, 0.6, 0.3 );
```

(continued on next page)

```
        land_x        : constant FLOAT_ARRAY(1..15) :=
                        ( 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
                          0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 );
        land_y        : constant FLOAT_ARRAY(1..15) :=
                        ( 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
                          0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375,
                          0.385 );
        bw_x_values   : constant FLOAT_ARRAY(1..9) :=
                        ( 0.0, 0.0, 0.2, 0.2, 0.25, 0.25,  1.0, 1.0, 0.0 );
        bw_y_values   : constant FLOAT_ARRAY(1..9) :=
                        ( 0.0, 0.15, 0.15, 0.3, 0.3, 0.15, 0.15, 0.0, 0.0 );

        begin
        gks_set_text_height( status, larger );
        gks_set_pmark_type( status, GKS_K_MARKERTYPE_PLUS );
        gks_set_fill_int_style( status, GKS_K_INTSTYLE_SOLID );
        gks_set_pline_linetype( status, GKS_K_LINETYPE_DASHED_DOTTED );

--      Obtain the workstation type.
        gks_inq_ws_type( status, ws_id, error_status, dummy_string,
            ws_type, dummy_integer );

--      Make sure that you don't ask for a line wider than the
--      workstation's widest line.
        gks_inq_pline_fac( status, ws_type, error_status, dummy_integer,
            dummy_int_array, dummy_integer, nom_width, dummy_real,
            max_width, dummy_integer, dummy_integer );

        while ( wider * nom_width ) > max_width loop
            wider := wider - 0.1;
        end loop;


        gks_set_pline_linewidth( status, wider );

        gks_create_seg( status, title );
        gks_text( status, text_start_x, text_start_y, "Starry Night" );
        gks_close_seg( status );

        gks_create_seg( status, stars );
        gks_polymarker( status, num_stars, stars_x_values, stars_y_values );
        gks_close_seg( status );
```

(continued on next page)

```
                gks_create_seg( status, tree );
                gks_fill_area( status, num_tree_pts, tree_x, tree_y );
                gks_close_seg( status );

--              Check to see if you are working with a color workstation.
                gks_inq_color_fac( status, ws_type, error_status, dummy_integer,
                    color_flag, num_indexes );

--              For all workstations that have less than three color indexes,
--              use GKS$FILL_AREA instead of GKS$CELL_ARRAY for the
--              sidewalk and road.
                if num_indexes < 3 then
                    gks_create_seg( status, side );
                    gks_set_fill_int_style( status, GKS_K_INTSTYLE_HATCH );
                    gks_fill_area( status, bw_num_pts, bw_x_values, bw_y_values );
                    gks_set_fill_int_style( status, GKS_K_INTSTYLE_SOLID );
                    gks_close_seg( status );
                else
                    gks_create_seg( status, side );
                    gks_cell_array( status, side_start_x, side_start_y,
                        side_diag_x, side_diag_y, side_off_col, side_off_row,
                        side_num_col, side_num_row, side_colors );
                    gks_close_seg( status );
                    gks_create_seg( status, road );
                    gks_cell_array( status, road_start_x, road_start_y,
                        road_diag_x, road_diag_y, road_off_col,
                        road_off_row, road_num_col, road_num_row,
                        road_colors );
                    gks_close_seg( status );
                end if;

                gks_create_seg( status, horizon );
                gks_polyline( status, num_land_pts, land_x, land_y );
                gks_close_seg( status );

                gks_create_seg( status, house );
--              Only change the color index if working with a workstation
--              with more than three color indexes.
                if num_indexes >= 3 then
                    gks_set_fill_color_index( status, dark );
                end if;
                gks_fill_area( status, num_house_pts, house_x, house_y );
                gks_close_seg ( status );

            end draw_picture;
--          ************************************************************
--          Clean up the DEC GKS and the workstation environments...
            procedure clean_up( ws_id: in integer ) is
                status : integer;
                dummy_string : string(1..1);
                begin
                gks_update_ws( status, ws_id, GKS_K_PERFORM_FLAG );
```

(continued on next page)

Example B–5 (Cont.): VAX Ada Sample Program

```
                get( dummy_string );
                gks_deactivate_ws( status, ws_id );
                gks_close_ws( status, ws_id );
                gks_close_gks( status );

            end clean_up;
--          main procedure starry_night
        begin

            set_up( ws_id );
            draw_picture( ws_id, title, stars, tree, side, road, house, horizon );
            clean_up( ws_id );
        end starry_night;
```

B.6 VAX PL/I

Example B–6 presents the Starry Night program written in VAX PL/I.

Example B–6: VAX PL/I Sample Program

```
starry_night: PROCEDURE OPTIONS( MAIN );

    /* External procedure declarations for GKS */
    %INCLUDE 'sys$library:gksdefs.pli';

    DECLARE SET_UP ENTRY( FIXED BIN );
    DECLARE CLEAN_UP ENTRY( FIXED BIN );
    DECLARE DRAW_PICTURE ENTRY
        ( FIXED BIN, FIXED BIN, FIXED BIN, FIXED BIN,
          FIXED BIN, FIXED BIN, FIXED BIN, FIXED BIN );

    DECLARE WS_ID FIXED BIN INITIAL( 1 ),
            TITLE FIXED BIN INITIAL( 1 ),
            STARS FIXED BIN INITIAL( 2 ),
            TREE  FIXED BIN INITIAL( 3 ),
            SIDE  FIXED BIN INITIAL( 4 ),
            ROAD  FIXED BIN INITIAL( 5 ),
            HORIZON FIXED BIN INITIAL( 6 ),
            HOUSE FIXED BIN INITIAL( 7 );

    CALL SET_UP( WS_ID );
    CALL DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON );
    CALL CLEAN_UP( WS_ID );

    END starry_night;
```

(continued on next page)

```
/* Set up the DEC GKS and the workstation environments. */
set_up: PROCEDURE( WS_ID );

%INCLUDE 'sys$library:gksdefs.pli';

DECLARE (WS_ID, WS_TYPE, ERROR_STATUS, CATEGORY,
         DUMMY_INTEGER, DEF_MODE, REGEN_FLAG) FIXED BIN;
DECLARE INQUIRY_OKAY FIXED BIN INITIAL( 0 );
DECLARE DUMMY_STRING CHAR(80);

CALL GKS$OPEN_GKS( 'SYS$ERROR:' );

CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT, ERROR_STATUS, CATEGORY );

/* Make sure that the workstation type is valid. */
IF (( ERROR_STATUS ^= INQUIRY_OKAY ) |
   (( CATEGORY ^= GKS$K_WSCAT_OUTPUT ) &
   (( CATEGORY ^= GKS$K_WSCAT_OUTIN )))) THEN
        DO;
        PUT SKIP LIST('The specified workstation type is invalid.');
        PUT SKIP LIST('Error status:', ERROR_STATUS );
        STOP;
        END;

CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_WSTYPE_DEFAULT );
CALL GKS$ACTIVATE_WS( WS_ID );

/*
 * Make sure that the deferral mode and regeneration flag are
 * properly set.
 */
CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
                      WS_TYPE, DUMMY_INTEGER );

CALL GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,
                      DEF_MODE, REGEN_FLAG );

/*
 * You can check the status of the inquiry routine execution, as
 * follows:
 */
IF ( ERROR_STATUS ^= INQUIRY_OKAY ) THEN
    DO;
    PUT SKIP LIST ('The deferral inquiry caused an error.');
    PUT SKIP LIST ('Error status:', ERROR_STATUS );
    STOP;
    END;
```

(continued on next page)

```
/*
 * Defer output as long as possible and suppress
 * implicit regenerations.
 */
IF (( DEF_MODE ^= GKS$K_ASTI ) &
    ( REGEN_FLAG ^= GKS$K_IRG_SUPPRESSED )) THEN
        CALL GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI, GKS$K_IRG_SUPPRESSED );

END set_up;


/* Draw the picture, and place each primitive in a segment. */
draw_picture: PROCEDURE( WS_ID, TITLE, STARS, TREE, SIDE, ROAD,
                         HOUSE, HORIZON );

%INCLUDE 'sys$library:gksdefs.pli';

DECLARE (WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON,
    ERROR_STATUS,
    DUMMY_INTEGER, WS_TYPE, DUMMY_INT_ARRAY( 50 ),
    COLOR_FLAG, NUM_INDEXES ) FIXED BIN;

DECLARE DUMMY_STRING CHAR(80);

DECLARE ( MAX_WIDTH, DUMMY_REAL, NOM_WIDTH ) FLOAT BIN;
```

(continued on next page)

```
DECLARE TEXT_START_X FLOAT BIN INITIAL( 0.05 ),
    TEXT_START_Y FLOAT BIN INITIAL( 0.9 ),
    NUM_STARS    FIXED BIN INITIAL( 6 ),
    NUM_TREE_PTS FIXED BIN INITIAL( 29 ),
    NUM_HOUSE_PTS FIXED BIN INITIAL( 12 ),
    NUM_LAND_PTS FIXED BIN INITIAL( 15 ),
    SIDE_START_X FLOAT BIN INITIAL( 0.2 ),
    SIDE_START_Y FLOAT BIN INITIAL( 0.3 ),
    SIDE_DIAG_X  FLOAT BIN INITIAL( 0.25 ),
    SIDE_DIAG_Y  FLOAT BIN INITIAL( 0.15 ),
    SIDE_OFF_COL FIXED BIN INITIAL( 1 ),
    SIDE_OFF_ROW FIXED BIN INITIAL( 1 ),
    SIDE_NUM_COL FIXED BIN INITIAL( 1 ),
    SIDE_NUM_ROW FIXED BIN INITIAL( 2 ),
    ROAD_START_X FLOAT BIN INITIAL( 0.0 ),
    ROAD_START_Y FLOAT BIN INITIAL( 0.15 ),
    ROAD_DIAG_X  FLOAT BIN INITIAL( 1.0 ),
    ROAD_DIAG_Y  FLOAT BIN INITIAL( 0.0 ),
    ROAD_OFF_COL FIXED BIN INITIAL( 1 ),
    ROAD_OFF_ROW FIXED BIN INITIAL( 1 ),
    ROAD_NUM_COL FIXED BIN INITIAL( 10 ),
    ROAD_NUM_ROW FIXED BIN INITIAL( 1 ),
    LIGHT FIXED BIN INITIAL( 2 ),
    DARK FIXED BIN INITIAL( 3 ),
    LARGER FLOAT BIN INITIAL( 0.04 ),
    WIDER FLOAT BIN INITIAL( 3.0 ),
    THREE FIXED BIN INITIAL( 3 ),
    BW_NUM_PTS FIXED BIN INITIAL( 9 );

DECLARE BW_X_VALUES (9) FLOAT BIN INITIAL
    ( 0.0, 0.0, 0.2, 0.2, 0.25, 0.25, 1.0, 1.0, 0.0 );
DECLARE BW_Y_VALUES (9) FLOAT BIN INITIAL
    ( 0.0, 0.15, 0.15, 0.3, 0.3, 0.15, 0.15, 0.0, 0.0 );


DECLARE SIDE_COLORS ( 1, 2 ) FIXED BIN INITIAL ( 2, 3 );
DECLARE ROAD_COLORS ( 10, 1 ) FIXED BIN INITIAL
    ( 2, 3, 2, 3, 2, 3, 2, 3, 2, 3 );

DECLARE STARS_X_VALUES (6) FLOAT BIN INITIAL
    ( 0.05, 0.06, 0.36, 0.66, 0.835, 0.92 );
DECLARE STARS_Y_VALUES (6) FLOAT BIN INITIAL
    ( 0.7, 0.86, 0.81, 0.86, 0.701, 0.82 );
```

(continued on next page)

```
DECLARE TREE_X (29) FLOAT BIN INITIAL
    ( 0.425, 0.5, 0.52, 0.54, 0.6, 0.575,
      0.56, 0.559, 0.64, 0.69, 0.689, 0.66,
      0.63, 0.645, 0.59, 0.53, 0.48, 0.45,
      0.42, 0.375, 0.35, 0.375, 0.44, 0.45,
      0.515, 0.51, 0.495, 0.475, 0.425 );
DECLARE TREE_Y (29) FLOAT BIN INITIAL
    ( 0.28, 0.3, 0.26, 0.3, 0.28, 0.33,
      0.42, 0.49, 0.53, 0.57, 0.61, 0.64,
      0.66, 0.71, 0.76, 0.78, 0.75, 0.71,
      0.65, 0.645, 0.6, 0.55, 0.54, 0.5,
      0.5, 0.425, 0.38, 0.33, 0.28 );

DECLARE HOUSE_X (12) FLOAT BIN INITIAL
    ( 0.1, 0.3, 0.3, 0.325, 0.3, 0.3,
      0.25, 0.25, 0.2, 0.075, 0.1, 0.1 );
DECLARE HOUSE_Y (12) FLOAT BIN INITIAL
    ( 0.3, 0.3, 0.6, 0.6, 0.64, 0.75,
      0.75, 0.7, 0.75, 0.6, 0.6, 0.3 );

DECLARE LAND_X (15) FLOAT BIN INITIAL
    ( 0.0, 0.04, 0.055, 0.08, 0.1, 0.3,
      0.375, 0.44, 0.49, 0.56, 0.68, 0.8, 0.9, 0.95, 1.0 );
DECLARE LAND_Y (15) FLOAT BIN INITIAL
    ( 0.35, 0.375, 0.376, 0.36, 0.365, 0.366,
      0.38, 0.385, 0.375, 0.36, 0.38, 0.35, 0.359, 0.375, 0.385 );

CALL GKS$SET_TEXT_HEIGHT( LARGER );
CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS );
CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID );
CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED );

/* Obtain the workstation type. */
CALL GKS$INQ_WS_TYPE( WS_ID, ERROR_STATUS, DUMMY_STRING,
                      WS_TYPE, DUMMY_INTEGER );


/*
 * Make sure that you don't ask for a line wider than the
 * workstation's widest line.
 */
CALL GKS$INQ_PLINE_FAC( WS_TYPE, ERROR_STATUS,
        DUMMY_INTEGER, DUMMY_INT_ARRAY, DUMMY_INTEGER,
        NOM_WIDTH, DUMMY_REAL, MAX_WIDTH, DUMMY_INTEGER, DUMMY_INTEGER );

DO WHILE (( WIDER * NOM_WIDTH ) > MAX_WIDTH );
    WIDER = WIDER - 0.1;
END;

CALL GKS$SET_PLINE_LINEWIDTH( WIDER );

CALL GKS$CREATE_SEG( TITLE );
CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' );
CALL GKS$CLOSE_SEG();
```

(continued on next page)

```
CALL GKS$CREATE_SEG( STARS );
CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES, STARS_Y_VALUES );
CALL GKS$CLOSE_SEG();

CALL GKS$CREATE_SEG( TREE );
CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X, TREE_Y );
CALL GKS$CLOSE_SEG();

/* Check to see if you are working with a color workstation. */
CALL GKS$INQ_COLOR_FAC( WS_TYPE, ERROR_STATUS,
                        DUMMY_INTEGER, COLOR_FLAG, NUM_INDEXES );

/*
 * For all workstations with less than three color indexes,
 * use GKS$FILL_AREA instead of GKS$CELL_ARRAY for the sidewalk
 * and road.
 */
IF ( NUM_INDEXES < THREE ) THEN
    DO;
    CALL GKS$CREATE_SEG( SIDE );
    CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_HATCH );
    CALL GKS$FILL_AREA( BW_NUM_PTS, BW_X_VALUES, BW_Y_VALUES );
    CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID );
    CALL GKS$CLOSE_SEG();
    END;
ELSE


    DO;
    CALL GKS$CREATE_SEG( SIDE );
    CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,
              SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL,
              SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,
              SIDE_COLORS );
    CALL GKS$CLOSE_SEG();
    CALL GKS$CREATE_SEG( ROAD );
    CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,
              ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL,
              ROAD_OFF_ROW, ROAD_NUM_COL, ROAD_NUM_ROW,
              ROAD_COLORS );
    CALL GKS$CLOSE_SEG();
    END;

CALL GKS$CREATE_SEG( HORIZON );
CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X, LAND_Y );
CALL GKS$CLOSE_SEG();
```

(continued on next page)

```
CALL GKS$CREATE_SEG( HOUSE );
/*
 * Only change the color index if working with a
 * workstation with more than three color indexes.
 */
IF ( NUM_INDEXES >= THREE ) THEN
    CALL GKS$SET_FILL_COLOR_INDEX( DARK );

CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X, HOUSE_Y );
CALL GKS$CLOSE_SEG();

END draw_picture;

/* Clean up the DEC GKS and the workstation environments. */
clean_up: PROCEDURE( WS_ID );

%INCLUDE 'sys$library:gksdefs.pli';

DECLARE (WS_ID, DUMMY) FIXED BIN;

CALL GKS$UPDATE_WS( WS_ID, 1 );
GET LIST( DUMMY );

CALL GKS$DEACTIVATE_WS( WS_ID );
CALL GKS$CLOSE_WS( WS_ID );
CALL GKS$CLOSE_GKS();

END clean_up;
```

B.7 VAX BASIC

Example B–7 presents the Starry Night program written in VAX BASIC.

Example B–7: VAX BASIC Sample Program

```
10  OPTION TYPE = EXPLICIT

    DECLARE LONG CONSTANT WS_ID   =  1%
    DECLARE LONG CONSTANT TITLE   =  1%
    DECLARE LONG CONSTANT STARS   =  2%
    DECLARE LONG CONSTANT TREE    =  3%
    DECLARE LONG CONSTANT SIDE    =  4%
    DECLARE LONG CONSTANT ROAD    =  5%
    DECLARE LONG CONSTANT HORIZON =  6%
    DECLARE LONG CONSTANT HOUSE   =  7%

    CALL SET_UP( WS_ID )
    CALL DRAW_PICTURE( WS_ID, TITLE, STARS, TREE, SIDE, ROAD, HOUSE, HORIZON )
    CALL CLEAN_UP( WS_ID )
    END

100
    !*************************************************************
    !    Clean up DEC GKS and the workstation environment.

    SUB CLEAN_UP( LONG WS_ID )

    OPTION TYPE = EXPLICIT
    DECLARE STRING DUMMY_STRING

    !   Update the workstation and wait for keypress to continue.
    CALL GKS$UPDATE_WS( WS_ID, 1% )
    INPUT '';DUMMY_STRING

    !   Close up the workstation and GKS.
    CALL GKS$DEACTIVATE_WS( WS_ID )
    CALL GKS$CLOSE_WS( WS_ID )
    CALL GKS$CLOSE_GKS()
    END SUB

200
    !*************************************************************
    !    Set up DEC GKS and the workstation environment.

    SUB SET_UP( LONG WS_ID )
    OPTION TYPE = EXPLICIT
    %NOLIST
    %INCLUDE "SYS$LIBRARY:GKSDEFS.BAS"
    %LIST
```

(continued on next page)

```
    DECLARE LONG    CONSTANT INQUIRY_OKAY = 0%

    DECLARE LONG    WS_TYPE
    DECLARE LONG    ERROR_STATUS, CATEGORY
    DECLARE LONG    DUMMY_INTEGER, DEF_MODE, REGEN_FLAG
    DECLARE STRING  DUMMY_STRING

    !   Open GKS and inquire the workstation category.
    CALL GKS$OPEN_GKS( 'SYS$ERROR:' )
    CALL GKS$INQ_WS_CATEGORY( GKS$K_WSTYPE_DEFAULT, ERROR_STATUS, CATEGORY )

    !   Make sure that the workstation type is valid for this program
    !   and that an error did not occur in inquiring the workstation category.
    !   Change the category below to GKS$K_WSCAT_WISS

210 IF ERROR_STATUS <> INQUIRY_OKAY         &
        OR  CATEGORY <> GKS$K_WSCAT_OUTPUT          &
        AND CATEGORY <> GKS$K_WSCAT_OUTIN        THEN
            PRINT "The specified workstation type is invalid."
            PRINT "Error status:", ERROR_STATUS
            STOP
    END IF

    !   Open and activate the workstation.
    CALL GKS$OPEN_WS( WS_ID, GKS$K_CONID_DEFAULT, GKS$K_WSTYPE_DEFAULT )
    CALL GKS$ACTIVATE_WS( WS_ID )

    !   Make sure that the deferral mode and regeneration flag are
    !   properly set.
    CALL GKS$INQ_WS_TYPE(                                              &
        WS_ID, ERROR_STATUS, DUMMY_STRING, WS_TYPE, DUMMY_INTEGER )

    CALL GKS$INQ_DEF_DEFER_STATE( WS_TYPE, ERROR_STATUS,               &
        DEF_MODE, REGEN_FLAG )

    !   Make sure that the defer inquiry did not cause an error.
    IF ( ERROR_STATUS <> INQUIRY_OKAY ) THEN
        PRINT 'The deferral inquiry caused an error.'
        PRINT 'Error status:', ERROR_STATUS
        STOP
    END IF


    !   Defer output as long as possible and suppress implicit
    !   regenerations.
    IF DEF_MODE <> GKS$K_ASTI AND REGEN_FLAG <> GKS$K_IRG_SUPPRESSED &
    THEN
        CALL GKS$SET_DEFER_STATE( WS_ID, GKS$K_ASTI, GKS$K_IRG_SUPPRESSED )
    END IF
    END SUB

300
    !************************************************************
    !   Draw the picture, and place each primitive in a segment.
```

(continued on next page)

```
SUB DRAW_PICTURE( LONG WS_ID, TITLE, STARS, TREE, SIDE,          &
    ROAD, HOUSE, HORIZON )

OPTION TYPE = EXPLICIT
%NOLIST
%INCLUDE "SYS$LIBRARY:GKSDEFS.BAS"
%LIST
DECLARE SINGLE  CONSTANT TEXT_START_X  = 0.05
DECLARE SINGLE  CONSTANT TEXT_START_Y  = 0.9
DECLARE LONG    CONSTANT NUM_STARS     = 6%
DECLARE LONG    CONSTANT NUM_TREE_PTS  = 29%
DECLARE LONG    CONSTANT NUM_HOUSE_PTS = 12%
DECLARE LONG    CONSTANT NUM_LAND_PTS  = 15%
DECLARE SINGLE  CONSTANT SIDE_START_X  = 0.2
DECLARE SINGLE  CONSTANT SIDE_START_Y  = 0.3
DECLARE SINGLE  CONSTANT SIDE_DIAG_X   = 0.25
DECLARE SINGLE  CONSTANT SIDE_DIAG_Y   = 0.15
DECLARE LONG    CONSTANT SIDE_OFF_COL  = 0%
DECLARE LONG    CONSTANT SIDE_OFF_ROW  = 0%
DECLARE LONG    CONSTANT SIDE_NUM_COL  = 1%
DECLARE LONG    CONSTANT SIDE_NUM_ROW  = 2%
DECLARE SINGLE  CONSTANT ROAD_START_X  = 0.0
DECLARE SINGLE  CONSTANT ROAD_START_Y  = 0.15
DECLARE SINGLE  CONSTANT ROAD_DIAG_X   = 1.0
DECLARE SINGLE  CONSTANT ROAD_DIAG_Y   = 0.0
DECLARE LONG    CONSTANT ROAD_OFF_COL  = 0%
DECLARE LONG    CONSTANT ROAD_OFF_ROW  = 0%
DECLARE LONG    CONSTANT ROAD_NUM_COL  = 10%
DECLARE LONG    CONSTANT ROAD_NUM_ROW  = 1%
DECLARE LONG    CONSTANT LIGHT  = 2%
DECLARE LONG    CONSTANT DARK   = 3%
DECLARE SINGLE  CONSTANT LARGER = 0.04
DECLARE LONG    CONSTANT THREE  = 3%
DECLARE LONG    CONSTANT BW_NUM_PTS = 9%


DECLARE LONG    ERROR_STATUS, DUMMY_INTEGER
DECLARE LONG    WS_TYPE, COLOR_FLAG, NUM_INDEXES
DECLARE SINGLE  MAX_WIDTH, DUMMY_REAL,NOM_WIDTH
DECLARE SINGLE  WIDER
DECLARE STRING  DUMMY_STRING

DECLARE LONG SIDE_COLORS( 0, 1 )
DECLARE LONG ROAD_COLORS( 9, 0 )
DECLARE LONG DUMMY_INT_ARRAY( 49 )
DECLARE LONG I

DECLARE SINGLE STARS_X_VALUES( 5 ), STARS_Y_VALUES( 5 ), TREE_X( 28 )
DECLARE SINGLE TREE_Y( 28 ), HOUSE_X( 11 ), HOUSE_Y( 11 ), LAND_X( 14 )
DECLARE SINGLE LAND_Y( 14 ), BW_X_VALUES( 8 ), BW_Y_VALUES( 8 )

!Data Section
```

(continued on next page)

Example B–7 (Cont.): VAX BASIC Sample Program

```
!BW_X_VALUES
DATA 0.0, 0.0, 0.2, 0.2, 0.25, 0.25, 1.0, 1.0, 0.0

!BW_Y_VALUES
DATA 0.0, 0.15, 0.15, 0.3, 0.3, 0.15, 0.15, 0.0, 0.0

!SIDE_COLORS
DATA 2, 3

!ROAD_COLORS
DATA 2, 3, 2, 3, 2, 3, 2, 3, 2, 3

!STARS_X_VALUES
DATA 0.05, 0.06, 0.36, 0.66, 0.835, 0.92

!STARS_Y_VALUES
DATA 0.7, 0.86, 0.81, 0.86, 0.701, 0.82

!TREE_X
DATA 0.425, 0.5, 0.52, 0.54, 0.6, 0.575, 0.56, 0.559, 0.64, 0.69
DATA 0.689, 0.66, 0.63, 0.645, 0.59, 0.53, 0.48, 0.45, 0.42, 0.375
DATA 0.35, 0.375, 0.44, 0.45, 0.515, 0.51, 0.495, 0.475, 0.425

!TREE_Y
DATA 0.28, 0.3, 0.26, 0.3, 0.28, 0.33, 0.42, 0.49, 0.53, 0.57, 0.61
DATA 0.64, 0.66, 0.71, 0.76, 0.78, 0.75, 0.71, 0.65, 0.645, 0.6, 0.55
DATA 0.54, 0.5, 0.5, 0.425, 0.38, 0.33, 0.28

!HOUSE_X
DATA 0.1, 0.3, 0.3, 0.325, 0.3, 0.3, 0.25, 0.25, 0.2, 0.075, 0.1, 0.1

!HOUSE_Y
DATA 0.3, 0.3, 0.6, 0.6, 0.64, 0.75, 0.75, 0.7, 0.75, 0.6, 0.6, 0.3

!LAND_X
DATA 0.0, 0.04, 0.055, 0.08, 0.1, 0.3, 0.375, 0.44, 0.49, 0.56, 0.68
DATA 0.8, 0.9, 0.95, 1.0

!LAND_Y
DATA 0.35, 0.375, 0.376, 0.36, 0.365, 0.366, 0.38, 0.385, 0.375, 0.36
DATA 0.38, 0.35, 0.359, 0.375, 0.385

! Read in the data into the appropriate arrays.
FOR I = 0 TO 8
    READ BW_X_VALUES( I )
NEXT I
FOR I = 0 TO 8
    READ BW_Y_VALUES( I )
NEXT I

FOR I = 0 TO 1
    READ SIDE_COLORS( 0, I )
NEXT I
```

(continued on next page)

```
FOR I = 0 TO 9
    READ ROAD_COLORS( I, 0 )
NEXT I

FOR I = 0 TO 5
    READ STARS_X_VALUES( I )
NEXT I

FOR I = 0 TO 5
    READ STARS_Y_VALUES( I )
NEXT I

FOR I = 0 TO 28
    READ TREE_X( I )
NEXT I

FOR I = 0 TO 28
    READ TREE_Y( I )
NEXT I

FOR I = 0 TO 11
    READ HOUSE_X( I )
NEXT I


FOR I = 0 TO 11
    READ HOUSE_Y( I )
NEXT I

FOR I = 0 TO 14
    READ LAND_X( I )
NEXT I

FOR I = 0 TO 14
    READ LAND_Y( I )
NEXT I

!   Set selected output attributes.
WIDER = 3.0
CALL GKS$SET_TEXT_HEIGHT( LARGER )
CALL GKS$SET_PMARK_TYPE( GKS$K_MARKERTYPE_PLUS )
CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
CALL GKS$SET_PLINE_LINETYPE( GKS$K_LINETYPE_DASHED_DOTTED )

!   Obtain the workstation type.
CALL GKS$INQ_WS_TYPE(                                                  &
    WS_ID, ERROR_STATUS, DUMMY_STRING, WS_TYPE, DUMMY_INTEGER )

!   Make sure that you don't ask for a line wider than the
!   workstation's widest line.
CALL GKS$INQ_PLINE_FAC( WS_TYPE, ERROR_STATUS, DUMMY_INTEGER,          &
    DUMMY_INT_ARRAY(), DUMMY_INTEGER, NOM_WIDTH, DUMMY_REAL,           &
    MAX_WIDTH, DUMMY_INTEGER, DUMMY_INTEGER )
```

(continued on next page)

```
!   If the linewidth is wider than the workstation's widest line
!   decrement it until it is not.
WHILE ( WIDER * NOM_WIDTH ) > MAX_WIDTH
     WIDER = WIDER - 0.1
NEXT

!   Set the polyline linewidth.
CALL GKS$SET_PLINE_LINEWIDTH( WIDER )

!   Ouput the title as a segment.
CALL GKS$CREATE_SEG( TITLE )
CALL GKS$TEXT( TEXT_START_X, TEXT_START_Y, 'Starry Night' )
CALL GKS$CLOSE_SEG()


!   Output the stars as a segment.
CALL GKS$CREATE_SEG( STARS )
CALL GKS$POLYMARKER( NUM_STARS, STARS_X_VALUES(), STARS_Y_VALUES() )
CALL GKS$CLOSE_SEG()

!   Output the tree as a segment.
CALL GKS$CREATE_SEG( TREE )
CALL GKS$FILL_AREA( NUM_TREE_PTS, TREE_X(), TREE_Y() )
CALL GKS$CLOSE_SEG()

!   Check to see if you are working with a color workstation.
CALL GKS$INQ_COLOR_FAC( WS_TYPE, ERROR_STATUS,                      &
    DUMMY_INTEGER, COLOR_FLAG, NUM_INDEXES )

!   Output the sidewalk and the road as segments.
!   For all monochrome that have less than 3 color indexes,
!   use GKS$FILL_AREA instead of GKS$CELL_ARRAY for the
!   sidewalk and road.
IF NUM_INDEXES < THREE THEN
    CALL GKS$CREATE_SEG( SIDE )
    CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_HATCH )
    CALL GKS$FILL_AREA( BW_NUM_PTS, BW_X_VALUES(), BW_Y_VALUES() )
    CALL GKS$SET_FILL_INT_STYLE( GKS$K_INTSTYLE_SOLID )
    CALL GKS$CLOSE_SEG()
ELSE
    CALL GKS$CREATE_SEG( SIDE )
    CALL GKS$CELL_ARRAY( SIDE_START_X, SIDE_START_Y,                &
        SIDE_DIAG_X, SIDE_DIAG_Y, SIDE_OFF_COL,                     &
        SIDE_OFF_ROW, SIDE_NUM_COL, SIDE_NUM_ROW,                   &
        SIDE_COLORS( , ) )
    CALL GKS$CLOSE_SEG()
    CALL GKS$CREATE_SEG( ROAD )
    CALL GKS$CELL_ARRAY( ROAD_START_X, ROAD_START_Y,                &
        ROAD_DIAG_X, ROAD_DIAG_Y, ROAD_OFF_COL,                     &
        ROAD_OFF_ROW, ROAD_NUM_COL, ROAD_NUM_ROW,                   &
        ROAD_COLORS( , ) )
    CALL GKS$CLOSE_SEG()
END IF
```

(continued on next page)

Example B–7 (Cont.): VAX BASIC Sample Program

```
!   Output the horizon as a segment.
CALL GKS$CREATE_SEG( HORIZON )
CALL GKS$POLYLINE( NUM_LAND_PTS, LAND_X(), LAND_Y() )
CALL GKS$CLOSE_SEG()

!   Output the house as a segment.
CALL GKS$CREATE_SEG( HOUSE )
!   Only change the color index if working with a workstation
!   that has more than two color indexes.
IF NUM_INDEXES >= THREE THEN
    CALL GKS$SET_FILL_COLOR_INDEX( DARK )
END IF

CALL GKS$FILL_AREA( NUM_HOUSE_PTS, HOUSE_X(), HOUSE_Y() )
CALL GKS$CLOSE_SEG()
END SUB
```

B.8 VAX COBOL

Example B–8 presents the Starry Night program written in VAX COBOL. The DIGITAL Command Language commands assume that you entered the code listed in the BUILDESC routine (refer to Appendix F, Language-Specific Programming Information, in the *DEC GKS Reference Manual*). To link this program, you need to execute the following commands on the DCL line:

```
$ MACRO  BUILDESC RETURN
$ LINK   cobol_program, BUILDESC RETURN
```

Example B–8: VAX COBOL Sample Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Starry-Night.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY GKSDEFS.
01      ONE PIC S9(9)           USAGE IS COMP   VALUE 1.
01      WS-ID PIC S9(9)         USAGE IS COMP   VALUE 1.
01      WS-TYPE PIC S9(9)       USAGE IS COMP   VALUE 1.
01      TITLE PIC S9(9)         USAGE IS COMP   VALUE 1.
01      STARS PIC S9(9)         USAGE IS COMP   VALUE 2.
01      TREE  PIC S9(9)         USAGE IS COMP   VALUE 3.
01      SIDE  PIC S9(9)         USAGE IS COMP   VALUE 4.
01      ROAD  PIC S9(9)         USAGE IS COMP   VALUE 5.
01      HORIZON PIC S9(9)       USAGE IS COMP   VALUE 6.
01      HOUSE  PIC S9(9)        USAGE IS COMP   VALUE 7.
01      ARRAY_D.
        05 desc OCCURS 11 TIMES PIC S9(9)      USAGE IS COMP.
01      DUMMY-STRING Display.
    03  MAIN-DUMMY PIC X(80).
01      ERROR-MESSAGE Display.
    03  TEXT-PART PIC X(14)     VALUE "Error Status: ".
    03  ERROR-OUT PIC Z(9)      USAGE IS DISPLAY.

01      ERROR-STATUS PIC S9(9)  USAGE IS COMP.
01      CATEGORY PIC S9(9)      USAGE IS COMP.
01      INQUIRY-OKAY PIC S9(9)  USAGE IS COMP   VALUE ZERO.
01      DUMMY-INTEGER PIC S9(9) USAGE IS COMP.
01      DEF-MODE PIC S9(9)      USAGE IS COMP.
01      REGEN-FLAG PIC S9(9)    USAGE IS COMP.
01      NUM_STARS PIC S9(9)     USAGE IS COMP   VALUE 6.
01      NUM_TREE_PTS PIC S9(9)  USAGE IS COMP   VALUE 29.
01      NUM_HOUSE_PTS PIC S9(9) USAGE IS COMP   VALUE 12.
01      NUM_LAND_PTS PIC S9(9)  USAGE IS COMP   VALUE 15.
01      SIDE-OFF-COL PIC S9(9)  USAGE IS COMP   VALUE 1.
01      SIDE-OFF-ROW PIC S9(9)  USAGE IS COMP   VALUE 1.
01      SIDE-NUM-COL PIC S9(9)  USAGE IS COMP   VALUE 1.
01      SIDE-NUM-ROW PIC S9(9)  USAGE IS COMP   VALUE 2.
01      SIDE-START-X            USAGE IS COMP-1 VALUE 0.2.
01      SIDE-START-Y            USAGE IS COMP-1 VALUE 0.3.
01      SIDE-DIAG-X             USAGE IS COMP-1 VALUE 0.25.
01      SIDE-DIAG-Y             USAGE IS COMP-1 VALUE 0.15.
```

(continued on next page)

```
01      SIDEWALK.
    05 SIDE-DATA.
        10  FILLER PIC S9(9)        USAGE IS COMP  VALUE 2.
        10  FILLER PIC S9(9)        USAGE IS COMP  VALUE 3.
    05 SIDE-STUFF REDEFINES SIDE-DATA.
        10 DIM1 OCCURS 1 TIMES.
            15 SIDE-COLORS OCCURS 2 TIMES PIC S9(9)        USAGE IS COMP.
01      ROAD-OFF-COL PIC S9(9)      USAGE IS COMP   VALUE 1.
01      ROAD-OFF-ROW PIC S9(9)      USAGE IS COMP   VALUE 1.
01      ROAD-NUM-COL PIC S9(9)      USAGE IS COMP   VALUE 10.
01      ROAD-NUM-ROW PIC S9(9)      USAGE IS COMP   VALUE 1.
01      ROAD-START-X                USAGE IS COMP-1 VALUE 0.0.
01      ROAD-START-Y                USAGE IS COMP-1 VALUE 0.15.
01      ROAD-DIAG-X                 USAGE IS COMP-1 VALUE 1.0.
01      ROAD-DIAG-Y                 USAGE IS COMP-1 VALUE 0.0.
01      ROAD-ARRAY.
    05 ROAD-DATA.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 2.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 3.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 2.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 3.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 2.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 3.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 2.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 3.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 2.
        10  FILLER  PIC S9(9)       USAGE IS COMP  VALUE IS 3.
    05 ROAD-STUFF REDEFINES ROAD-DATA.
        10 DIM1 OCCURS 10 TIMES.
            15 ROAD-COLORS OCCURS 1 TIMES PIC S9(9)        USAGE IS COMP.
01      LIGHT   PIC S9(9)           USAGE IS COMP   VALUE 2.
01      DARK    PIC S9(9)           USAGE IS COMP   VALUE 3.
01      THREE   PIC S9(9)           USAGE IS COMP   VALUE 3.
01      BW-NUM-PTS PIC S9(9)        USAGE IS COMP   VALUE 9.
01      COLOR-FLAG PIC S9(9)        USAGE IS COMP.
01      NUM-INDEXES PIC S9(9)       USAGE IS COMP.
01      INT-ARRAY.
```

(continued on next page)

Example B–8 (Cont.): VAX COBOL Sample Program

```
        05 DUMMY-INT-ARRAY OCCURS 50 TIMES PIC S9(9) USAGE IS COMP.
01      TEXT-START-X               USAGE IS COMP-1 VALUE 0.05.
01      TEXT-START-Y               USAGE IS COMP-1 VALUE 0.9.
01      LARGER                     USAGE IS COMP-1 VALUE 0.04.
01      WIDER                      USAGE IS COMP-1 VALUE 3.0.
01      MAX-WIDTH                  USAGE IS COMP-1.
01      DUMMY-REAL                 USAGE IS COMP-1.
01      NOM_WIDTH                  USAGE IS COMP-1.
01      WIDTH-RESULT               USAGE IS COMP-1.
01      ARRAY-DEFINITIONS-FLOAT    USAGE IS COMP-1.
    05  X-STAR-ARRAY.
        10  FILLER  VALUE 0.05.
        10  FILLER  VALUE 0.06.
        10  FILLER  VALUE 0.36.
        10  FILLER  VALUE 0.66.
        10  FILLER  VALUE 0.835.
        10  FILLER  VALUE 0.92.
    05  STARS-X-VALUES REDEFINES X-STAR-ARRAY.
        10  X-STAR OCCURS 6 TIMES.
    05  Y-STAR-ARRAY.
        10  FILLER  VALUE 0.7.
        10  FILLER  VALUE 0.86.
        10  FILLER  VALUE 0.81.
        10  FILLER  VALUE 0.86.
        10  FILLER  VALUE 0.701.
        10  FILLER  VALUE 0.82.
    05  STARS-Y-VALUES REDEFINES Y-STAR-ARRAY.
        10  Y-STAR OCCURS 6 TIMES.
    05 X-TREE-ARRAY.
        10 FILLER VALUE 0.425.
        10 FILLER VALUE 0.5.
        10 FILLER VALUE 0.52.
        10 FILLER VALUE 0.54.
        10 FILLER VALUE 0.6.
        10 FILLER VALUE 0.575.
        10 FILLER VALUE 0.56.
        10 FILLER VALUE 0.559.
        10 FILLER VALUE 0.64.
        10 FILLER VALUE 0.69.
        10 FILLER VALUE 0.689.
        10 FILLER VALUE 0.66.
        10 FILLER VALUE 0.63.
        10 FILLER VALUE 0.645.
        10 FILLER VALUE 0.59.
        10 FILLER VALUE 0.53.
        10 FILLER VALUE 0.48.
```

(continued on next page)

```
            10 FILLER VALUE 0.45.
            10 FILLER VALUE 0.42.
            10 FILLER VALUE 0.375.
            10 FILLER VALUE 0.35.
            10 FILLER VALUE 0.375.
            10 FILLER VALUE 0.44.
            10 FILLER VALUE 0.45.
            10 FILLER VALUE 0.515.
            10 FILLER VALUE 0.51.
            10 FILLER VALUE 0.495.
            10 FILLER VALUE 0.475.
            10 FILLER VALUE 0.425.
        05  TREE-X REDEFINES X-TREE-ARRAY.
            10 TREE-X-ITEM OCCURS 29 TIMES.
        05  Y-TREE-ARRAY.
            10 FILLER VALUE 0.28.
            10 FILLER VALUE 0.3.
            10 FILLER VALUE 0.26.
            10 FILLER VALUE 0.3.
            10 FILLER VALUE 0.28.
            10 FILLER VALUE 0.33.
            10 FILLER VALUE 0.42.
            10 FILLER VALUE 0.49.
            10 FILLER VALUE 0.53.
            10 FILLER VALUE 0.57.
            10 FILLER VALUE 0.61.
            10 FILLER VALUE 0.64.
            10 FILLER VALUE 0.66.
            10 FILLER VALUE 0.71.
            10 FILLER VALUE 0.76.
            10 FILLER VALUE 0.78.
            10 FILLER VALUE 0.75.
            10 FILLER VALUE 0.71.
            10 FILLER VALUE 0.65.
            10 FILLER VALUE 0.645.
            10 FILLER VALUE 0.6.
            10 FILLER VALUE 0.55.
            10 FILLER VALUE 0.54.
            10 FILLER VALUE 0.5.
            10 FILLER VALUE 0.5.
            10 FILLER VALUE 0.425.
            10 FILLER VALUE 0.38.
            10 FILLER VALUE 0.33.
            10 FILLER VALUE 0.28.
        05 TREE-Y REDEFINES Y-TREE-ARRAY.
            10 TREE-Y-ITEM OCCURS 29 TIMES.
```

(continued on next page)

```
05 X-HOUSE-ARRAY.
   10 FILLER VALUE 0.1.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.325.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.25.
   10 FILLER VALUE 0.25.
   10 FILLER VALUE 0.2.
   10 FILLER VALUE 0.075.
   10 FILLER VALUE 0.1.
   10 FILLER VALUE 0.1.
05 HOUSE-X REDEFINES X-HOUSE-ARRAY.
   10 HOUSE-X-ITEM OCCURS 12 TIMES.
05 Y-HOUSE-ARRAY.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.6.
   10 FILLER VALUE 0.6.
   10 FILLER VALUE 0.64.
   10 FILLER VALUE 0.75.
   10 FILLER VALUE 0.75.
   10 FILLER VALUE 0.7.
   10 FILLER VALUE 0.75.
   10 FILLER VALUE 0.6.
   10 FILLER VALUE 0.6.
   10 FILLER VALUE 0.3.
05 HOUSE-Y REDEFINES Y-HOUSE-ARRAY.
   10 HOUSE-Y-ITEM OCCURS 12 TIMES.
05 LAND-X-ARRAY.
   10 FILLER VALUE 0.0.
   10 FILLER VALUE 0.04.
   10 FILLER VALUE 0.055.
   10 FILLER VALUE 0.08.
   10 FILLER VALUE 0.1.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.375.
   10 FILLER VALUE 0.44.
   10 FILLER VALUE 0.49.
   10 FILLER VALUE 0.56.
   10 FILLER VALUE 0.68.
   10 FILLER VALUE 0.8.
   10 FILLER VALUE 0.9.
   10 FILLER VALUE 0.95.
   10 FILLER VALUE 1.0.
```

(continued on next page)

```
05 LAND-X REDEFINES LAND-X-ARRAY.
   10 LAND-X-ITEM OCCURS 15 TIMES.
05 LAND-Y-ARRAY.
   10 FILLER VALUE 0.35.
   10 FILLER VALUE 0.375.
   10 FILLER VALUE 0.376.
   10 FILLER VALUE 0.36.
   10 FILLER VALUE 0.365.
   10 FILLER VALUE 0.366.
   10 FILLER VALUE 0.38.
   10 FILLER VALUE 0.385.
   10 FILLER VALUE 0.375.
   10 FILLER VALUE 0.36.
   10 FILLER VALUE 0.38.
   10 FILLER VALUE 0.35.
   10 FILLER VALUE 0.359.
   10 FILLER VALUE 0.375.
   10 FILLER VALUE 0.385.
05 LAND-Y REDEFINES LAND-Y-ARRAY.
   10 LAND-Y-ITEM OCCURS 15 TIMES.
05 BW-X-VALUES-ARRAY.
   10 FILLER VALUE 0.0.
   10 FILLER VALUE 0.0.
   10 FILLER VALUE 0.2.
   10 FILLER VALUE 0.2.
   10 FILLER VALUE 0.25.
   10 FILLER VALUE 0.25.
   10 FILLER VALUE 1.0.
   10 FILLER VALUE 1.0.
   10 FILLER VALUE 0.0.
05 BW-X-VALUES REDEFINES BW-X-VALUES-ARRAY.
   10 BW-X-ITEM OCCURS 9 TIMES.
05 BW-Y-VALUES-ARRAY.
   10 FILLER VALUE 0.0.
   10 FILLER VALUE 0.15.
   10 FILLER VALUE 0.15.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.3.
   10 FILLER VALUE 0.15.
   10 FILLER VALUE 0.15.
   10 FILLER VALUE 0.0.
   10 FILLER VALUE 0.0.
05 BW-Y-VALUES REDEFINES BW-Y-VALUES-ARRAY.
   10 BW-Y-ITEM OCCURS 9 TIMES.
```

(continued on next page)

```
PROCEDURE DIVISION.
TOP-LEVEL.
    Perform SET-UP.
    Perform DRAW-PICTURE.
    Perform CLEAN-UP.
    STOP RUN.
***
*** Set Up the DEC GKS and Workstation environments ...
***
SET-UP.
    Call "GKS$OPEN_GKS" using
        BY DESCRIPTOR "SYS$ERROR:".

    Call "GKS$INQ_WS_CATEGORY" using
        BY REFERENCE GKS$K_WSTYPE_DEFAULT,
        BY REFERENCE ERROR-STATUS,
        BY REFERENCE CATEGORY.

*** Make sure the workstation type is valid
    If ERROR-STATUS is not equal to INQUIRY-OKAY or
       ( CATEGORY is not equal to GKS$K_WSCAT_OUTPUT and
         is not equal to GKS$K_WSCAT_OUTIN )

       Display "The Specified Workstation Type is invalid"
       Move ERROR-STATUS to ERROR-OUT
       Display ERROR-MESSAGE
       Go To EMERGENCY-EXIT.

    Call "GKS$OPEN_WS" using
        BY REFERENCE WS-ID,
        BY DESCRIPTOR GKS$K_CONID_DEFAULT,
        BY REFERENCE GKS$K_WSTYPE_DEFAULT.

    Call "GKS$ACTIVATE_WS" using
        BY REFERENCE WS-ID.

*** Make sure that the deferral mode and regeneration flag are
*** properly set
    Call "GKS$INQ_WS_TYPE" using
        BY REFERENCE WS-ID,
        BY REFERENCE ERROR-STATUS,
        BY DESCRIPTOR DUMMY-STRING,
        BY REFERENCE WS-TYPE,
        BY REFERENCE DUMMY-INTEGER.


    Call "GKS$INQ_DEF_DEFER_STATE" using
        BY REFERENCE WS-TYPE,
        BY REFERENCE ERROR-STATUS,
        BY REFERENCE DEF-MODE,
        BY REFERENCE REGEN-FLAG.
```

(continued on next page)

Example B–8 (Cont.): VAX COBOL Sample Program

```
*** You can check the status  of the inquiry function execution, as
*** follows:
    If ERROR-STATUS is not equal to INQUIRY-OKAY
       Display "The deferral inquiry caused an error"
       Move ERROR-STATUS to ERROR-OUT
       Display ERROR-MESSAGE
       Go To EMERGENCY-EXIT.

*** Defer output as long as possible and suppress implicit
*** regenerations
    If DEF-MODE is not equal to  GKS$K_ASTI and
       REGEN-FLAG is not equal to GKS$K_IRG_SUPPRESSED
        Call "GKS$SET_DEFER_STATE" using
        BY REFERENCE WS-ID,
        BY REFERENCE GKS$K_ASTI,
        BY REFERENCE GKS$K_IRG_SUPPRESSED.

***
*** Draw the picture and place each primitive in a segment
***
DRAW-PICTURE.

    Call "GKS$SET_TEXT_HEIGHT" using
        BY REFERENCE LARGER.

    Call "GKS$SET_PMARK_TYPE" using
        BY REFERENCE GKS$K_MARKERTYPE_PLUS.

    Call "GKS$SET_FILL_INT_STYLE" using
        BY REFERENCE GKS$K_INTSTYLE_SOLID.

    Call "GKS$SET_PLINE_LINETYPE" using
        BY REFERENCE GKS$K_LINETYPE_DASHED_DOTTED.

*** Obtain the workstation Type
    Call "GKS$INQ_WS_TYPE" using
        BY REFERENCE WS-ID,
        BY REFERENCE ERROR-STATUS,
        BY DESCRIPTOR DUMMY-STRING,
        BY REFERENCE WS-TYPE,
        BY REFERENCE DUMMY-INTEGER.
```

(continued on next page)

```
*** Make sure that you don't ask for a line wider than the
*** workstations widest line
    Call "GKS$INQ_PLINE_FAC" using
        BY REFERENCE WS-TYPE,
        BY REFERENCE ERROR-STATUS,
        BY REFERENCE DUMMY-INTEGER,
        BY DESCRIPTOR INT-ARRAY,
        BY REFERENCE DUMMY-INTEGER,
        BY REFERENCE NOM-WIDTH,
        BY REFERENCE DUMMY-REAL,
        BY REFERENCE MAX-WIDTH,
        BY REFERENCE DUMMY-INTEGER,
        BY REFERENCE DUMMY-INTEGER.

    Multiply WIDER by MAX-WIDTH giving WIDTH-RESULT.
    Perform Until WIDTH-RESULT is Less Than MAX-WIDTH
        SUBTRACT 0.1 FROM WIDER
        Multiply WIDER by MAX-WIDTH giving WIDTH-RESULT
        End-Perform.
    Call "GKS$SET_PLINE_LINEWIDTH" using
        BY REFERENCE WIDER.

    Call "GKS$CREATE_SEG" using
        BY REFERENCE TITLE.
    Call "GKS$TEXT" using
        BY REFERENCE TEXT-START-X,
        BY REFERENCE TEXT-START-Y,
        BY DESCRIPTOR "Starry Night".
    Call "GKS$CLOSE_SEG".

    Call "GKS$CREATE_SEG" using
        BY REFERENCE STARS.
    Call "GKS$POLYMARKER" using
        BY REFERENCE NUM-STARS,
        BY REFERENCE STARS_X_VALUES,
        BY REFERENCE STARS_Y_VALUES.
    Call "GKS$CLOSE_SEG".


    Call "GKS$CREATE_SEG" using
        BY REFERENCE TREE.
    Call "GKS$FILL_AREA" using
        BY REFERENCE NUM-TREE-PTS,
        BY REFERENCE TREE-X,
        BY REFERENCE TREE-Y.
    Call "GKS$CLOSE_SEG".
```

(continued on next page)

```
*** Check to see if working with a color workstation
    Call "GKS$INQ_COLOR_FAC" using
        BY REFERENCE WS-TYPE,
        BY REFERENCE ERROR-STATUS,
        BY REFERENCE DUMMY-INTEGER,
        BY REFERENCE COLOR-FLAG,
        BY REFERENCE NUM-INDEXES.

*** For all workstations with less than three color indexes,
*** USE GKS$FILL_AREA instead of GKS$CELL_ARRAY for the sidewalk
*** and road
    If NUM-INDEXES is less than THREE
        Call "GKS$CREATE_SEG" using
            BY REFERENCE SIDE
        Call "GKS$SET_FILL_INT_STYLE" using
            BY REFERENCE GKS$K_INTSTYLE_HATCH
        Call "GKS$FILL_AREA" using
            BY REFERENCE BW_NUM_PTS,
            BY REFERENCE BW_X_VALUES,
            BY REFERENCE BW_Y_VALUES
        Call "GKS$SET_FILL_INT_STYLE" using
            BY REFERENCE GKS$K_INTSTYLE_SOLID
        Call "GKS$CLOSE_SEG"


    Else
***  Build a Descriptor of the Side Walk Color array
        Call "BUILDESC" using
            BY REFERENCE ARRAY_D,
            BY DESCRIPTOR SIDE_COLORS(1,1),
            BY VALUE SIDE_NUM_COL,
            BY VALUE SIDE_NUM_ROW
        Call "GKS$CREATE_SEG" using
            BY REFERENCE SIDE
        Call "GKS$CELL_ARRAY" using
            BY REFERENCE SIDE_START_X,
            BY REFERENCE SIDE_START_Y,
            BY REFERENCE SIDE_DIAG_X,
            BY REFERENCE SIDE_DIAG_Y,
            BY REFERENCE SIDE_OFF_COL,
            BY REFERENCE SIDE_OFF_ROW,
            BY REFERENCE SIDE_NUM_COL
            BY REFERENCE SIDE_NUM_ROW,
            BY REFERENCE ARRAY_D
        Call "GKS$CLOSE_SEG"
```

(continued on next page)

```
***  Build a descriptor of the road color array
        Call "BUILDESC" using
            BY REFERENCE ARRAY_D,
            BY DESCRIPTOR ROAD_COLORS(1,1),
            BY VALUE ROAD_NUM_COL,
            BY VALUE ROAD_NUM_ROW
        Call "GKS$CREATE_SEG" using
            BY REFERENCE ROAD
        Call "GKS$CELL_ARRAY" using
            BY REFERENCE ROAD_START_X,
            BY REFERENCE ROAD_START_Y,
            BY REFERENCE ROAD_DIAG_X,
            BY REFERENCE ROAD_DIAG_Y,
            BY REFERENCE ROAD_OFF_COL,
            BY REFERENCE ROAD_OFF_ROW,
            BY REFERENCE ROAD_NUM_COL
            BY REFERENCE ROAD_NUM_ROW,
            BY REFERENCE ARRAY_D
        Call "GKS$CLOSE_SEG".


    Call "GKS$CREATE_SEG" using
        BY REFERENCE HORIZON.
    Call "GKS$POLYLINE" using
        BY REFERENCE NUM-LAND-PTS,
        BY REFERENCE LAND-X,
        BY REFERENCE LAND-Y.
    Call "GKS$CLOSE_SEG".

    Call "GKS$CREATE_SEG" using
        BY REFERENCE HOUSE.
    If NUM-INDEXES is Greater than THREE or equal to THREE
        Call" GKS$SET_FILL_COLOR_INDEX" using
            BY REFERENCE DARK.
    Call "GKS$FILL_AREA" using
        BY REFERENCE NUM-HOUSE-PTS,
        BY REFERENCE HOUSE-X,
        BY REFERENCE HOUSE-Y.
    Call "GKS$CLOSE_SEG".

***
***  Clean up the DEC GKS and workstation Environments
```

(continued on next page)

Example B–8 (Cont.): VAX COBOL Sample Program

```
CLEAN-UP.
    Call "GKS$UPDATE_WS" using
        BY REFERENCE WS-ID,
        BY REFERENCE ONE.
    Accept DUMMY-STRING.
    Call "GKS$DEACTIVATE_WS" using
        BY REFERENCE WS-ID.
    Call "GKS$CLOSE_WS" using
        BY REFERENCE WS-ID.
    Call "GKS$CLOSE_GKS".

***
*** Stop the program if we get an error

EMERGENCY-EXIT.
    STOP RUN.
```

B.9 VAX BLISS

Example B–9 presents the Starry Night program written in VAX BLISS.

Example B–9: VAX BLISS Sample Program

```
%TITLE 'SAMPLE - DEC GKS Sample Program, coded in BLISS-32'
MODULE SAMPLE (                                 ! DEC GKS Sample Program
                IDENT = '1-001',                ! File: SAMPLE.B32
                MAIN = SAMPLE
                ) =
BEGIN
%SBTTL 'Declarations'
!
! SWITCHES:
!

SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL, NONEXTERNAL = WORD_RELATIVE);

!
! LINKAGES:
!
!       NONE
!
! TABLE OF CONTENTS:
!
```

(continued on next page)

```
FORWARD ROUTINE
    SAMPLE;                                         ! The sample program

!
! INCLUDE FILES:
!

LIBRARY 'SYS$LIBRARY:STARLET';                      ! System symbols

REQUIRE 'SYS$LIBRARY:GKSDEFS.R32';                  ! DEC GKS symbols


!
! MACROS:
!
!+
! Macro to declare PSECTs for a facility, given the facility prefix.
! The declarations are very dependent on the linker algorithm for
! sorting PSECTs.  Currently that algorithm divides PSECTs into four
! groups depending on WRITE vs NOWRITE and EXECUTE vs NOEXECUTE.
! Therefore in order to get compact programs, PLIT is made EXECUTABLE
! to get it close to CODE.
!
!       Example of use:
!
! PSECT DECLARATIONS:
!
!       DECLARE_PSECTS (FOR);   ! Declare PSECTs for FOR$ facility
!
! Note: since the methodology manual does not yet specify where PSECT
! declarations go in a module, they are put between EQUATED SYMBOLS
! and OWN STORAGE (which is after INCLUDE files).
!-

MACRO
    DECLARE_PSECTS (FAC) =
    PSECT
    CODE = %NAME ('_', FAC, $CODE) (READ, NOWRITE, EXECUTE, SHARE, PIC,
        ADDRESSING_MODE (WORD_RELATIVE)),
    PLIT = %NAME ('_', FAC, $CODE) (READ, NOWRITE, EXECUTE, SHARE, PIC,
        ADDRESSING_MODE (WORD_RELATIVE)),
    OWN  = %NAME ('_', FAC, $DATA) (READ, WRITE, NOEXECUTE, NOSHARE, PIC,
        ADDRESSING_MODE (LONG_RELATIVE)),
    GLOBAL = %NAME ('_', FAC, $DATA) (READ, WRITE, NOEXECUTE, NOSHARE, PIC,
        ADDRESSING_MODE (LONG_RELATIVE)) %;
```

(continued on next page)

```
!+
! Define macro for declaring PIC (position independent) dispatch tables
! as OWN storage (would be better if BIND table = PLIT (...), however,
! BLISS doesn't allow table to be referenced inside PLIT definition,
! so use OWN storage instead).  The OWN storage is temporarily defined
! to be same PSECT as code, then DECLARE_PSECTS should be called again
! to restore OWN to _fac$DATA PSECT.
!-

MACRO
    DISPATCH_PSECTS (FAC) =
    PSECT
    CODE = %NAME ('_', FAC, $CODE) (READ, NOWRITE, EXECUTE, SHARE, PIC,
        ADDRESSING_MODE (WORD_RELATIVE)),
    PLIT = %NAME ('_', FAC, $CODE) (READ, NOWRITE, EXECUTE, SHARE, PIC,
        ADDRESSING_MODE (WORD_RELATIVE)),
    OWN  = %NAME ('_', FAC, $CODE) (READ, NOWRITE, EXECUTE, SHARE, PIC,
        ADDRESSING_MODE (WORD_RELATIVE)),
    GLOBAL = %NAME ('_', FAC, $CODE) (READ, NOWRITE, EXECUTE, SHARE, PIC,
        ADDRESSING_MODE (WORD_RELATIVE)) %;

!
! EQUATED SYMBOLS:
!
!       NONE
!
! FIELDS:
!
!       NONE
!
! STRUCTURES:
!
!       NONE
!
! PSECTS:
!
DECLARE_PSECTS (GKS);                   ! Declare PSECTs for GKS$ facility
!
! OWN STORAGE:
!
!       NONE
!
! EXTERNAL REFERENCES:
!
```

(continued on next page)

```
EXTERNAL ROUTINE
    STR$COPY_DX,                                ! Copy a string, by descriptor
    STR$CONCAT,                                 ! Concatenate strings
    LIB$GET_INPUT,                              ! Get a line from SYS$INPUT
    STR$COPY_R,                                 ! Copy a string, by reference
    STR$FREE1_DX,                               ! Free a dynamic string
    LIB$GET_VM,                                 ! Get virtual memory
    LIB$FREE_VM,                                ! Free virtual memory
    LIB$PUT_OUTPUT;                             ! Write a line on SYS$OUTPUT:

%SBTTL 'Package of macros for string processing'
!+
! Macro to initialize a dynamic descriptor.
!-

MACRO
    INIT_DESCRIPTOR (DESCR) =
        DESCR [DSC$W_LENGTH] = 0;
        DESCR [DSC$B_DTYPE] = DSC$K_DTYPE_T;
        DESCR [DSC$B_CLASS] = DSC$K_CLASS_D;
        DESCR [DSC$A_POINTER] = 0;
    %,
!+
! Macro to discard a dynamic descriptor.
!-
    DISCARD_DESCRIPTOR (DESCR) =
        BEGIN

        LOCAL
            FREE_STATUS;

        FREE_STATUS = STR$FREE1_DX (DESCR);

        IF ( NOT .FREE_STATUS) THEN SIGNAL_STOP (.FREE_STATUS);

        END;
    %,
!+
! Macro to build a text line using FAO.  This is a convenience macro.
!-
    BUILD_TEXT_LINE (DESCR, CTL_STRING, FAO_ARGS) =
        BEGIN

        LOCAL
            FAO_STATUS,
            COPY_STATUS;
```

(continued on next page)

```
        CTL_STR_DSC [DSC$W_LENGTH] = %CHARCOUNT (CTL_STRING);
        CTL_STR_DSC [DSC$B_DTYPE] = DSC$K_DTYPE_T;
        CTL_STR_DSC [DSC$B_CLASS] = DSC$K_CLASS_S;
        CTL_STR_DSC [DSC$A_POINTER] = CH$PTR (UPLIT (CTL_STRING));
        FAO_STATUS = $FAO (                 !
            CTL_STR_DSC,                    !
            OUT_LENGTH,                     !
            TEMP_STR_DSC,                   !
            %REMOVE (FAO_ARGS));

        IF ( NOT .FAO_STATUS) THEN SIGNAL_STOP (.FAO_STATUS);

        COPY_STATUS = STR$COPY_R (DESCR, OUT_LENGTH,
            .TEMP_STR_DSC [DSC$A_POINTER]);
        .COPY_STATUS
        END
    %,
!+
! Macro to format and print a line.  Errors are returned to the caller.
! This is a convenience macro.
!-
    PRINT_LINE (TEXT, VARS) =
        BEGIN

        LOCAL
            BUILD_STATUS,
            PRINT_STATUS;

        BUILD_STATUS = BUILD_TEXT_LINE (LINE_DESC, %STRING (%REMOVE (TEXT)),
            VARS);

        IF ( NOT .BUILD_STATUS) THEN RETURN (.BUILD_STATUS);

        PRINT_STATUS = LIB$PUT_OUTPUT (LINE_DESC);

        IF ( NOT .PRINT_STATUS) THEN RETURN (.PRINT_STATUS);

        END
    %;


%SBTTL 'SAMPLE - DEC GKS Sample Program'
ROUTINE SAMPLE                              ! DEC GKS Sample Program
    =
```

(continued on next page)

```
!++
! FUNCTIONAL DESCRIPTION:
!
!       This routine uses DEC GKS to display the sample picture.
!
! CALLING SEQUENCE:
!
!       ret_status.wlc.v = SAMPLE ()
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! COMPLETION STATUS:
!
!       SS$_NORMAL     Normal successful completion
!       Any error from calling the VMS RTL.
!
! SIDE EFFECTS:
!
!       Does output using DEC GKS.
!       Waits for the picture to be admired before returning.
!
!--

    BEGIN

    LOCAL
!+
! Stuff for BUILD_TEXT_LINE
!-
        CTL_STR_DSC : BLOCK [8, BYTE],
        TEMP_STR_DSC : BLOCK [8, BYTE],
        TEMP_STRING : VECTOR [132, BYTE],
        OUT_LENGTH,
```

(continued on next page)

```
!+
! Stuff for PRINT_LINE
!-
        LINE_DESC : BLOCK [8, BYTE],
!+
! End of stuff for PRINT_LINE
!-
        ERROR_STATUS,
        CATEGORY,
        INQUIRE_OKAY,
        DUMMY_INTEGER,
        DEF_MODE,
        REGEN_FLAG,
        WS_TYPE,
        ERROR_FILE : BLOCK [8, BYTE],
        DUMMY_DSC : BLOCK [8, BYTE],
        WS_ID;

    LABEL
        SET_UP,
        DRAW_PICTURE,
        CLEAN_UP;

!+
! Set up TEMP_STR_DSC for BUILD_TEXT_LINE
!-
    TEMP_STR_DSC [DSC$W_LENGTH] = 132;
    TEMP_STR_DSC [DSC$B_DTYPE] = DSC$K_DTYPE_T;
    TEMP_STR_DSC [DSC$B_CLASS] = DSC$K_CLASS_S;
    TEMP_STR_DSC [DSC$A_POINTER] = CH$PTR (TEMP_STRING);
!+
! Set up LINE_DESC for PRINT_LINE
!-
    INIT_DESCRIPTOR (LINE_DESC);
!+
! Initialize DEC GKS.  Error messages to SYS$ERROR:.
!-
SET_UP :
    BEGIN
    ERROR_FILE [DSC$W_LENGTH] = %CHARCOUNT ('SYS$ERROR:');
    ERROR_FILE [DSC$B_DTYPE] = DSC$K_DTYPE_T;
    ERROR_FILE [DSC$B_CLASS] = DSC$K_CLASS_S;
    ERROR_FILE [DSC$A_POINTER] = UPLIT ('SYS$ERROR:');
    GKS$OPEN_GKS (ERROR_FILE);
```

(continued on next page)

```
!+
! Make sure that this is a suitable workstation.
!-
    GKS$INQ_WS_CATEGORY (GKS$K_WSTYPE_DEFAULT, ERROR_STATUS, CATEGORY);

    IF ((.ERROR_STATUS NEQ 0) OR
        ((.CATEGORY NEQ GKS$K_WSCAT_OUTIN) AND (.CATEGORY NEQ GKS$K_WSCAT_MO)))
    THEN
        BEGIN
        PRINT_LINE (<'The specified workstation type is invalid'>, <' '>);
        PRINT_LINE (<'Error status: !SL'>, <.ERROR_STATUS>);
        RETURN (SS$_NORMAL);
        END;

    INIT_DESCRIPTOR (DUMMY_DSC);
    WS_ID = 1;
    GKS$OPEN_WS (WS_ID, DUMMY_DSC, %REF (GKS$K_WSTYPE_DEFAULT));
    GKS$ACTIVATE_WS (WS_ID);
!+
! Make sure the deferral mode and regeneration flags are properly set.
!-
    GKS$INQ_WS_TYPE (WS_ID, ERROR_STATUS, DUMMY_DSC, WS_TYPE, DUMMY_INTEGER);
    GKS$INQ_DEF_DEFER_STATE (WS_TYPE, ERROR_STATUS, DEF_MODE, REGEN_FLAG);
!+
! Check the status of the inquiry function execution.
!-

    IF (.ERROR_STATUS NEQ 0)
    THEN
        BEGIN
        PRINT_LINE (<'The deferral inquiry caused an error'>, <' '>);
        PRINT_LINE (<'Error status: !SL'>, <.ERROR_STATUS>);
        RETURN (SS$_NORMAL);
        END;


!+
! Defer output as long as possible and suppress implicit regenerations.
!-

    IF ((.DEF_MODE NEQ GKS$K_ASTI) AND (.REGEN_FLAG NEQ GKS$K_IRG_SUPPRESSED))
    THEN
        BEGIN
        GKS$SET_DEFER_STATE (WS_ID, %REF (GKS$K_ASTI),
            %REF (GKS$K_IRG_SUPPRESSED));
        END;

    END;
```

(continued on next page)

```
!+
! Draw the picture, placing each primitive in a segment.
!-
DRAW_PICTURE :
    BEGIN

    LOCAL
        WIDER,
        NOM_WIDTH,
        MAX_WIDTH,
        DUMMY_REAL,
        NUM_INDEXES,
        TITLE_DSC : BLOCK [8, BYTE],
        INTEGER_DSC : BLOCK [20, BYTE],
        DUMMY_INTEGER_ARRAY : VECTOR [50],
        SIDE_ARRAY_DSC : BLOCK [44, BYTE],
        SIDE_COLORS : REF VECTOR [2],
        ROAD_ARRAY_DSC : BLOCK [44, BYTE],
        ROAD_COLORS : REF VECTOR [10],
        COLOR_FLAG,
        DSC_PTR : REF VECTOR;

    INTEGER_DSC [DSC$W_LENGTH] = 4;
    INTEGER_DSC [DSC$B_DTYPE] = DSC$K_DTYPE_L;
    INTEGER_DSC [DSC$B_CLASS] = DSC$K_CLASS_A;
    INTEGER_DSC [DSC$A_POINTER] = DUMMY_INTEGER_ARRAY [0];
    INTEGER_DSC [DSC$B_SCALE] = 0;
    INTEGER_DSC [DSC$B_DIGITS] = 0;
    INTEGER_DSC [DSC$B_AFLAGS] = 0;
    INTEGER_DSC [DSC$B_DIMCT] = 1;
    INTEGER_DSC [DSC$L_ARSIZE] = 50*4;
    INTEGER_DSC [DSC$A_A0] = DUMMY_INTEGER_ARRAY [0];
```

(continued on next page)

```
!
    SIDE_COLORS = UPLIT (2, 3);
    SIDE_ARRAY_DSC [DSC$W_LENGTH] = 4;
    SIDE_ARRAY_DSC [DSC$B_DTYPE] = DSC$K_DTYPE_L;
    SIDE_ARRAY_DSC [DSC$B_CLASS] = DSC$K_CLASS_A;
    SIDE_ARRAY_DSC [DSC$A_POINTER] = SIDE_COLORS [0];
    SIDE_ARRAY_DSC [DSC$B_SCALE] = 0;
    SIDE_ARRAY_DSC [DSC$B_DIGITS] = 0;
    SIDE_ARRAY_DSC [DSC$B_AFLAGS] = 0;
    SIDE_ARRAY_DSC [DSC$V_FL_COEFF] = 1;
    SIDE_ARRAY_DSC [DSC$V_FL_BOUNDS] = 1;
    SIDE_ARRAY_DSC [DSC$B_DIMCT] = 2;
    SIDE_ARRAY_DSC [DSC$L_ARSIZE] = 1*2*4;
    SIDE_ARRAY_DSC [DSC$A_A0] = SIDE_COLORS [0];
    DSC_PTR = SIDE_ARRAY_DSC [DSC$L_M1];
    DSC_PTR [0] = 1;                             ! M1
    DSC_PTR [1] = 2;                             ! M2
    DSC_PTR [2] = 0;                             ! L1
    DSC_PTR [3] = 0;                             ! U1
    DSC_PTR [4] = 0;                             ! L2
    DSC_PTR [5] = 1;                             ! U2
!
    ROAD_COLORS = UPLIT (2, 3, 2, 3, 2, 3, 2, 3, 2, 3);
    ROAD_ARRAY_DSC [DSC$W_LENGTH] = 4;
    ROAD_ARRAY_DSC [DSC$B_DTYPE] = DSC$K_DTYPE_L;
    ROAD_ARRAY_DSC [DSC$B_CLASS] = DSC$K_CLASS_A;
    ROAD_ARRAY_DSC [DSC$A_POINTER] = ROAD_COLORS [0];
    ROAD_ARRAY_DSC [DSC$B_SCALE] = 0;
    ROAD_ARRAY_DSC [DSC$B_DIGITS] = 0;
    ROAD_ARRAY_DSC [DSC$B_AFLAGS] = 0;
    ROAD_ARRAY_DSC [DSC$V_FL_COEFF] = 1;
    ROAD_ARRAY_DSC [DSC$V_FL_BOUNDS] = 1;
    ROAD_ARRAY_DSC [DSC$B_DIMCT] = 2;
    ROAD_ARRAY_DSC [DSC$L_ARSIZE] = 10*1*4;
    ROAD_ARRAY_DSC [DSC$A_A0] = ROAD_COLORS [0];
    DSC_PTR = ROAD_ARRAY_DSC [DSC$L_M1];
    DSC_PTR [0] = 10;                            ! M1
    DSC_PTR [1] = 1;                             ! M2
    DSC_PTR [2] = 0;                             ! L1
    DSC_PTR [3] = 9;                             ! U1
    DSC_PTR [4] = 0;                             ! L2
    DSC_PTR [5] = 0;                             ! U2
    GKS$SET_TEXT_HEIGHT (%REF (%E'0.04'));
    GKS$SET_PMARK_TYPE (%REF (GKS$K_MARKERTYPE_PLUS));
    GKS$SET_FILL_INT_STYLE (%REF (GKS$K_INTSTYLE_SOLID));
    GKS$SET_PLINE_LINETYPE (%REF (GKS$K_LINETYPE_DASHED_DOTTED));
```

(continued on next page)

Example B–9 (Cont.): VAX BLISS Sample Program

```
!+
! Obtain the workstation type.
!-
    GKS$INQ_WS_TYPE (WS_ID, ERROR_STATUS, DUMMY_DSC, WS_TYPE, DUMMY_INTEGER);
!+
! Don't ask for too wide a line.
!-
    GKS$INQ_PLINE_FAC (WS_TYPE, ERROR_STATUS, DUMMY_INTEGER, INTEGER_DSC,
        DUMMY_INTEGER, NOM_WIDTH, DUMMY_REAL, MAX_WIDTH, DUMMY_INTEGER,
        DUMMY_INTEGER);
    BEGIN

    LOCAL
        TEMP;

    BUILTIN
        MULF,
        CMPF,
        SUBF;

    WIDER = %E'3.0';

    WHILE (MULF (WIDER, NOM_WIDTH, TEMP); CMPF (TEMP, MAX_WIDTH) GTR 0) DO
        BEGIN
        SUBF (WIDER, %E'0.1', WIDER);
        END;

    END;
    GKS$SET_PLINE_LINEWIDTH (WIDER);
    GKS$CREATE_SEG (%REF (1));                    ! Title
    TITLE_DSC [DSC$W_LENGTH] = %CHARCOUNT ('Starry Night');
    TITLE_DSC [DSC$B_DTYPE] = DSC$K_DTYPE_T;
    TITLE_DSC [DSC$B_CLASS] = DSC$K_CLASS_S;
    TITLE_DSC [DSC$A_POINTER] = UPLIT ('Starry Night');
    GKS$TEXT (%REF (%E'0.05'), %REF (%E'0.9'), TITLE_DSC);
    GKS$CLOSE_SEG ();
    GKS$CREATE_SEG (%REF (2));                    ! Stars
    GKS$POLYMARKER (%REF (6),
        UPLIT (%E'0.05', %E'0.06', %E'0.36', %E'0.66', %E'0.835', %E'0.92'),
        UPLIT (%E'0.7', %E'0.86', %e'0.81', %E'0.86', %E'0.701', %E'0.82'));
    GKS$CLOSE_SEG ();
```

(continued on next page)

```
    GKS$CREATE_SEG (%REF (3));                    ! Tree
    GKS$FILL_AREA (%REF (29),
        UPLIT (%E'0.425', %E'0.5', %E'0.52', %E'0.54', %E'0.6', %E'0.575',
        %E'0.56', %E'0.559', %E'0.64', %E'0.69', %E'0.689', %E'0.66', %E'0.63',
        %E'0.645', %E'0.59', %E'0.53', %E'0.48', %E'0.45', %E'0.42', %E'0.375',
        %E'0.35', %E'0.375', %E'0.44', %E'0.45', %E'0.515', %E'0.51', %E'0.495',
        %E'0.475', %E'0.425'),
        UPLIT (%E'0.28', %E'0.3', %E'0.26', %E'0.3', %E'0.28', %E'0.33',
        %E'0.42', %E'0.49', %E'0.53', %E'0.57', %E'0.61', %E'0.64', %E'0.66',
        %E'0.71', %E'0.76', %E'0.78', %E'0.75', %E'0.71', %E'0.65', %E'0.645',
        %E'0.6', %E'0.55', %E'0.54', %E'0.5', %E'0.5', %E'0.425', %E'0.38',
        %E'0.33', %E'0.28'));
    GKS$CLOSE_SEG ();
!+
! Check for a color workstation.
!-
    GKS$INQ_COLOR_FAC (WS_TYPE, ERROR_STATUS, DUMMY_INTEGER, COLOR_FLAG,
        NUM_INDEXES);
!+
! For all workstations with less than 3 color indexes,
! use GKS$FILL_AREA instead of GKS$CELL_ARRAY for the sidewalk
! and road.
!-

    IF (.COLOR_FLAG NEQ GKS$K_COLOR)
    THEN
        BEGIN
        GKS$CREATE_SEG (%REF (4));                ! Side
        GKS$SET_FILL_INT_STYLE (%REF (GKS$K_INTSTYLE_HATCH));
        GKS$FILL_AREA (%REF (9),
            UPLIT (%E'0.0', %E'0.0', %E'0.2', %E'0.2', %E'0.25', %E'0.25',
            %E'1.0', %E'1.0', %E'0.0'),
            UPLIT (%E'0.0', %E'0.15', %E'0.15', %E'0.3', %E'0.3', %E'0.15',
            %E'0.15', %E'0.0', %E'0.0'));
        GKS$SET_FILL_INT_STYLE (%REF (GKS$K_INTSTYLE_SOLID));
        GKS$CLOSE_SEG ();
        END

    ELSE
        BEGIN
        GKS$CREATE_SEG (%REF (4));                ! Side
        GKS$CELL_ARRAY (%REF (%E'0.2'), %REF (%E'0.3'), %REF (%E'0.25'),
            %REF (%E'0.15'), %REF (0), %REF (0), %REF (1), %REF (2),
            SIDE_ARRAY_DSC);
        GKS$CLOSE_SEG ();
        GKS$CREATE_SEG (%REF (5));                ! Road
        GKS$CELL_ARRAY (%REF (%E'0.0'), %REF (%E'0.15'), %REF (%E'1.0'),
            %REF (%E'0.0'), %REF (0), %REF (0), %REF (10), %REF (1),
            ROAD_ARRAY_DSC);
        GKS$CLOSE_SEG ();
        END;
```

(continued on next page)

```
    GKS$CREATE_SEG (%REF (6));                  ! Horizon
    GKS$POLYLINE (%REF (15),
        UPLIT (%E'0.0', %E'0.04', %E'0.055', %E'0.08', %E'0.1', %E'0.3',
        %E'0.375', %E'0.44', %E'0.49', %E'0.56', %E'0.68', %E'0.8', %E'0.9',
        %E'0.95', %E'1.0'),
        UPLIT (%E'0.35', %E'0.375', %E'0.376', %E'0.36', %E'0.365', %E'0.366',
        %E'0.38', %E'0.385', %E'0.375', %E'0.36', %E'0.38', %E'0.35', %E'0.359',
        %E'0.375', %E'0.385'));
    GKS$CLOSE_SEG ();
    GKS$CREATE_SEG (%REF (7));                  ! House
!+
! Only change the color index if working with a workstation
! with more than two color indexes.
!-
    IF (.COLOR_FLAG EQL GKS$K_COLOR)
    THEN
        BEGIN
        GKS$SET_FILL_COLOR_INDEX (%REF (3));
        END;

    GKS$FILL_AREA (%REF (12),
        UPLIT (%E'0.1', %E'0.3', %E'0.3', %E'0.325', %E'0.3', %E'0.3', %E'0.25',
        %E'0.25', %E'0.2', %E'0.075', %E'0.1', %E'0.1'),
        UPLIT (%E'0.3', %E'0.3', %E'0.6', %E'0.6', %E'0.64', %E'0.75', %E'0.75',
        %E'0.7', %E'0.75', %E'0.6', %E'0.6', %E'0.3'));
    GKS$CLOSE_SEG ();
    END;

CLEAN_UP :
    BEGIN
    GKS$UPDATE_WS (WS_ID, %REF (GKS$K_PERFORM_FLAG));
    LIB$GET_INPUT (DUMMY_DSC);
    GKS$DEACTIVATE_WS (WS_ID);
    GKS$CLOSE_WS (WS_ID);
    GKS$CLOSE_GKS ();
    END;
    DISCARD_DESCRIPTOR (LINE_DESC);
    RETURN (SS$_NORMAL);
    END;                                        ! of routine SAMPLE
END                                             ! End of module SAMPLE
```

Index

Reader's Comments

DEC GKS User Manual
AA–HW45C–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:

Page Description

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________________

______________________ Phone __________

-- Do Not Tear - Fold Here and Tape --------------------------------

digital™

No Postage
Necessary
if Mailed
in the
United States

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

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here --------------------------------

Cut Along Dotted Line

Reader's Comments

DEC GKS User Manual
AA–HW45C–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:

Page Description

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________________

______________________ Phone __________

Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

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

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line