VAXstation Native Graphics Procedures

Order Number AA-AG30A-TE

June 1984

This document describes the five native graphics procedures in the VAXstation Display Management Library.

SUPERSESSION/UPDATE INFORMATION:	New document
OPERATING SYSTEM AND VERSION:	VAX/VMS Version 3.4 (or greater)
SOFTWARE: DECLIT AA VAX AG30A	VAXstation Version 1.0

digital equipment corporation • marlboro, massachusetts

First Printing, June 1984

Printed in the U.S.A.

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

Contents

Chapter 4 Printing Text

Chapter 5 Filling an Area

Chapter 6 Flooding an Area

Appendix A Hardware Model Summary

Appendix B Example Program

Glossary

Index

Examples

Figures

Tables

Preface

The VAXstation Display System implements five basic output operations in hardware and firmware:

- Copying an area
- Drawing a curve
- Printing text
- Filling an area
- Flooding an area

Five procedures in the VAXstation Display Management Library (VSTA) provide direct access to these operations in the context of virtual displays, pasteboards, and windows. This manual documents these five "native graphics" procedures.

The native graphics procedures are not documented with the rest of the library in order to limit their use. The VAXstation Display System provides a "soft" interface to the VAX, that is, it is defined in the loadable firmware. This interface is newly developed, and it will be enhanced and tuned, for example, change the partitioning of functions between the VAX and the display processor. Therefore, customers should limit their use of this interface.

> The software interface defined in this manual may change in future releases of the VAXstation Display Management Library and VAXstation firmware.

The following guidelines are recommended:

- Use the standard VAXstation Display Management Library and VAXstation CORE Graphics Library procedures whenever possible.
- Isolate use of the native graphics procedures in your software so that changing them will entail less effort.

Audience

This manual is intended for graphics applications programmers who are familiar with the general concepts of bitmap graphics. Readers should also be familiar with the VAX/VMS operating system, at least one high-level language, and the standard procedures in the VAXstation Display Management Library and the VAXstation CORE Graphics Library.

Related Documents

The documentation set for the VAXstation Display System software consists of:

- *Programming for the VAXstation Display System*
  Order Number AA-P153A-TE

  Describes the procedures in the VAXstation Display Management Library and the VAXstation CORE Graphics Library.

- *VAXstation User's Guide*
  Order Number AA-N660A-TE

  Introduces users to the VAXstation Display System.

- *VAXstation Software Installation Guide*
  Order Number AA-N661A-TE

  Describes the installation procedure for the VAXstation software.

1

Introduction

The five native graphics procedures in the VAXstation Display Management Library provide access to the firmware in the VAXstation Display System in a form appropriate for programs written in high-level languages.

The VAXstation Display System

The VAXstation Display System is a display processor consisting of a controlling microprocessor with its private instruction and data memory, a display monitor, a frame buffer memory from which the monitor is refreshed, and an interface to memory in the host computer. The system also has additional memory for staging graphics operations or caching commonly used images, fonts, halftones, and so on.

Mapped into the display processor's address space are the processor's private memory, the frame buffer memory, and some section of host memory. The display processor can perform operations on data residing in any of these memories. Thus the display system processor can operate on bitmaps stored in host memory as well as in local display system memory.

Model of the Display Processor

At the most basic level, the display processor is a machine that inputs a source and uses it to modify a destination in a specified way. However, in the most general case, only certain pixels in the source are used; and only certain pixels in the destination are available for modification.

Figure 1–1 illustrates this machine.

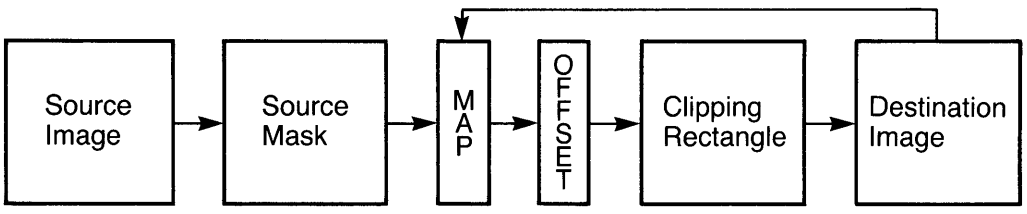


Figure 1–1. Model of Display Processor

This machine is logically divided into three parts: the source, the destination, and the map. The source and destination are always required; the map is optional.

- Source. The source image and source mask define the source. The source image provides pixel values; the source mask determines which pixels in the source image are used to update the destination.
- Destination. The destination image, clipping rectangles, and destination offset define the destination. Pixel values in the destination image are modified. Clipping rectangles can be used to include only those pixels in the rectangle, and thus to restrict the pixels that can be modified. The destination offset can change the location of the area to be modified relative to the source.
- Map. In the simplest case, no map is used. The values of the destination pixels are replaced by the values of the source pixels. The map allows destination pixels to be replaced with logical functions of the source pixel values or logical functions of the source and destination pixel values.

Display Processor Operations

The display processor instructions perform five output operations:

- Copying an area from a source to a destination
- Drawing a curve
- Printing text
- Filling an area
- Flooding an area

Native Graphics Procedures

The five native graphics procedures that implement the five display system output instructions are:

- VSTA$COPY_AREA
- VSTA$DRAW_CURVE
- VSTA$PRINT_TEXT
- VSTA$FILL_AREA
- VSTA$FLOOD_AREA

The parameters to these procedures provide the information required by the display processor. The procedures build command packets for the display processor and queue the packets for output.

The five native graphics procedures differ from the actual display processor instructions in two ways:

- The display processor instructions operate only on bitmaps, including the frame buffer from which the screen is refreshed. The procedures operate on virtual displays or program-controlled bitmaps, but do not directly modify the frame buffer.

- The display processor itself can be used with different host computers and different operating systems. The VAXstation Display System (which includes the VAXstation software as well as the display processor) operates only with a VAX host running VMS. The native graphics procedures, therefore, follow VAX/VMS conventions for subroutine libraries and use the standard VAX/VMS methods for synchronizing I/O.

Virtual Displays as Source and Destination

The VAXstation software implements a human interface in which the person using the system can have several active applications and can determine what is actually visible on the screen at any time. The physical screen is thus shared by applications, none of which write directly to the screen.

Application programs write instead to virtual displays, which are created and controlled with procedures in the standard VAXstation Display Management Library.

Virtual displays can also be either the source or the destination in the native graphics procedures.

Bitmaps as Source and Destination

A bitmap is an array of pixels, each of which represents an addressable point on the screen and has a value which determines the color or brightness of the screen at that point.

A bitmap has three dimensions: width, height, and depth. The width and height correspond to the horizontal and vertical dimensions of the bitmap and determine the Cartesian coordinates of each pixel. Each pixel is addressed by its horizontal (x) and vertical (y) coordinates. The depth of a bitmap represents the number of bits used for each pixel value. A bitmap whose depth is one has one bit per pixel. Therefore, each pixel can have one of two values — zero or one. A bitmap whose depth is two has two bits per pixel, and the values of the pixels range from zero to three. The number of values each pixel can assume determines the number of colors or number of gray shades that can appear on the screen at any one time.

The maximum number of bits per pixel is fixed for a given hardware implementation. The VAXstation 100, for example, has one bit per pixel. Since both the display processor instructions and the native graphics procedures are designed for use with several display processors, the depth of a bitmap is specified in the data structure.

Application programs can define bitmap data structures in their address space. These data structures can be either the source image or the destination image for the native graphics procedures.

Using bitmaps differs from using virtual displays in the following ways:

- Method of referencing. Virtual displays are referenced by a channel number; bitmaps are referenced by a bitmap descriptor.
- Address space. Virtual displays are maintained in the VAXstation ACP address space; bitmaps are maintained in the program's address space.

- Overlapping. Virtual displays never overlap; bitmaps can be specified so that the same address space contains all or parts of more than one bitmap.
- Visibility. Only virtual displays can be visible on the VAXstation screen. Program-controlled bitmaps are never visible, but must be copied to a virtual display.

Synchronizing Output

The VAXstation Display System is a Direct Memory Access (DMA) device. The primary attribute of a DMA device is that it accesses its data directly from the computer memory rather than through a buffered communication link. Since the device is accessing the same memory that the computer can access, this memory must remain intact and uncorrupted during the I/O operation.

Memory for bitmaps and data structures such as path lists must be allocated by the program and must not be modified or deallocated while an output request is outstanding.

The standard VAX/VMS mechanisms for synchronizing direct I/O are event flags and AST routines. The five native graphics procedures allow the caller to specify the normal VAX/VMS I/O synchronization parameters: event flag, AST service routine, and a parameter to be passed to the specified routine.

Calling the Procedures

The five native graphics procedures follow the same calling conventions as the standard VSTA library procedures. Refer to *Programming for the VAXstation Display System* for more information.

Error Returns

When the five native graphics procedures are called as functions, they return one of two status codes. If the procedure successfully queues output, it returns the status code SS$_NORMAL. If an error occurs, each procedure returns its error code (listed with the procedure description).

If an error occurs, an error block exists and can be retrieved with the procedure VSTA$GET_ERROR_BLOCK.

Since the procedures are asynchronous, the status codes returned as the value of the function indicate whether queueing the output request is successful. The procedures follow the VAX/VMS convention of using an I/O status block to return information about the success of the output operation itself.

Symbol Definitions

The symbols listed in this manual are defined in the same files as the symbols for the standard VSTA library procedures. The files SYS$LIBRARY:VSTAGBL.*** contain symbol definitions. The files SYS$LIBRARY:VSTAMSG.*** contain status code definitions. The files with the appropriate language-specific extensions are listed in *Programming for the VAXstation Display System*.

2

Copying an Area

Description

Copying one area to another is the fundamental operation of the display system. In the simplest form, copying an area merely moves a source image to a destination image. The source and destination are identically-sized rectangles; each source pixel replaces the destination pixel at the corresponding coordinates.

Copying can also use only part of the source or destination images. A subset of the source image can be selected using a source mask and source offset. The placement of the source image in the destination image can be changed with a destination offset. Clipping rectangles can restrict the parts of the destination image to be modified.

In addition to simple replacement, a map can be specified so that the values moved to the destination are a function of the source pixel values or a function of both source and destination values.

The exact behavior of the copy area operation is determined by:

- Source type and source image
- Source offset
- Source mask type and source mask
- Destination type and destination image
- Destination offset
- Map type
- Clipping rectangles

Source

The source is an image whose pixel values are used to update the destination. The source image can be one of five types:

- Constant. The source can be a single constant value that replaces all pixels in the destination. A constant is used, for example, to set the entire destination to a single value (color or shade).

- Bitmap. The source can be a program-controlled bitmap. If the destination is also a program-controlled bitmap and the two bitmaps overlap, the copy operation sequences through the pixels so that source pixels are not modified before they are used to update the destination.
- Halftone bitmap. The source can be a program-controlled bitmap containing a halftone that is used to tile the destination. The bitmap is replicated horizontally and vertically as needed to fill the destination.
- Virtual display. Copying an area when the source is a virtual display is the same copying as when the source is a bitmap.
- Halftone virtual display. Copying an area when the source is a halftone virtual display is the same as copying an area when the source is a halftone bitmap.

Source Offset

The source offset specifies the point, relative to the upper left corner of the source image, at which the upper left corner of the source mask is placed.

Source Mask

The source mask defines a rectangular subset of the source to be used to modify destination pixels. That is, the source mask restricts the set of source pixels used in the copy operation.

Three mask formats are available, depending on how the source subset is to be determined.

- Rectangle Mask. A rectangle mask selects a rectangular subset of the source image. Only the width and height of the rectangle are specified, since the source offset determines the location of the upper left corner of the rectangle on the source image.
- Virtual Display Mask. This is the most general form. The one-valued pixels in the mask specify a subset of the pixels in the source image. Only this subset is copied to the destination. The mask can be thought of as a template or a stencil, with the one-valued pixels making some pattern to be copied. When the mask is placed on the source image, the one-valued pixels in the mask select pixels in the same pattern from the source image. Once again, the source offset determines how the mask is to be placed relative to the source image.
- Bitmap Mask. A program-controlled bitmap mask is used in the same way as a virtual display mask.

Any of the three types of source mask can be selected with any type of source image. For example, a rectangle mask with a constant or halftone source generates a rectangle filled with that color or halftone for use as the source. Copying an area can thus perform a simple rectangle fill. (More complex fill operations can be performed with VSTA$FILL_AREA.)

A mask is not required. If no mask is specified, the result is the same as using a rectangle mask the same size as the destination. With a constant or halftone source, the entire destination is filled. With a virtual display or bitmap source, the entire source is copied to the destination.

The source image or the destination image can also be specified as the mask.

Destination

The two types of destination image are:

- Virtual display
- Bitmap

Although the destination can be a bitmap, program-controlled bitmaps are never visible on the VAXstation screen. The bitmap must be copied to a virtual display before it is visible.

Destination Offset

The destination offset determines the placement of source pixels in the destination image. The source offset and source mask define a set of source image pixels whose origin is relative to the upper left corner of the source image. The destination offset specifies where that origin should be placed in the destination image.

Figure 2–1 shows a source image, a rectangle source mask, and a destination image after the source image is copied to it. The location of the source offset is marked on the source image. The location of the destination offset is marked on the destination image.

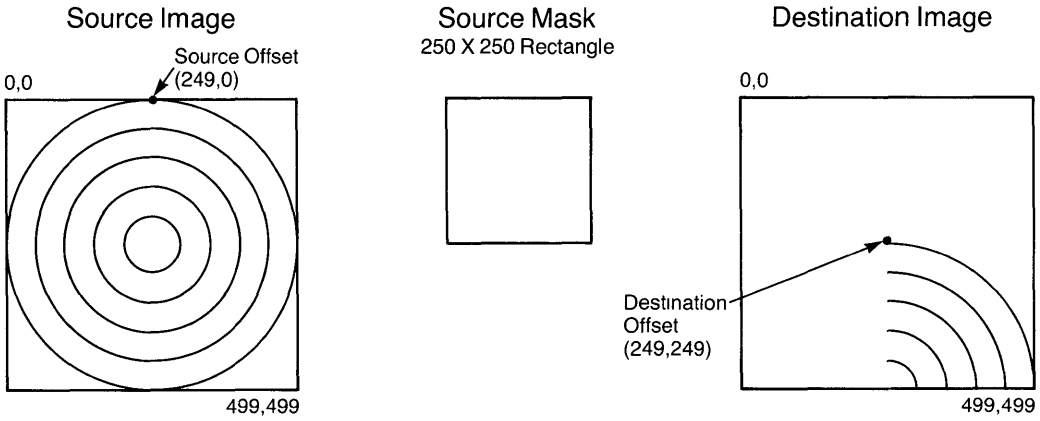


Figure 2–1. Source Offset and Destination Offset

Map

The map defines the values used to replace the selected destination pixels.

For a one-bitplane device, the map types specify the 16 logical functions of the source and destination pixel values. (For multiplane devices, other types of mapping are possible, such as mapping several source pixel values to one destination pixel value.)

Table 2–1 lists the symbols used to specify the map types for a one-bitplane device and describes the logical function performed by each type.

Table 2–1. Map Types

Symbol	Function
VSTA$K_MAP_SRC	No transformation. Source pixel values replace the values of the selected destination pixels.
VSTA$K_MAP_NOTSRC	NOT source. The NOT function is applied to the source pixel values. On a device with one bitplane, the result is reverse video of the source image before it is copied to the destination.
VSTA$K_MAP_DST	No transformation. This function also has no effect; destination pixels replace themselves.
VSTA$K_MAP_NOTDST	NOT destination. The NOT function is applied to the values of the selected destination pixels. On a device with one bitplane, the result is reverse video of the destination image.
VSTA$K_MAP_SRC_AND_DST	Source AND destination. The AND function is applied to the values of the source and destination pixels.
VSTA$K_MAP_NOTSRC_AND_DST	NOT source AND destination. The NOT function is applied to source pixel values; the results are ANDed with destination pixel values.
VSTA$K_MAP_SRC_AND_NOTDST	Source AND NOT destination. The NOT function is applied to destination pixel values; the results are ANDed with source pixel values.
VSTA$K_MAP_NOTSRC_AND_NOTDST	NOT source AND NOT destination. The NOT function is applied to the source pixels values and to the destination pixel values; the results are ANDed.
VSTA$K_MAP_SRC_OR_DST	Source OR destination. The values of the source and destination pixels are ORed.
VSTA$K_MAP_NOTSRC_OR_DST	NOT source OR destination. The NOT function is applied to the source pixel values; the results are ORed with the destination values.
VSTA$K_MAP_SRC_OR_NOTDST	Source OR NOT destination. The NOT function is applied to the destination pixel values; the results are ORed with the source values.
VSTA$K_MAP_NOTSRC_OR_NOTDST	NOT source OR NOT destination. The NOT function is applied to the source pixel values and to the destination pixel values; the results are ORed.
VSTA$K_MAP_SRC_XOR_DST	Source XOR destination. The XOR function is applied to the source and destination pixel values.
VSTA$K_MAP_NOT_SRCXORDST	NOT (source XOR destination). The values of the source and destination pixels are XORed; the NOT function is applied to the results.
VSTA$K_MAP_BLACK	Replace all destination pixel values with zero.
VSTA$K_MAP_WHITE	Replace all destination pixel values with one.

Some results can be achieved in more than one way. For example, all pixels in the destination image can be set to black by specifying a constant source and the no transformation map type (VSTA$K_MAP_SRC). Specifying the black map type (VSTA$K_MAP_BLACK) with any source type has the same result, but requires transformations. Generally the method that requires the fewest transformations is the most efficient.

Clipping Rectangles

Clipping rectangles restrict the copying operation to some subset of the pixels in the destination. Each clipping rectangle specifies an area of the destination that can be modified. Thus, the union of all clipping rectangles provides a boundary that limits the copying operation. The clipping rectangles should not overlap; if they do, the results are unpredictable.

The VSTA$COPY_AREA Procedure

The VSTA$COPY_AREA procedure implements the copy-area function.

Format

```
Status = VSTA$COPY_AREA (source-type, source-image, source-offset-x, source-offset-y,
         mask-type, mask, dest-type, dest-image, dest-offset-x, dest-offset-y, map-type,
         [reserved1], [reserved2], num-of-rectangles, clipping-rectangles, [wait-flag], [efn],
         [astadr], [astprm], [iosb])
```

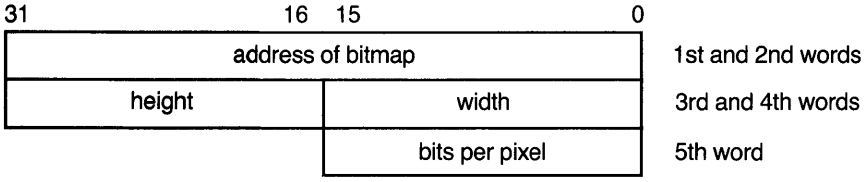
Parameters

source-type. One of five symbols that specify the five source image types. Passed by reference.

VSTA$K_SRC_CONST	Source image is a constant.
VSTA$K_SRC_BM	Source image is a bitmap.
VSTA$K_SRC_VD	Source image is a virtual display.
VSTA$K_SRC_HT_BITMAP	Source image is a bitmap containing a halftone pattern.
VSTA$K_SRC_HT_VD	Source image is a virtual display containing a halftone pattern.

source-image. An integer constant, a bitmap descriptor, or a channel number. Passed by reference. The source image must be the type specified by the source-type parameter.

A bitmap descriptor is a five-word block in the following format:



source-offset-x, source-offset-y. Integers used to identify the upper left corner of the subset of the source image used as the mask.

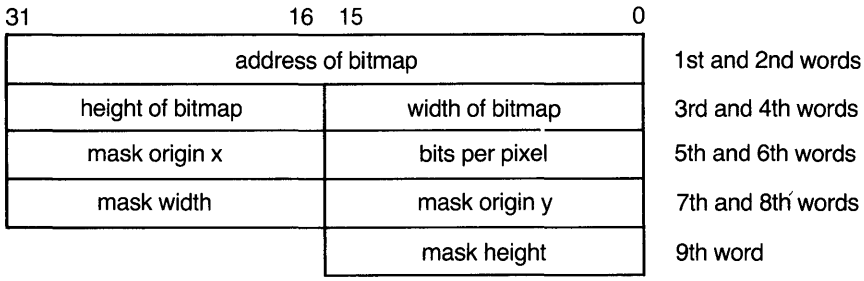
mask-type. One of four symbols that specify the four types of mask. Passed by reference.

VSTA$K_MSK_NONE	No mask. The mask parameter is ignored.
VSTA$K_MSK_RECTANGLE	Mask is a rectangle.
VSTA$K_MSK_BITMAP	Mask is a bitmap.
VSTA$K_MSK_VD	Mask is a virtual display.

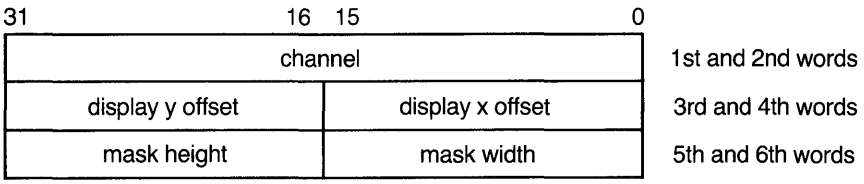
mask. The mask. Passed by reference.

If the mask is a rectangle, the mask parameter is a 2-word block. The first word contains the width of the rectangle; the second word contains the height. (The location of the upper left corner of the rectangle in the source image is specified by the source offset.)

If the mask is a bitmap, the mask parameter is a sub-bitmap descriptor with the following format:



If the mask is a virtual display, the mask parameter is a six-word block with the following format:



dest-type. One of two symbols that specify the two destination image types. Passed by reference.

VSTA$K_DST_BITMAP	Destination is a bitmap.
VSTA$K_DST_VD	Destination is a virtual display.

dest-image. The destination image. Passed by reference. If the destination is a bitmap, the dest-image parameter is a bitmap descriptor. If the destination is a virtual display, the parameter is a channel number.

dest-offset-x, dest-offset-y. Integers specifying horizontal and vertical offsets from the upper left corner of the destination image. The specified area of the source image is copied beginning at the location specified by the destination offset. Passed by reference.

map-type. One of 16 symbols that specify the 16 map function types. Passed by reference. The symbols are:

VSTA$K_MAP_SRC	No transformation.
VSTA$K_MAP_NOTSRC	NOT source.
VSTA$K_MAP_DST	No transformation; no effect.
VSTA$K_MAP_NOTDST	NOT destination.
VSTA$K_MAP_SRC_AND_DST	Source AND destination.
VSTA$K_MAP_NOTSRC_AND_DST	NOT source AND destination.
VSTA$K_MAP_SRC_AND_NOTDST	Source AND NOT destination.
VSTA$K_MAP_NOTSRC_AND_NOTDST	NOT source AND NOT destination.
VSTA$K_MAP_SRC_OR_DST	Source OR destination.
VSTA$K_MAP_NOTSRC_OR_DST	NOT source OR destination.

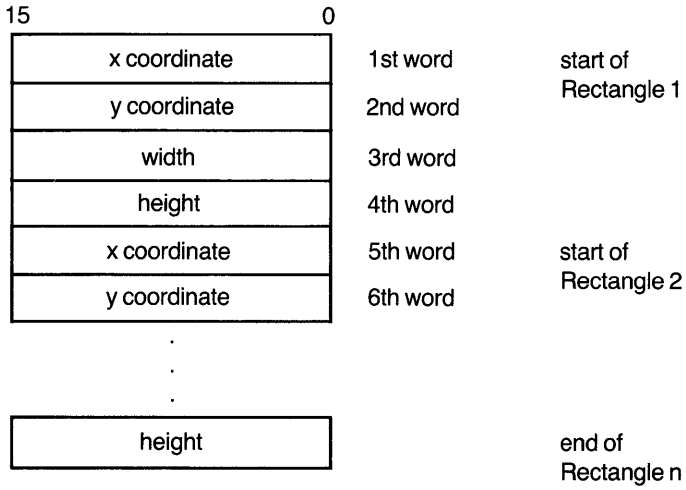
VSTA$K_MAP_SRC_OR_NOTDST	Source OR NOT destination.
VSTA$K_MAP_NOTSRC_OR_NOTDST	NOT source OR NOT destination.
VSTA$K_MAP_SRC_XOR_DST	Source XOR destination.
VSTA$K_MAP_NOT_SRCXORDST	NOT (source XOR destination).
VSTA$K_MAP_BLACK	Replace all destination pixel values with zero.
VSTA$K_MAP_WHITE	Replace all destination pixel values with one.

[reserved1, reserved2]. Reserved for future use.

num-of-rectangles. An integer specifying the number of rectangles in the clipping rectangle list. Passed by reference.

clipping-rectangles. A list of rectangles in the destination image. Passed by reference.

The format of the rectangle list is:



The rectangles must not intersect.

[wait-flag]. An integer flag specifying whether the procedure returns control to the calling program after queuing the output request, or waits until I/O completion. Passed by reference.

This procedure uses either VMS event flag zero or the specified event flag as the wait flag.

[efn]. An event flag number. The specified event flag is cleared when the I/O request is queued, and set when the I/O is completed. Passed by reference.

The default event flag is event flag zero. Event flag zero is also the default event flag for all VMS system services that use event flags and for the standard VSTA procedures that read input from the mouse, the keyboard, and the tablet. Specifying another event flag is recommended (even if the event flag is not tested) to avoid possible conflicts.

[astadr]. The address of the AST service routine to be executed when I/O is completed. See the appropriate language reference manual for the method of passing routine addresses. NOTE: Zero must be passed by value to specify that there is no AST routine.

[astprm]. An integer value to be passed as a parameter to the AST routine. Passed by reference. NOTE: The parameter is passed to the AST routine by value.

[iosb]. An I/O status block. Passed by reference.

Return Status

VSTA$_CPYARE
Message: Copy area failed.

Example Code

The following FORTRAN subroutine calls VSTA$COPY_AREA to copy one bitmap to another bitmap. The subroutine is called by the example program listed in Appendix B. The example program also illustrates other uses of the VSTA$COPY_AREA procedure.

Example 2–1 Using VSTA$COPY_AREA

```
!-----------------------------------------------------------------
      SUBROUTINE copy_bitmap (source_descr, map_function)
!-----------------------------------------------------------------
!
! Function:
!
!     To copy either the bug bitmap or the bar bitmap to the
!     scratch bitmap using the mapping function map_function.
!     The bitmaps are the same size, there is no mask, and the
!     destination offset is 0,0.
!
! Input args:
!     source_descr = address of the source bitmap descriptor.
!     map_function = map function
!
! Output args: none
!
      IMPLICIT NONE
!
! Include VAXstation defined symbols.
!
      INCLUDE 'SYS$LIBRARY:VSTAGBL.FOR'

!
! Argument declarations:
!
      INTEGER*4     source_descr, map_function
!
! Routine declarations:
      INTEGER*4     status, VSTA$COPY_AREA
      INTEGER*4     scratch(2,49), scratch_address
      INTEGER*2     scratch_descr(5)
      EQUIVALENCE   (scratch_address, scratch_descr(1))
      COMMON        scratch_descr, scratch
```

```
!
! Set up the descriptor for the scratch bitmap.
!
      scratch_address = %LOC(scratch)
      scratch_descr(3) = 64
      scratch_descr(4) = 49
      scratch_descr(5) = 1

            status = VSTA$COPY_AREA(
     1              VSTA$K_SRC_BM, source_descr, 0, 0,
     2              VSTA$K_MSK_NONE, ,
     3              VSTA$K_DST_BITMAP, scratch_descr, 0, 0,
     4              map_function, , ,
     5              0, , 1, 5, , , )
            IF ( .NOT. status ) CALL show_error(status)

      RETURN
      END
```

3

Drawing a Curve

Description

Drawing a curve paints a source image through a sequence of points, drawing either a straight or a curved line from point to point.

Drawing a curve is similar to copying an area in some respects. Both operations specify the source with a source image, source offset, and source mask; and specify the destination with a destination image and destination offset. Drawing a curve can also use a map to modify the way pixels are replaced, and clipping rectangles to limit the area of the destination in which the curve is drawn in the same way.

In addition, drawing a curve requires a path, that is, a list of points in the destination that define the line to be drawn. Drawing a curve can use a pattern for drawing the line.

Drawing a curve proceeds as follows. First, the source is determined by the source image, source mask, and source offset. Second, all points on the line to be drawn are determined from the path. For a straight line, the path must contain at least two points. For a curved line, the path must contain at least three points. Curves are drawn using a cubic spline algorithm so that the curved line has no abrupt bends. Then the source image is copied with its origin at each point on the line.

The first point in any path is the destination offset. Each supplied point defines the end of a segment of the path. Thus the first segment of the path goes from the offset to the first point in the list of points; the second segment goes from the first point to the second point, and so on.

If the source image is specified as a halftone, the halftone pattern is always aligned with the destination to give the effect of painting with a halftone.

A pattern string provides for drawing of dashed or patterned lines and curves. A patterned line alternates between writing and not writing pieces of a segment based on strings of ones and zeros in the pattern string. Patterned lines also

allow the specification of a secondary source. A patterned line alternates between the source and the secondary source, based on ones and zeros in the pattern string. A pattern string and a secondary source can be used, for example, to draw a line that alternates between two colors.

The exact behavior of drawing a curve is determined by:

- Source type and source image
- Source offset
- Mask type and mask
- Destination type and destination image
- Destination offset
- Map type
- Path
- Pattern mode and pattern
- Secondary source type and source image
- Secondary source offset
- Clipping rectangles

The source type, image, and offset; the destination type, image, and offset; the map type, and the clipping rectangles are the same as for copying an area.

The source mask is specified and used in the same way as for copying an area. However, the default is different when no mask is specified. For drawing a curve, specifying no mask is the same as specifying a one-by-one rectangle, which draws a line one pixel wide.

Path

A path is specified as a list of segments. Each path segment is described by the x and y coordinates of its end point and a flag word. The starting point is the end of the previous segment. (For the first segment, the starting point is the destination offset.) The flag word describes the characteristics of that segment, and contains the following flags:

- Origin or relative flag. Indicates how the end point coordinates are to be interpreted. In origin mode, the coordinates are relative to the destination origin. In relative mode, the coordinates are relative to the end of the previous segment in the path.
- Draw or move flag. Indicates whether the segment should actually be drawn. If drawing is specified, the straight line or curve is actually drawn. If moving is specified, the line is not drawn; the position is simply advanced to the next point.

  Since the starting point for the first segment of a path is the destination offset, the move flag is often set for the first point in the path so that an invisible line is drawn from the destination offset to the first point. Such invisible lines can also be used to provide information to the cubic spline algorithm, or to draw several disconnected lines in a single draw curve operation. When move is specified, the pattern string does not advance.

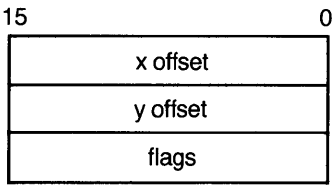
- Straight or curved flag. Indicates whether the segment is curved or straight. If the flag is not set, a straight line is drawn from the previous point to this point. If the flag is set, a curve is drawn using a cubic spline algorithm. That is, the curve merges smoothly with the preceding and following segments without abrupt bends. For a curve to be drawn through a point, then, the point must have a predecessor and a successor to define the tangents to the point.
- Start closed figure flag. Indicates that this point is the first in a series of segments that define a closed figure. (A path is also specified for filling an area, which requires a closed figure. Although the start and end closed figure flags do not affect the way the curve is drawn, they are required for filling an area.)
- End closed figure flag. Indicates that this is the last point in the closed figure. The first and last points (that is, the start and end points) for a closed figure must be identical. For example, a circle can be specified by the following five-point path:

  point 1: move to P1, start closed figure
  point 2: draw curve to P2
  point 3: draw curve to P3
  point 4: draw curve to P4
  point 5: draw curve to P1, end closed figure

- Draw last image flag. In paths with several segments, the last point in one segment is also the first point in the next. Therefore, either the pixel at this point or some rectangular set of pixels whose origin is at the last point are specified twice. The draw last image flag allows this last image to be drawn only once.

  When source pixel values directly replace destination pixel values, the result is the same whether the last image is drawn once or twice. However, for some of the mapping functions where the current state of the destination affects the pixel values, drawing the last image twice can change the result.

The path is thus a list of points, where the format of each point is:



Various combinations of source and mask can be used to draw lines of different widths. Assume that the path contains the following points, move flags, and origin flags:

endpoint 1, x = 10, y = 10, origin, move, straight
endpoint 2, x = 10, y = 80, origin, draw, straight
endpoint 3, x = 50, y = 80, origin, draw, straight
endpoint 4, x = 50, y = 10, origin, draw, straight
endpoint 5, x = 10, y = 10, origin, draw, straight

Figure 3–1 shows the rectangle drawn along this path when the source is a constant and there is no mask. For drawing a curve, specifying no mask is equivalent to specifying a 1 x 1 rectangle mask. The line drawn is one pixel wide.

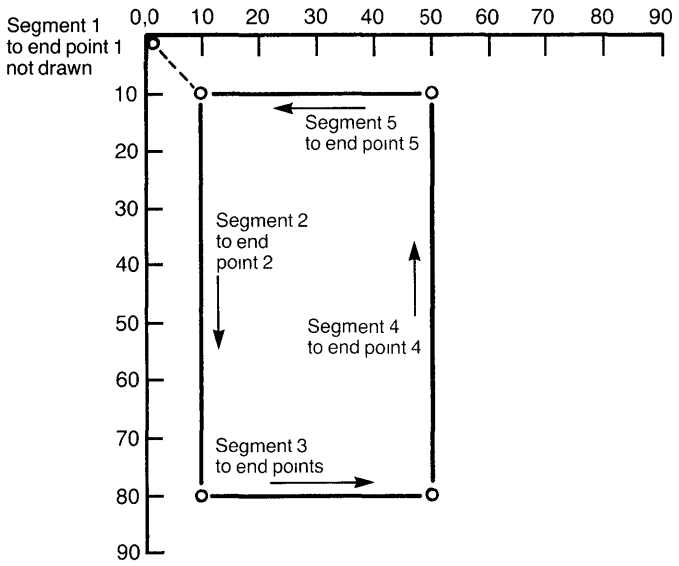


Figure 3–1. Drawing a Curve

Figure 3–2 shows the same path when the source is a constant and the mask is a 5 x 10 rectangle.

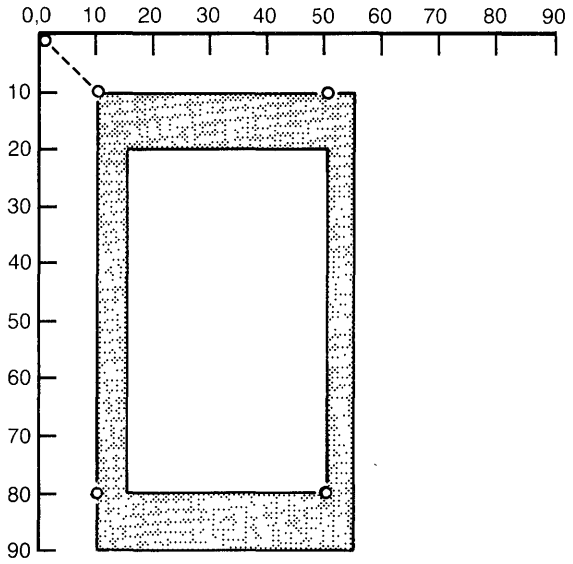


Figure 3–2. Drawing a Curve with Wide Lines

The dots show each of the segment end points as in the preceding illustration. The 5 x 10 rectangle mask is drawn with its upper left corner at 10, 10, then replicated vertically until a 5 x 10 mask is drawn with its upper left corner at 80, 10.

Because the width of the rectangle is less than its height, the vertical lines are narrower than the horizontal lines.

Generally, drawing curves is faster when paths with several segments are specified in one procedure call than when each segment is specified in a separate procedure call. Overhead for procedure calls, for data transfers, and for setting up the display processor is minimized with multi-segment paths.

Patterned Lines

Patterns simplify the drawing of linearly patterned lines. A pattern can be used in either single-source or alternate-source mode, and consists of the pattern string and pattern multiplier. The current state of the pattern after the curve is drawn can be returned to the calling program so that a pattern can be continued across several curves.

Pattern Mode — The pattern string operates in two modes. Both modes select the pattern in the same way, that is, with the pattern string. The pattern string specifies which pixels in the destination are to be modified. The modes specify how the pixels are modified.

The two modes are:

- Single-source mode. In this mode, the occurrence of a one bit in the pattern causes the source to be copied to the current destination pixel along the curve path, while a zero bit causes the destination pixel to be skipped. Single-source mode allows the existing background to "show through" the spaces in the pattern. Single-source mode can be used, for example, to write a dashed line.
- Alternate-source mode. In this mode, the occurrence of a one bit in the pattern causes the source to be copied to the current destination pixel along the curve path, as above. However, a zero bit causes a different source to be copied to the destination. Specifying one constant as the source and another constant as the secondary source writes a patterned line that alternates between two colors.

Thus, single-source mode alternates between writing and not writing a single source. Alternate-source mode alternates between writing two different sources.

Pattern String — The pattern string is a list of 0 to 16 bits. The ones and zeros in the pattern string modify the writing of the source as it is painted from pixel to pixel in the destination. When the pattern string is exhausted, it is repeated again from the start. The pattern is only applied to drawn segments. That is, invisible (move mode) lines do not advance the pattern. A pattern string of length 0 is identical to a pattern string with length 1 and value 1.

Pattern Multiplier — The pattern multiplier specifies the number of times each bit in the pattern string is repeated before moving on to the next bit. For example, if the pattern string is 10 and the pattern multiplier is 3, the pattern 111000 is used to generate the patterned lines.

Pattern State — The pattern state allows drawing a curve to continue at the pattern string position at which a previous curve was completed. In this way, pattern strings can be used across several draw curve procedures.

The pattern state consists of two 16-bit words, the pattern position and the pattern count. The pattern position specifies the zero-origin offset of the starting bit within the pattern string, and must be less than or equal to the size of the string minus one. Pattern count specifies the starting count to be used for that bit on its first use, and must be less than or equal to the pattern multiplier.

If a pattern state is specified, scanning the pattern string begins at the specified pattern position. The specified starting bit is then repeated pattern-multiplier minus pattern-count times. The scan then continues at the following bit, with each bit used pattern-multiplier times.

The pattern state can be updated after the curve is drawn to indicate where in the pattern string the next curve should begin.

Secondary Source

In alternate-source mode, the secondary source specifies the source that is copied to the destination whenever a zero bit in the pattern string is encountered. The secondary source can be specified as any of the types allowed for the single source.

Secondary Source Offset

The secondary source offset specifies how the source mask is applied to the secondary source, when the secondary source is selected in alternate-source mode.

The VSTA$DRAW_CURVE Procedure

The VSTA$DRAW_CURVE procedure implements the draw-curve function.

Format

```
Status = VSTA$DRAW_ CURVE (source-type, source-image, source-offset-x, source-offset-y,
         mask-type, mask, dest-type, dest-image, dest-offset-x, dest-offset-y, map-type,
         [reserved1], [reserved2], num-of-points, path, pattern-mode, pattern-action,
         pattern-block, pattern-state, sec-source-type, sec-source, sec-source-offset-x,
         sec-source-offset-y, num-of-rectangles, clipping-rectangles, [wait-flag], [efn],
         [astadr], [astprm], [iosb])
```

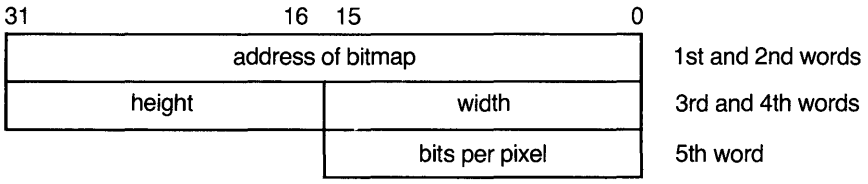
Parameters

source-type. One of five defined symbols that specify the five source image types. Passed by reference.

VSTA$K_SRC_CONST	Source image is a constant.
VSTA$K_SRC_BM	Source image is a bitmap.
VSTA$K_SRC_VD	Source image is a virtual display.
VSTA$K_SRC_HT_BITMAP	Source image is a bitmap containing a halftone pattern.
VSTA$K_SRC_HT_VD	Source image is a virtual display containing a halftone pattern.

source-image. An integer constant, a bitmap descriptor, or a channel number. Passed by reference. The source image must be the type specified by the source-type parameter.

A bitmap descriptor is a five-word block in the following format:



source-offset-x, source-offset-y. Integers specifying the upper left corner of the mask on the source image.

mask-type. One of four defined symbols that specify the four types of mask. Passed by reference.

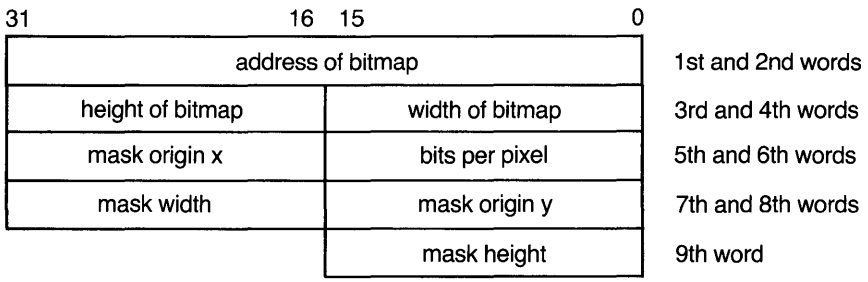
VSTA$K_MSK_NONE	There is no mask. In this case, the mask parameter is ignored. Specifying no mask draws a line one pixel wide.
VSTA$K_MSK_RECTANGLE	Mask is a rectangle.
VSTA$K_MSK_BITMAP	Mask is a bitmap.
VSTA$K_MSK_VD	Mask is a virtual display.

The default type is VSTA$K_MSK_NONE. For drawing a curve, the default mask is a 1 x 1 rectangle, which draws a line one pixel wide.

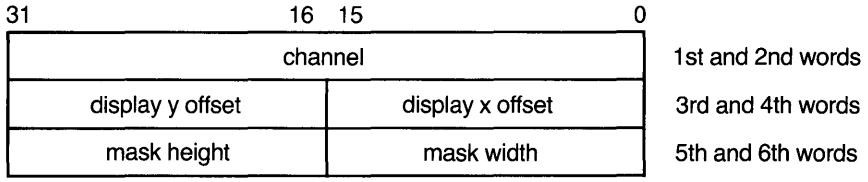
mask. The mask. Passed by reference.

If the mask is a rectangle, the mask parameter is a 2-word block. The first word contains the width of the rectangle; the second word contains the height. (The location of the upper left corner of the rectangle in the source image is specified by the source offset.)

If the mask is a bitmap, the mask parameter is a sub-bitmap descriptor with the following format:



If the mask is a virtual display, the mask parameter is a block with the following format:



dest-type. One of two symbols that specify the two destination image type Passed by reference.

VSTA$K_DST_BITMAP	Destination is a bitmap.
VSTA$K_DST_VD	Destination is a virtual display.

dest-image. The destination image. Passed by reference. If the destination is bitmap, the dest-image parameter is a bitmap descriptor. If the destination is virtual display, the parameter is a channel number.

dest-offset-x, dest-offset-y. Integers specifying horizontal and vertical offse from the upper left corner of the destination image. The source image is copi beginning at the location specified by the destination offset. Passed by referenc

map-type. One of 16 symbols that specify the 16 map function types. Passed l reference.

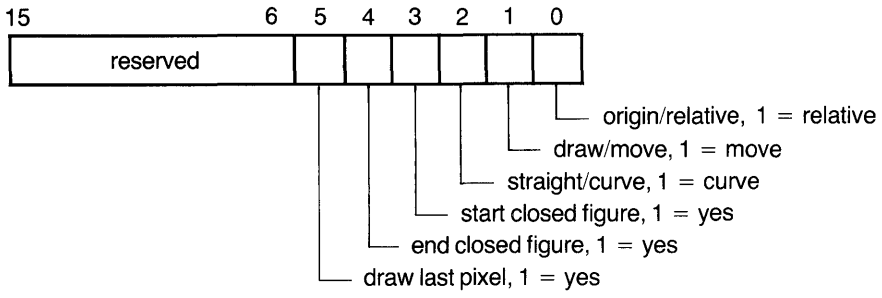
VSTA$K_MAP_SRC	No transformation.
VSTA$K_MAP_NOTSRC	NOT source.
VSTA$K_MAP_DST	No transformation; no effect.
VSTA$K_MAP_NOTDST	NOT destination.
VSTA$K_MAP_SRC_AND_DST	Source AND destination.
VSTA$K_MAP_NOTSRC_AND_DST	NOT source AND destination.
VSTA$K_MAP_SRC_AND_NOTDST	Source AND NOT destination.
VSTA$K_MAP_NOTSRC_AND_NOTDST	NOT source AND NOT destination.
VSTA$K_MAP_SRC_OR_DST	Source OR destination.
VSTA$K_MAP_NOTSRC_OR_DST	NOT source OR destination.
VSTA$K_MAP_SRC_OR_NOTDST	Source OR NOT destination.
VSTA$K_MAP_NOTSRC_OR_NOTDST	NOT source OR NOT destination.
VSTA$K_MAP_SRC_XOR_DST	Source XOR destination.
VSTA$K_MAP_NOT_SRCXORDST	NOT (source XOR destination).
VSTA$K_MAP_BLACK	Replace all destination pixel values with zero.
VSTA$K_MAP_WHITE	Replace all destination pixel values with one.

[reserved1, reserved2]. Reserved for future use.

num-of-points. An integer specifying the number of points in the path. Pass by reference.

path. A list of points, each of which consists of three 16-bit words containing integer x value, an integer y value, and a flag word. Passed by reference.

The format of the flag word is:



pattern-mode. One of two symbols that specify either single-source mode or alternate-source mode. Passed by reference.

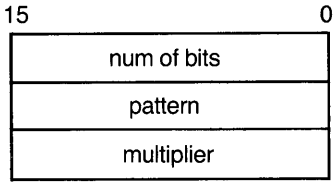
VSTA$K_PTN_SINGLE_SRC	Mode is single source.
VSTA$K_PTN_ALTERN_SRC	Mode is alternate source.

pattern-action. One of two symbols specifying whether the pattern state should be updated on return from the procedure. Passed by reference.

VSTA$K_PTN_UPDATE	Update the pattern state parameter.
VSTA$K_PTN_NO_UPDATE	Do not update the pattern state parameter.

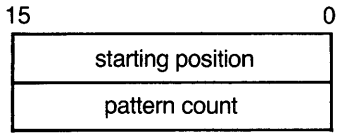
pattern-block. A three-word block containing the number of bits in the pattern, the pattern itself, and the multiplier. Passed by reference.

The format is:



pattern-state. A two-word block containing a pattern position and multiplier count. Passed by reference. These two words are updated after the curve is drawn if updating is specified with the pattern-action parameter.

The format is:



sec-source-type. One of five symbols that specify the five source types. Passed by reference. The symbols are listed with the source-type parameter. This parameter is ignored if the mode is single source.

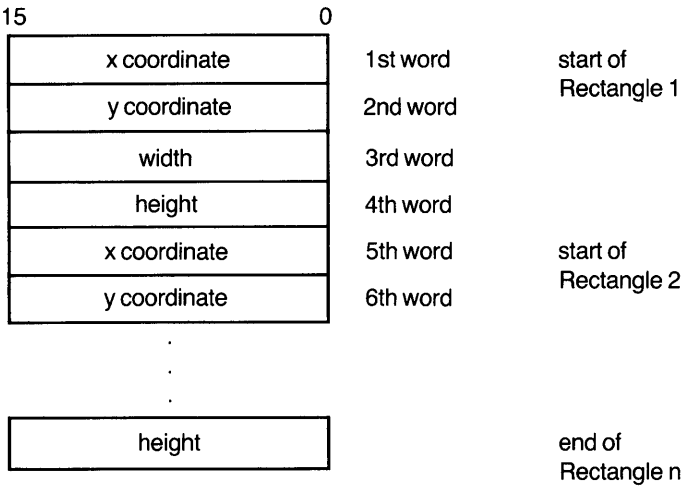
sec-source. A secondary source used in alternate-source mode. Passed by reference. See the source parameter for the possible formats. This parameter is ignored if the mode is single source.

sec-source-offset-x, sec-source-offset-y. Integers specifying the location of the upper left corner of the source mask in the secondary source. Passed by reference. These parameters are ignored if the mode is single source.

num-of-rectangles. An integer specifying the number of rectangles in the clipping rectangle list. Passed by reference.

clipping-rectangles. A list of rectangles in the destination image. Passed by reference.

The format of the rectangle list is:



The rectangles must not intersect.

[wait-flag]. An integer flag specifying whether the procedure returns control the calling program after queuing the output request, or waits until I/O compl tion. Passed by reference.

This procedure uses either VMS event flag zero or the specified event flag as t wait flag.

[efn]. An event flag number. The specified event flag is cleared when the I request is queued, and set when the I/O is completed. Passed by reference.

The default event flag is event flag zero. Event flag zero is also the default eve flag for all VMS system services that use event flags and for the standard VST procedures that read input from the mouse, the keyboard, and the tablet. Spe fying another event flag is recommended (even if the event flag is not tested) avoid possible conflicts.

[astadr]. The address of the AST service routine to be executed when I/O is co pleted. See the appropriate language reference manual for the method of passi routine addresses. NOTE: Zero must be passed by value to specify that there no AST routine.

[astprm]. An integer value to be passed as a parameter to the AST routir Passed by reference. NOTE: The parameter is passed to the AST routine value.

[iosb]. An I/O status block. Passed by reference.

Return Status

VSTA$_DRACUR
Message: Draw curve failed.

Example Code

The following FORTRAN subroutine calls VSTA$DRAW_CURVE to draw a circle. The example program listed in Appendix B calls this subroutine. The example program also shows other uses of the procedure VSTA$DRAW_CURVE.

Example 3–1: Using VSTA$DRAW_CURVE

```
!-----------------------------------------------------------------------
      SUBROUTINE draw_circle ( vd_id, width, color, radius, x, y,
     1                                 offx, offy )
!-----------------------------------------------------------------------
!
! Function:
!
!     Draw a circle on the display whose channel number is vd_id,
!         the circle having a line width of width, a line color of
!         color, a radius of  radius, and centered at (x,y) .
!
      IMPLICIT NONE
!
! Include the VAXstation defined symbols.
!
      INCLUDE 'SYS$LIBRARY:VSTAGBL.FOR'
!
! Input declarations.
!
      INTEGER*4     vd_id, width, color, radius, x, y, offx, offy
!
! Routine declarations.
!
      PARAMETER     f_rel = 1, f_move = 2, f_curve = 4,
     1              f_start_closed = 8, f_end_closed = 16,
     2              f_draw_last = 32
      INTEGER*2     pathlist(21), mask(2), pattern_block(3),
     1              pattern_state(2)
      INTEGER*4     status, VSTA$DRAW_CURVE

      DATA   pattern_block / 3 * 0 /,
     1       pattern_state / 2 * 0 /
!
! Fill a pathlist for a circle.
!
      pathlist(1) = x - radius
      pathlist(2) = y
      pathlist(3) = f_move + f_start_closed
      pathlist(4) = x
      pathlist(5) = y - radius
      pathlist(6) = f_move + f_curve
      pathlist(7) = x + radius
      pathlist(8) = y
      pathlist(9) = f_curve
```

```
        pathlist(10) = x
        pathlist(11) = y + radius
        pathlist(12) = f_curve
        pathlist(13) = x - radius
        pathlist(14) = y
        pathlist(15) = f_curve
        pathlist(16) = x
        pathlist(17) = y - radius
        pathlist(18) = f_curve
        pathlist(19) = x + radius
        pathlist(20) = y
        pathlist(21) = f_curve + f_move + f_end_closed
!
! Make the mask a square of dimension width to draw a wide line.
!
        mask(1) = width
        mask(2) = width
!
! Draw the circle.
!
        status = VSTA$DRAW_CURVE(
        1               VSTA$K_SRC_CONST, color, 0, 0,
        2               VSTA$K_MSK_RECTANGLE, mask,
        3               VSTA$K_DST_VD, vd_id, offx, offy,
        4               VSTA$K_MAP_SRC, , ,
        5               7, pathlist,
        6               VSTA$K_PTN_SINGLE_SRC, VSTA$K_PTN_NO_UPDATE,
        7                       pattern_block, pattern_state,
        8                , , , , 0, , 1, 5)
        IF ( .NOT. status ) CALL show_error(status)
!
! End of routine.
!
        RETURN
        END
```

4

Printing Text

Description

Printing text outputs a character string. The operation requires a font and a text string containing character codes. A font is a data structure containing both bitmap images of the characters and the information required to locate each image and to determine its width.

The print text operation scans the text string and copies each selected character to the destination. The rectangles containing the bitmap images of the characters are written horizontally.

To support simple string justification, the print text operation can add a fixed number of pixels after each character or after each space character. An optional control string provides for special formatting of output strings, including horizontal and vertical adjustments (for example, for subscripts).

Fonts can be used in two ways to specify the character images:

- As a source image
- As a mask

When the font is used as a source image, each rectangular character cell is copied to the destination. The cell contains both the character image and a background. As with the copy area operation, a map can be used to transform the pixel values for the image and background. Multiplaned systems can use a source image font to contain antialiased characters, that is, characters with gray scale.

When the font is used as a mask (called a mask font), the cells of the mask font define only the shapes of the characters. A mask font is always one bit per pixel, independent of the number of bits per pixel of the implementation. When a mask font is used, the source image can be either a constant or a halftone that specifies the writing color. With a mask font, only the character shapes themselves are written. There is no background rectangle for the characters. Mask fonts thus write characters without obliterating the underlying rectangular area.

Mask fonts are useful for writing overstruck characters and for writing characters over lines. They also provide a storage-efficient mechanism for writing halftone or single-color characters.

Both source fonts and mask fonts can be one of two types:

- A defined font
- A program-supplied font data structure

Defined fonts are the same as the fonts used with the standard VSTA library procedures and must be specified by the procedures VSTA$DEFINE_FONT or VSTA$DEFINE_SYSTEM_FONT. Program-supplied fonts are data structures containing information about the font and the bitmaps for each character.

The exact behavior of printing text is determined by:

- Source type and source image
- Mask font
- Destination type and destination image
- Initial destination offset
- Map type
- Clipping rectangles
- Text string
- Control commands
- Character pad
- Space pad

The map type and the clipping rectangles are the same as for copying an area. The others are discussed below.

Source

The source image for print text can be one of five types:

- A program-supplied font. Each element in the text string copies the specified rectangular font cell, including background, to the destination. No mask font can be specified with this option.
- A defined font. The operation is the same as for a program-supplied font.
- A constant. The constant value is used as the writing color for the characters whose symbols are defined in a mask font.
- A halftone bitmap. The halftone is used as the writing color for the characters whose symbols are defined in a mask font.
- A halftone virtual display. A halftone virtual display is used in the same way as a halftone bitmap.

4

Printing Text

Description

Printing text outputs a character string. The operation requires a font and a text string containing character codes. A font is a data structure containing both bitmap images of the characters and the information required to locate each image and to determine its width.

The print text operation scans the text string and copies each selected character to the destination. The rectangles containing the bitmap images of the characters are written horizontally.

To support simple string justification, the print text operation can add a fixed number of pixels after each character or after each space character. An optional control string provides for special formatting of output strings, including horizontal and vertical adjustments (for example, for subscripts).

Fonts can be used in two ways to specify the character images:

- As a source image
- As a mask

When the font is used as a source image, each rectangular character cell is copied to the destination. The cell contains both the character image and a background. As with the copy area operation, a map can be used to transform the pixel values for the image and background. Multiplaned systems can use a source image font to contain antialiased characters, that is, characters with gray scale.

When the font is used as a mask (called a mask font), the cells of the mask font define only the shapes of the characters. A mask font is always one bit per pixel, independent of the number of bits per pixel of the implementation. When a mask font is used, the source image can be either a constant or a halftone that specifies the writing color. With a mask font, only the character shapes themselves are written. There is no background rectangle for the characters. Mask fonts thus write characters without obliterating the underlying rectangular area.

Mask fonts are useful for writing overstruck characters and for writing characters over lines. They also provide a storage-efficient mechanism for writing halftone or single-color characters.

Both source fonts and mask fonts can be one of two types:

- A defined font
- A program-supplied font data structure

Defined fonts are the same as the fonts used with the standard VSTA library procedures and must be specified by the procedures VSTA$DEFINE_FONT or VSTA$DEFINE_SYSTEM_FONT. Program-supplied fonts are data structures containing information about the font and the bitmaps for each character.

The exact behavior of printing text is determined by:

- Source type and source image
- Mask font
- Destination type and destination image
- Initial destination offset
- Map type
- Clipping rectangles
- Text string
- Control commands
- Character pad
- Space pad

The map type and the clipping rectangles are the same as for copying an area. The others are discussed below.

Source

The source image for print text can be one of five types:

- A program-supplied font. Each element in the text string copies the specified rectangular font cell, including background, to the destination. No mask font can be specified with this option.
- A defined font. The operation is the same as for a program-supplied font.
- A constant. The constant value is used as the writing color for the characters whose symbols are defined in a mask font.
- A halftone bitmap. The halftone is used as the writing color for the characters whose symbols are defined in a mask font.
- A halftone virtual display. A halftone virtual display is used in the same way as a halftone bitmap.

Program-Supplied Fonts

A font data structure consists of a header describing the font and a bitmap containing the character images.

The actual character images are stored in "strike" format. That is, all the character images are concatenated to form a horizontal bitmap strip. The first character (the one with the lowest index) is on the left, and the last is on the right. The characters are aligned vertically so they have a common baseline. This combined image is stored in a bitmap. The height of the bitmap is at least equal to the range between the bottom of the lowest descender and the top of the tallest character. There is no restriction on the height of a font or the width of the characters in it.

The font is specified as the address of a font data structure. The data structure must be contiguous in memory. It consists of a header containing information about the font, followed by the bitmap containing the characters.

A font header is a table with the following format:

BITMAP<79:0>	specification of the bitmap containing the character images
FIRSTCHAR<15:0>	the first valid character index in the font
LASTCHAR<15:0>	the last valid character index in the font
LEFTARRAY<31:0>	a pointer to an array of 16-bit elements (indexed from FIRSTCHAR to LASTCHAR + 1) each giving the first x coordinate of the associated character image in the font bitmap (required if WIDTH is zero or if any index does not have an associated character image)
BASELINE<15:0>	the height of the character baseline from the top of the character cells (since origin is upper left)
SPACE<15:0>	the index of the space character in the font (32 decimal for ASCII fonts)
WIDTH<15:0>	the width of all characters in a fixed-width font; otherwise, zero for a variable-width font

All pointers in this table, including the LEFTARRAY pointer and the 32-bit pointer in the BITMAP specification, are relative addresses that must be added to the font address (that is, the address of the head of the font data structure). A font is thus a contiguous data structure; the header table contains relative addresses from the start of that data structure to the various components.

The font is accessed by indexing each character code into LEFTARRAY to retrieve the origin of the character bitmap. The element selected is LEFTARRAY[CHAR-FIRSTCHAR]. The width of the character is the difference between the origin of the character and the origin of its numerical successor (that is, LEFTARRAY[CHAR-FIRSTCHAR + 1] – LEFTARRAY[CHAR-FIRSTCHAR]). The height of the character is the height of the font bitmap.

Although an entry in LEFTARRAY is required for every index (every character code) between FIRSTCHAR and LASTCHAR + 1, a corresponding image in the character bitmap is not required. A missing image is equivalent to an image of zero width. The x coordinates in LEFTARRAY for the missing character and for its successor are the same.

Any character not within the range (FIRSTCHAR,LASTCHAR) inclusive is an error; the display terminates processing and reports the error to the host.

Defined Fonts

Fonts are defined by two standard VSTA library procedures. VSTA$DEFINE_SYSTEM_FONT defines a font that can be used with any virtual display. VSTA$DEFINE_FONT defines a font for one virtual display. Fonts that are defined by either of these procedures can be used for the print text operation.

Mask Font

A mask font can be one of two types: a program-supplied font data structure or a defined font. In either case, the mask font defines the symbol shapes for the characters in the font. The mask font is used with a constant or halftone source that defines writing color for the characters in the text string.

Destination

The destination can be either a bitmap or a virtual display.

Initial Destination Offset

The initial destination offset is a point in the destination image where character writing begins. The upper left corner of the character cell specified by the first character in the text string is placed at the initial destination offset.

The print text operation optionally updates the offset at the end of the operation. When updating is specified, the updated offset indicates the location of the pixel following the last output character, which is the location where the next character begins when output is continued on the same line.

For example, keyboard input characters can be echoed by specifying updating, a single-character buffer, and the initial offset for a line of text. Each character received from the keyboard can be stored in the buffer, then printed. When the character is printed, the offset is updated and thus contains the correct x and y values for the next character.

Text String

The text string can be a list of 8-bit character codes or 16-bit character codes.

Control Commands

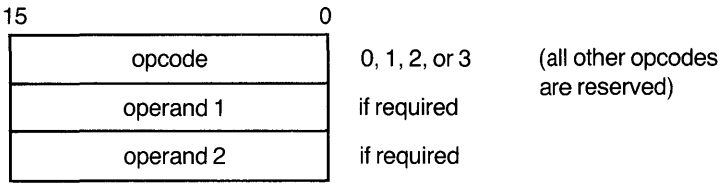
By default, characters are written on a single horizontal line with the spacing determined by the width of each character as stored in the font. This behavior can be modified with control commands to adjust the horizontal or vertical position and to skip some of the characters.

Each control command consists of a 16-bit opcode followed by zero to two 16-bit operands. The control string is a list of 16-bit words containing control commands. The commands are:

- OUT(n) (opcode = 0) outputs the next n characters of the text string, where n is the single operand.
- OUTALL (opcode = 1) outputs all remaining characters of the text string.

- SKIP(n) (opcode = 2) skips the next n characters of the text string, where n is the single operand.
- ADJUST(X,Y) (opcode = 3) adjusts the current character position by x and y, where x and y are signed horizontal and vertical adjustments specified in pixels.

The format of a control command is:



If the text string is exhausted before the control string, processing the control string continues with the following interpretation:

- Any ADJUST commands are obeyed.
- Any OUTALL commands are ignored.
- Any SKIP(n) or OUT(n) commands with n greater than zero cause termination of the operation with error status.

If the control string is exhausted before the text string, the operation terminates successfully at that point.

Character Pad

The character pad specifies the number of pixels to be added to the destination offset following each character (including the last character in the text string).

Space Pad

The space pad specifies the number of pixels to be added to the current destination offset following each space character. If both an character pad and a space pad are specified, the pixels specified by the character pad and the pixels specified by the space pad are added after space characters.

The VSTA$PRINT_TEXT Procedure

The VSTA$PRINT_TEXT procedure implements the print text operation.

Format

```
Status = VSTA$PRINT_TEXT (source-type, source-image, mask-type, mask-font, dest-type,
         dest-image, dest-offset-action, dest-offset, map-type, [reserved1], [reserved2], text-type,
         string, control-count, control-list, character-pad, space-pad, num-of-rectangles,
         clipping-rectangles [wait-flag], [efn], [astadr], [astprm], [iosb])
```

Parameters

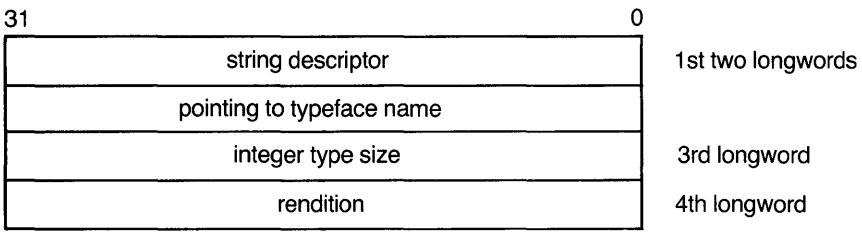
source-type. One of five defined symbols that specify the five source image types. Passed by reference.

VSTA$K_SRC_CONST	Source image is a constant. A mask font must be specified to define the characters.
VSTA$K_SRC_HT_BITMAP	Source image is a bitmap containing a halftone pattern. A mask font must be specified to define the characters.
VSTA$K_SRC_HT_VD	Source image is a virtual display containing a halftone pattern. A mask font must be specified to define the characters.
VSTA$K_SRC_FONT_BITMAP	Source image is a program-defined font data structure. A mask font cannot be used with this source type.
VSTA$K_SRC_DEFINED_FONT	Source image is a defined font. A mask font cannot be used with this source type.

source-image. An integer constant, a bitmap descriptor, a channel number, or a defined font descriptor. Passed by reference. The source image must be the type specified by the source-type parameter.

The format of a program-defined font data structure is defined above.

A defined font descriptor is a four-longword block containing the following information:



See *Programming for the VAXstation Display System* for more information about typeface names, type sizes, and rendition symbols.

mask-type. One of three symbols that specify the three types of mask font.

VSTA$K_MSK_NONE	There is no mask font.
VSTA$K_MSK_FONT_BITMAP	Mask font is a program-supplied font data structure.
VSTA$K_MSK_DEFINED_FONT	Mask font is a defined font.

mask-font. A font data structure (if the mask type is a font bitmap) or a defined font descriptor (if the mask type is a defined font). Passed by reference. This parameter is ignored if no mask font is specified.

dest-type. One of two symbols that specify the two destination image types. Passed by reference.

VSTA$K_DST_BITMAP	Destination is a bitmap.
VSTA$K_DST_VD	Destination is a virtual display.

dest-image. The destination image. Passed by reference. If the destination is a bitmap, the dest-image parameter is a bitmap descriptor. If the destination is a virtual display, the parameter is a channel number.

dest-offset-action. One of two symbols that specify whether the destination offsets are to be updated when the output is completed.

VSTA$K_DST_UPDATE	Update the destination x and y offsets.
VSTA$K_DST_NO_UPDATE	Do not update the destination offsets.

dest-offset. A two-word block containing the x and y offsets for the first character in the text string. If updating is requested, the x and y offsets after the text is printed are returned in this block. Passed by reference.

map-type. One of 16 symbols that specify the 16 map function types. Passed by reference. The symbols are:

VSTA$K_MAP_SRC	No transformation.
VSTA$K_MAP_NOTSRC	NOT source.
VSTA$K_MAP_DST	No transformation; no effect.
VSTA$K_MAP_NOTDST	NOT destination.
VSTA$K_MAP_SRC_AND_DST	Source AND destination.
VSTA$K_MAP_NOTSRC_AND_DST	NOT source AND destination.
VSTA$K_MAP_SRC_AND_NOTDST	Source AND NOT destination.
VSTA$K_MAP_NOTSRC_AND_NOTDST	NOT source AND NOT destination.
VSTA$K_MAP_SRC_OR_DST	Source OR destination.
VSTA$K_MAP_NOTSRC_OR_DST	NOT source OR destination.
VSTA$K_MAP_SRC_OR_NOTDST	Source OR NOT destination.
VSTA$K_MAP_NOTSRC_OR_NOTDST	NOT source OR NOT destination.
VSTA$K_MAP_SRC_XOR_DST	Source XOR destination.
VSTA$K_MAP_NOT_SRCXORDST	NOT (source XOR destination).
VSTA$K_MAP_BLACK	Replace all destination pixel values with zero.
VSTA$K_MAP_WHITE	Replace all destination pixel values with one.

[reserved1, reserved2]. Reserved for future use.

text-type. One of two symbols that specify the two types of indexing in the font.

VSTA$K_TXT_8BITS	Font is indexed by 8-bit text codes.
VSTA$K_TXT_16BITS	Font is indexed by 16-bit text codes.

string. A text string. Passed by descriptor.

control-count. An integer specifying the number of words in the control string. Passed by reference. If the control count is zero, the entire text string is printed with no special controls.

control-list. A list of 16-bit words containing control commands. Each command requires from one to three words. Passed by reference.

See the description of control commands above for the command opcodes and the required operands. Opcodes 0 through 3 are currently used. All others are reserved.

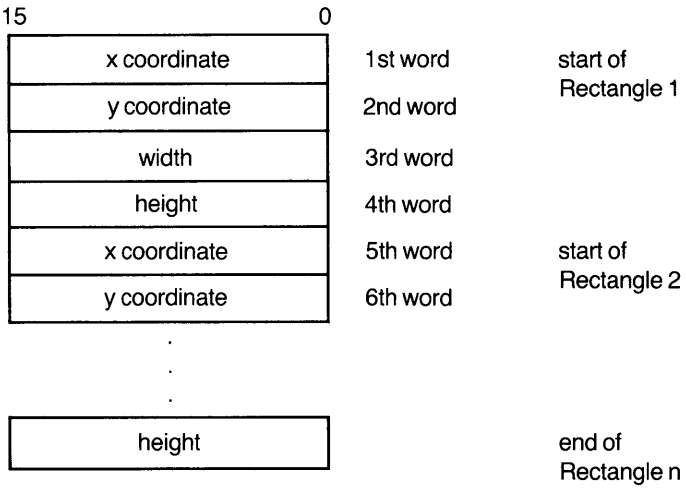
character-pad. An integer specifying the number of pixels to insert after every character, including the last. Passed by reference.

space-pad. An integer specifying the number of pixels to insert after every space character. The pixels specified for the character pad are also inserted after space characters. Passed by reference.

num-of-rectangles. An integer specifying the number of rectangles in the clipping rectangle list. Passed by reference.

clipping-rectangles. A list of rectangles in the destination image. Passed by reference.

The format of the rectangle list is:



The rectangles must not intersect.

[wait-flag]. An integer flag specifying whether the procedure returns control to the calling program after queuing the output request, or waits until I/O completion. Passed by reference.

This procedure uses either VMS event flag zero or the specified event flag as the wait flag.

[efn]. An event flag number. The specified event flag is cleared when the I/O request is queued, and set when the I/O is completed. Passed by reference.

The default event flag is event flag zero. Event flag zero is also the default event flag for all VMS system services that use event flags and for the standard VSTA procedures that read input from the mouse, the keyboard, and the tablet. Specifying another event flag is recommended (even if the event flag is not tested) to avoid possible conflicts.

[astadr]. The address of the AST service routine to be executed when I/O is completed. See the appropriate language reference manual for the method of passing routine addresses. NOTE: Zero must be passed by value to specify that there is no AST routine.

[astprm]. An integer value to be passed as a parameter to the AST routine. Passed by reference. NOTE: The parameter is passed to the AST routine by value.

[iosb]. An I/O status block. Passed by reference.

Return Status

VSTA$_PRITXT
Message: Print text failed.

Example Code

The following FORTRAN subroutine prints a text string using a defined font as a mask font. Appendix B shows a complete program that calls this subroutine.

Example 4–1: Using VSTA$PRINT_TEXT

```
!-----------------------------------------------------------------------
      SUBROUTINE print_line (vd_id, dest_offset, text)
!-----------------------------------------------------------------------
!
! Function:
!
!      Print a text string on vd_id at dest_offset in the defined
!      font KILTER.
!
! Input Args:
!
!      vd_id = virtual display id
!      dest_offset = destination offset
!      text = string to be printed
!
! Output Args:  none
!
      IMPLICIT NONE
!
! Include the VAXstation defined symbols
!
      INCLUDE 'SYS$LIBRARY:VSTAGBL.FOR'
!
! Argument declarations:
!
      INTEGER*4     vd_id, dest_offset
      CHARACTER*(*) text
!
! Declarations:
!
      INTEGER*4 status, VSTA$PRINT_TEXT
      INTEGER*4     Font_desc(4)
      CHARACTER*6   Font_TypeFace              ! Font Typeface.

!
! Set up font descriptor.
!
      Font_TypeFace = 'KILTER'
      Font_desc(1) = len(Font_TypeFace)   ! Length of font name.
      Font_desc(2) = %LOC(Font_TypeFace)  ! KILTER font type face.
      Font_desc(3) = 14                ! Font size.
      Font_desc(4) = 0                 ! Font rendition code.

      Status = VSTA$PRINT_TEXT (
      1              VSTA$K_SRC_CONST, 16,
      2              VSTA$K_MSK_DEFINED_FONT, font_desc,
      3              VSTA$K_DST_VD, vd_id,
      4              VSTA$K_DST_NO_UPDATE, dest_offset,
      5              VSTA$K_MAP_SRC, , ,
      6              VSTA$K_TXT_8BITS, text,
      7              0, , 0, 0, 0, , 1, 5, , , )
             IF ( .NOT. status ) CALL show_error(status)

      RETURN
      END
```

5

Filling an Area

Description

Filling an area uses a color or halftone source image to fill one or more closed shapes, then copies the shapes to the specified location in a destination image.

Filling an area is used when the boundary of the area to be filled is known and can be defined by a list of straight or curved segments. Flooding an area, on the other hand, is used when the boundary is not completely known, but the user can specify one internal point of the closed area.

The source image for filling an area is a constant or halftone. A path, as in drawing a curve, specifies a closed area in the destination image. Thus, filling an area causes one or more areas of a destination image to be filled with one or more colored shapes.

The exact behavior of filling an area is determined by:

- Source type and source image
- Destination type and destination image
- Destination offset
- Map type
- Clipping rectangle
- Path

The destination type and image, the destination offset, and the map type are the same as for copying an area. The function of the clipping rectangle is the same as for copying an area. However, only one clipping rectangle can be specified for filling an area.

Source

The three types of source image for filling an area are:

- Constant
- Halftone bitmap
- Halftone virtual display

The source specifies the constant pixel value or halftone with which the closed area specified by the path is filled. Bitmap or virtual display source images are not allowed.

Path

The path defines one or more closed areas in the destination to be filled by the pixel values specified by the source. The path is defined by a path list that is the same as the path list for drawing a curve. Each segment of the path can be a straight or a curved line.

Several closed areas can be filled with one path. In this case, the start point and the end point of each closed area must be specified with the start-closed-figure and end-closed-figure flags. The move flag specifies an invisible line from the end of one closed figure to the start of another.

Figure 5–1 shows the two filled diamonds generated by the following path.

segment	x	y	flags
1	25	10	origin, move, start closed figure
2	10	40	origin, draw
3	25	70	origin, draw
4	40	40	origin, draw
5	25	10	origin, draw, end closed figure
6	65	10	origin, move, start closed figure
7	50	40	origin, draw
8	65	70	origin, draw
9	80	40	origin, draw
10	65	10	origin, draw, end closed figure

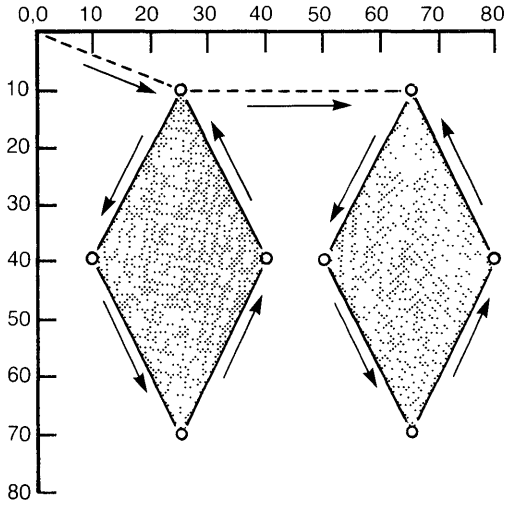


Figure 5–1. Filling Two Closed Areas

If the start- and end-closed-figure flags are not used, or if the move flag is used to draw an invisible line to any point except one that starts a closed figure, the results are unpredictable.

The VSTA$FILL_AREA Procedure

The VSTA$FILL_AREA procedure implements the fill-area operation.

Format

Status = VSTA$FILL_AREA (source-type, source-image, dest-type, dest-image, dest-offset-x, dest-offset-y, map-type, [reserved1], [reserved2], num-of-points, path, num-of-rectangles, clipping-rectangles [wait-flag], [efn], [astadr], [astprm], [iosb])

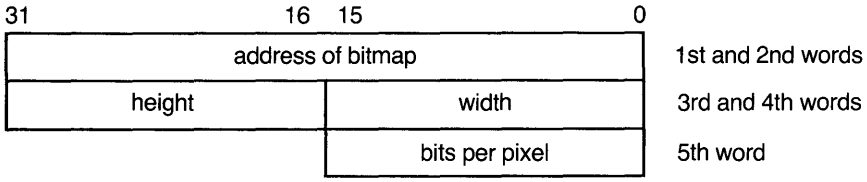
Parameters

source-type. One of three defined symbols that specify the three source image types. Passed by reference.

VSTA$K_SRC_CONST	Source image is a constant.
VSTA$K_SRC_HT_BITMAP	Source image is a bitmap containing a halftone pattern.
VSTA$K_SRC_HT_VD	Source image is a virtual display containing a halftone pattern.

source-image. An integer constant, a bitmap descriptor, or a channel number. Passed by reference. The source image must be the type specified by the source-type parameter.

A bitmap descriptor is a five-word block in the following format:



dest-type. One of two symbols that specify the two destination image types. Passed by reference.

VSTA$K_DST_BITMAP	Destination is a bitmap.
VSTA$K_DST_VD	Destination is a virtual display.

dest-image. A bitmap descriptor or an integer channel number. Passed by reference.

dest-offset-x, dest-offset-y. Integers specifying horizontal and vertical offsets from the upper left corner of the destination image. The filled shapes generated by the source and path parameters are placed relative to the destination offset. Passed by reference.

map-type. One of 16 symbols that specify the 16 map function types. Passed by reference.

VSTA$K_MAP_SRC	No transformation.
VSTA$K_MAP_NOTSRC	NOT source.
VSTA$K_MAP_DST	No transformation; no effect.
VSTA$K_MAP_NOTDST	NOT destination.
VSTA$K_MAP_SRC_AND_DST	Source AND destination.
VSTA$K_MAP_NOTSRC_AND_DST	NOT source AND destination.
VSTA$K_MAP_SRC_AND_NOTDST	Source AND NOT destination.
VSTA$K_MAP_NOTSRC_AND_NOTDST	NOT source AND NOT destination.

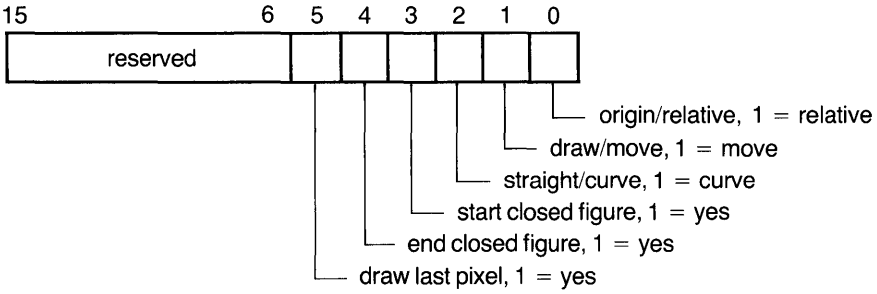
VSTA$K_MAP_SRC_OR_DST	Source OR destination.
VSTA$K_MAP_NOTSRC_OR_DST	NOT source OR destination.
VSTA$K_MAP_SRC_OR_NOTDST	Source OR NOT destination.
VSTA$K_MAP_NOTSRC_OR_NOTDST	NOT source OR NOT destination.
VSTA$K_MAP_SRC_XOR_DST	Source XOR destination.
VSTA$K_MAP_NOT_SRCXORDST	NOT (source XOR destination).
VSTA$K_MAP_BLACK	Replace all destination pixel values with zero.
VSTA$K_MAP_WHITE	Replace all destination pixel values with one.

[reserved1, reserved2]. Reserved for future use.

num-of-points. An integer specifying the number of points in the path. Passed by reference.

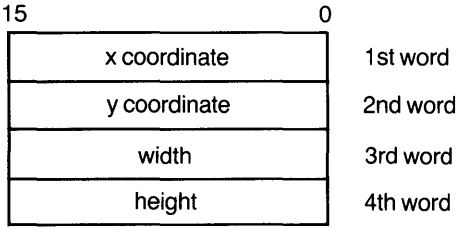
path. A list of points, each of which consists of three 16-bit words containing a signed integer x value, a signed integer y value, and a flag word. Passed by reference.

The format of the flag word is:



num-of-rectangles. An integer specifying either one clipping rectangle or no clipping rectangles. Passed by reference.

clipping-rectangles. A clipping rectangle in the following format:



Passed by reference.

[wait-flag]. An integer flag specifying whether the procedure returns control to the calling program after queuing the output request, or waits until I/O completion. Passed by reference.

This procedure uses either event flag zero or the specified event flag as the wait flag.

[efn]. An event flag number. The specified event flag is cleared when the I/O request is queued, and set when the I/O is completed. Passed by reference.

The default event flag is event flag zero. Event flag zero is also the default event flag for all VMS system services that use event flags and for the standard VSTA

procedures that read input from the mouse, the keyboard, and the tablet. Specifying another event flag is recommended (even if the event flag is not tested) to avoid possible conflicts.

[astadr]. The address of the AST service routine to be executed when I/O is completed. See the appropriate language reference manual for the method of passing routine addresses. NOTE: Zero must be passed by value to specify that there is no AST routine.

[astprm]. An integer value to be passed as a parameter to the AST routine. Passed by reference. NOTE: The parameter is passed to the AST routine by value.

[iosb]. An I/O status block. Passed by reference.

Return Status

VSTA$_FILARE
Message: Fill area failed.

Example Code

The following FORTRAN subroutine illustrates both the VSTA$FILL_AREA procedure and the VSTA$FLOOD_AREA procedure discussed in Chapter 6. The example program listed in Appendix B calls this subroutine.

Example 5–1. Using VSTA$FILL_AREA

```
!-----------------------------------------------------------------------
      SUBROUTINE fill_polygon( vd_id, xsize, ysize,
     1                                 background_color,
     2                                 foreground_color, flood_color)
!-----------------------------------------------------------------------
!
! Function:
!
!      Use FILL_AREA to draw a solid square, then use FLOOD_AREA to
!              flood the background.
!
! Input:
!
!      vd_id = channel number of virtual display
!      xsize = x dimension of virtual display
!      ysize = y dimension of virtual display
!      background_color = background color
!      foreground_color = color to fill rectangle
!      flood_color = color to flood background
!
      IMPLICIT NONE
!
! Include the VAXstation native mode graphics symbol table.
!
      INCLUDE 'SYS$LIBRARY:VSTAGBL.FOR'
!
! Argument declarations.
!
      INTEGER*4       vd_id, xsize, ysize, background_color,
     1                foreground_color, flood_color
```

```
!
! Routine declarations.
!
      BYTE            map
      INTEGER*4       mapsize, nclip, status,
      1               VSTA$FILL_AREA, VSTA$FLOOD_AREA
      INTEGER*2       pathlist(3,13), patternstate(2),
      1               patternblock(3), doff(2)

      DATA   patternstate  / 2 * 0 /,
      1      patternblock  / 3 * 0 /
!
! Fill a pathlist to form a rectangle based on vd size.
!
      pathlist(1,1) = xsize/4
      pathlist(2,1) = ysize/4
      pathlist(3,1) = 10
      pathlist(1,2) = xsize * 3 / 4
      pathlist(2,2) = pathlist(2,1)
      pathlist(3,2) = 0
      pathlist(1,3) = pathlist(1,2)
      pathlist(2,3) = ysize * 3 / 4
      pathlist(3,3) = 0
      pathlist(1,4) = pathlist(1,1)
      pathlist(2,4) = pathlist(2,3)
      pathlist(3,4) = 0
      pathlist(1,5) = pathlist(1,1)
      pathlist(2,5) = pathlist(2,1)
      pathlist(3,5) = 16
!
! Fill the rectangle.
!
      nclip = 0
      status = VSTA$FILL_AREA (
      1             VSTA$K_SRC_CONST, foreground_color,
      2             VSTA$K_DST_VD, vd_id, 0, 0,
      3             VSTA$K_MAP_SRC, , ,
      4             5, pathlist,
      5             nclip, , 1, 5)
      IF ( .NOT. status ) CALL show_error(status)
!
! Write some text on the square.
!
      doff(1) = xsize/2 - 30
      doff(2) = ysize/2
      CALL print_line ( vd_id, doff, 'fill')
!
! Set the boundary map for VSTA$FLOOD_AREA such that the color that
! the rectangle was filled with is the boundary, thus preventing
! leaking if the background color is a halftone.
!
      IF ( foreground_color .EQ. 0 ) THEN
          map = 1
      ELSE
          map = 2
      ENDIF
      mapsize = 1
```

```
!
! Flood the background.
!
      status = VSTA$FLOOD_AREA (
      1              VSTA$K_SRC_CONST, flood_color,
      2              VSTA$K_DST_VD, vd_id,
      3              mapsize, map,
      4              1, 1,
      5              nclip, , 1, 5)
      IF ( .NOT. status ) CALL show_error(status)
!
! Write some text on the background.
!
      doff(1) = xsize/2 - 30
      doff (2) = ysize * 7 / 8
      CALL print_line ( vd_id, doff, 'flood')
! End of routine.
!
      RETURN
      END
```

6

Flooding an Area

Description

The flood area operation floods bounded areas of a destination image with a single color or a halftone. The result of flooding an area is the same as the result of filling an area. However, filling an area requires that the program specify the area to be filled. The area to be flooded depends on the current state of the destination image and is determined by a flood algorithm.

Flooding is often used for interactive applications that allow the user to modify the image on the screen. The area to be flooded may be obvious to a user who has drawn the outlines of several objects and wants one of the outlines to be filled in with a solid color. To flood the area, that program can ask the user to specify an interior point. To fill the area, the program would have to construct a path from a history of the user's actions.

The flood algorithm determines the area of the destination to be flooded. The algorithm locates the inside and outside portion of the closed area and selects those pixels inside the flooded area.

The determination of the bounded area requires a seed point and a boundary map. The seed point specifies a single pixel in the destination image; this point must be in the bounded area.

The boundary map is a table of zeros and ones that determines whether points are interior or boundary points. The algorithm examines the eight pixels adjacent to the seed point (the seed point's 'neighbors'). The value of each pixel is an index into the boundary map table. If the corresponding value in the boundary map table is zero, then this point is an inside point; otherwise, it is a boundary point. If an inside point is found, the algorithm examines its neighbors. Processing continues until the neighbors of all internal points have been examined.

If the seed point is found to lie on a boundary, the algorithm terminates immediately.

Once the boundary has been determined, the inside area is flooded with the source.

The exact behavior of flooding an area is determined by:

- Source
- Destination
- Seed Point
- Clipping Rectangle
- Boundary Map

Source

The source image can be one of three types:

- A constant
- A halftone bitmap
- A halftone virtual display

The specified area in the destination is flooded with the color specified by the constant, or with the halftone in the bitmap or virtual display.

Since the source for flooding an area specifies only a color or halftone, a source mask and a source offset are not used.

Destination

The two types of destination image are bitmaps and virtual displays. The flood algorithm determines the area to be flooded according to the values of the pixels in the destination.

Seed Point

The seed point is a single point in the destination image that lies within the area to be flooded.

Clipping Rectangle

Only one clipping rectangle can be used with the flood area operation. If a clipping rectangle is specified, the operation is constrained by the clipping area. If no clipping rectangle is specified, the operation is constrained by the size of the destination image.

Boundary Map

The boundary map is a binary-valued table defining the internal and external points of the closed figure in the destination bitmap. Bit i in the table indicates whether pixels of value i are interior points (to be flooded) or boundary points. Any pixel value mapping to a zero entry in the table is an interior point; any value mapping to a one is a boundary point.

In general, in an n-bit-per-pixel system, 2^n bits are required to specify whether each of the possible 2^n pixel values is a boundary. For example, a system with three bits per pixel requires eight entries in the boundary map. Each of the possible pixel values is an index into the table; the zero and one entries in the table

specify whether a pixel with the index value is an interior pixel or a boundary pixel. In the following boundary map, pixel values 0, 2, 3, and 5 are boundary values; and pixel values 1, 4, 6, and 7 are interior values.

Pixel value (index)	7	6	5	4	3	2	1	0
Table entry	0	0	1	0	1	1	0	1

Figure 6–1 shows a destination image with two overlapping, irregular, closed curves. The background is white (pixel value 1); the lines are black (pixel value 0). If the boundary map table shown above is specified, pixels whose value is 1 are interior; pixels whose values are 0 are the boundary.

The seed point is location 45,50.

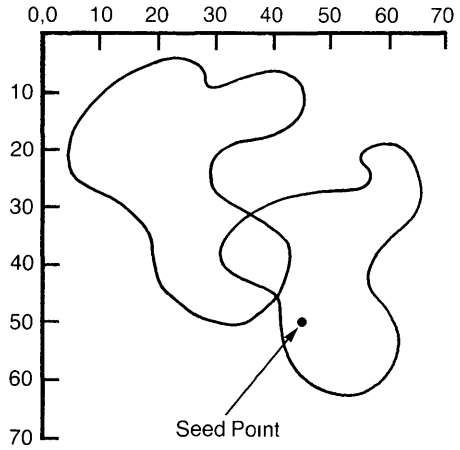


Figure 6–1. Destination Image before Flooding

The pixel at 45,50 is surrounded by pixels whose values are 1 (and therefore interior), so their neighbors are examined. Moving left from the seed point, the boundary is reached at 41,50.

Figure 6–2 shows the destination image after flooding.

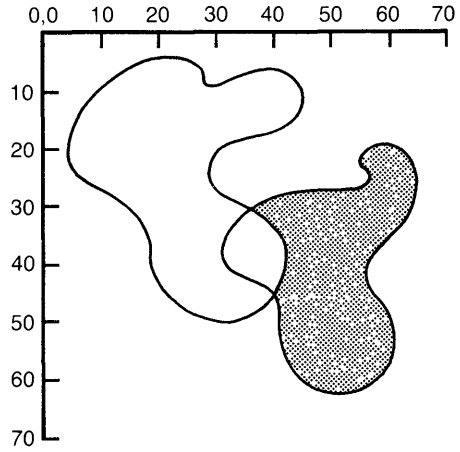


Figure 6–2. Destination Image after Flooding

The VSTA$FLOOD_AREA Procedure

The VSTA$FLOOD_AREA procedure implements the flood-area operation.

Format

Status = VSTA$FLOOD_AREA (source-type, source-image, dest-type, dest-image, boundary-map-size, boundary-map, seedpoint-x, seedpoint-y, num-of-rectangles, clipping-rectangles [wait-flag], [efn], [astadr], [astprm], [iosb])

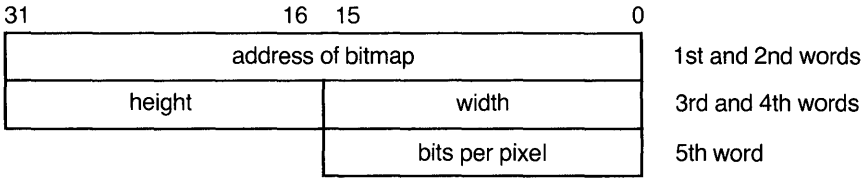
Parameters

source-type. One of three symbols that specify the three source image types. Passed by reference.

VSTA$K_SRC_CONST	Source image is a constant.
VSTA$K_SRC_HT_BITMAP	Source image is a bitmap containing a halftone pattern.
VSTA$K_SRC_HT_VD	Source image is a virtual display containing a halftone pattern.

source-image. An integer constant, a bitmap descriptor, or a channel number. Passed by reference. The source image must be the type specified by the source-type parameter.

A bitmap descriptor is a five-word block in the following format:



dest-type. One of two symbols that specify the two destination image types. Passed by reference.

VSTA$K_DST_BITMAP	Destination is a bitmap.
VSTA$K_DST_VD	Destination is a virtual display.

dest-image. The destination image. Passed by reference. If the destination is a bitmap, the dest-image parameter is a bitmap descriptor. If the destination is a virtual display, the parameter is a channel number.

boundary-map-size. An integer specifying the size of the boundary map in bytes. Passed by reference.

An error is generated if the specified size does not agree with the size calculated from the depth of the destination image.

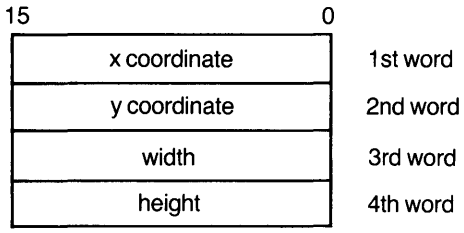
boundary-map. A bit table in which the zeros and ones represent pixel values that are interior points or boundary points. Passed by reference.

seedpoint-x. An integer specifying the x coordinate of the seed point. Passed by reference.

seedpoint-y. An integer specifying the y coordinate of the seed point. Passed by reference.

num-of-rectangles. An integer specifying either one clipping rectangle or no clipping rectangles. Passed by reference.

clipping-rectangles. A clipping rectangle in the following format:



Passed by reference. This parameter is ignored if the number of clipping rectangles is zero.

[wait-flag]. An integer flag specifying whether the procedure returns control to the calling program after queuing the output request, or waits until I/O completion. Passed by reference.

This procedure uses either event flag zero or the specified event flag as the wait flag.

[efn]. An event flag number. The specified event flag is cleared when the I/O request is queued, and set when the I/O is completed. Passed by reference.

The default event flag is event flag zero. Event flag zero is also the default event flag for all VMS system services that use event flags and for the standard VSTA procedures that read input from the mouse, the keyboard, and the tablet. Specifying another event flag is recommended (even if the event flag is not tested) to avoid possible conflicts.

[astadr]. The address of the AST service routine to be executed when I/O is completed. See the appropriate language reference manual for the method of passing routine addresses. NOTE: Zero must be passed by value to specify that there is no AST routine.

[astprm]. An integer value to be passed as a parameter to the AST routine. Passed by reference. NOTE: The parameter is passed to the AST routine by value.

[iosb]. An I/O status block. Passed by reference.

Return Status

VSTA$_FLDARE
Message: Flood area failed.

Example Code

The example code for Chapter 5 shows the VSTA$FLOOD_AREA procedure as well as the VSTA$FILL_AREA procedure.

A

Hardware Model Summary

This appendix lists the curent VAXstation Display System models and gives information about using the native graphics procedures that applies only to a specific model.

VAXstation 100 Display System

Constant Source Values

The value specified by a constant source is an index into a color lookup table. The corresponding entry in the table specifies the color. On the VAXstation 100, the range of constant values is 0 to 16. The constant 0 selects black; the constant 16 selects white. The constants from 1 through 15 select halftones. The halftone selected by the constant 1 has the highest proportion of black; the halftone selected by 15 has the highest proportion of white. The halftone selected by 8 is half and half.

Bitmap Storage Requirements

Bitmaps for the VAXstation 100 use one bit per pixel. The VAXstation 100 represents a bitmap as a sequence of horizontal scan lines stored in contiguous memory locations. Each scan line must begin on the boundary of a 16-bit word. That is, although a bitmap can have any horizontal width in pixels, the storage in which the bitmap is kept must have sufficient space so that each horizontal line can be word-aligned. If the horizontal width in pixels is not evenly divisible by 16, the last bits in the last word of the storage for each horizontal line are not used. For any bitmap of dimensions (X,Y) on the VAXstation 100, the storage requirement is ((X+15)/16)*Y words.

Halftone Representation

The VAXstation 100 uses only a single format for a halftone bitmap or a halftone virtual display. The halftone pattern must be specified as a square bitmap 16 pixels on a side. Therefore, to use a "standard" 4-by-4 halftone pattern, the pattern is replicated horizontally and vertically to form a 16-by-16 pattern.

B

Example Program

The FORTRAN example program listed in this appendix draws the illustrations in Figures B–1 and B–2. Figure B–1 shows the screen the first time the subroutine wait is called. Figure B–2 shows the screen the second time the same subroutine is called. The subroutines listed in the example code sections of Chapter 2 (copy_bitmap), Chapter 3 (draw_circle), Chapter 4 (print_line), and Chapter 5 (fill_polygon) are called by this program. These subroutines are not included in the appendix.

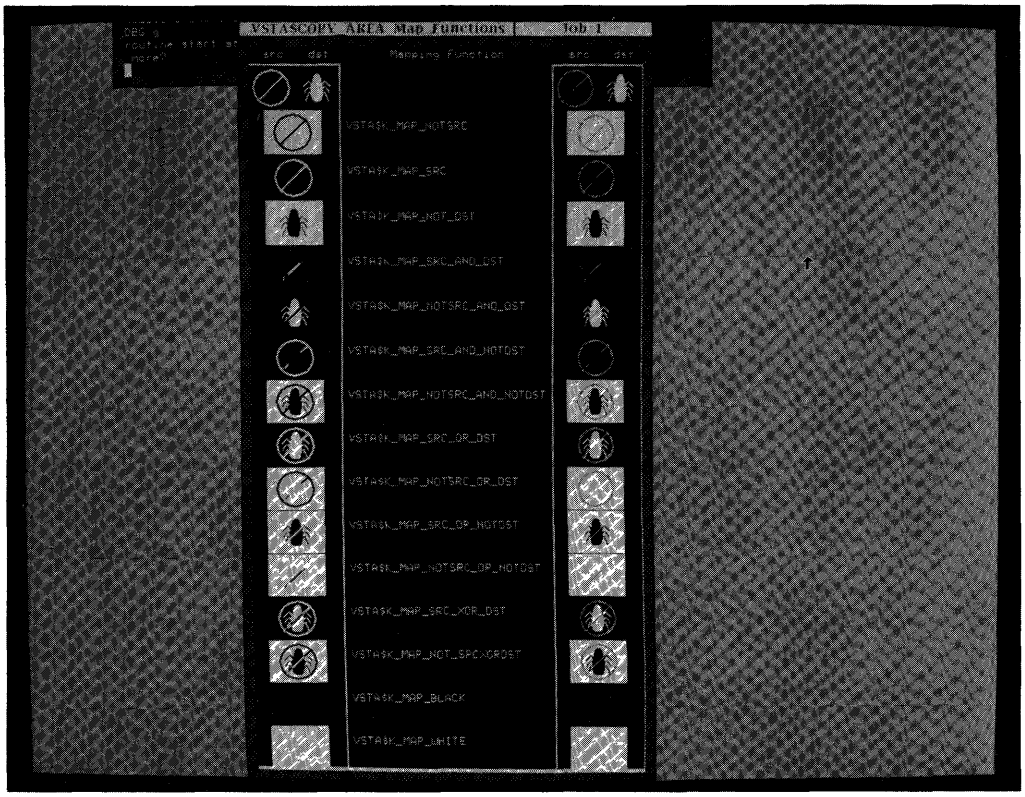


Figure B–1. Mapping Functions

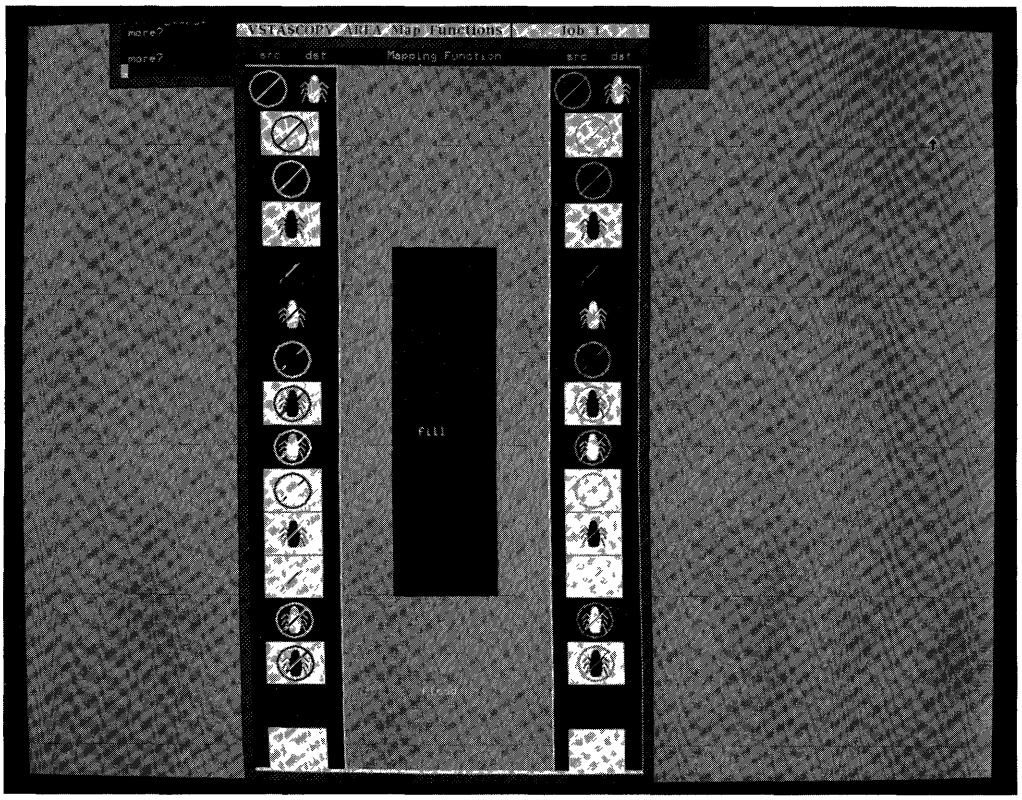


Figure B–2. Fill and Flood

```
!-----------------------------------------------------------------------
      PROGRAM bugandbar
!-----------------------------------------------------------------------
!
! Function:
!
!      Demonstrate the mapping functions of VSTA$COPY_AREA.
!
      IMPLICIT NONE
!
! Include the Vaxstation defined symbols.
!
      INCLUDE 'SYS$LIBRARY:VSTAGBL.FOR'
!
! Routine declarations.
!
      INTEGER*4     mapvalue(16)
      INTEGER*2     bug_descr(5), bar_descr(5), doff(2)
      INTEGER*4     vd_ids(4), bug(2,49), bug_address,
     1              bar_address, vd_x_size, vd_y_size, bar_color,
     2              i, j, offx, offy, origoffy, ind
      INTEGER*4     status, VSTA$COPY_AREA
      EQUIVALENCE   ( bug_address, bug_descr(1) ),
     1              ( bar_address, bar_descr(1) )
      CHARACTER*28  maptext(16)
      CHARACTER*10  ichar
```

```
INTEGER*4     xoff, yoff
INTEGER*4     bar1(2,49), bar2(2,49)
INTEGER*4     scratch(2,49), scratch_address
INTEGER*2     scratch_descr(5), mask(2)
EQUIVALENCE   (scratch_address, scratch_descr(1))
COMMON        scratch_descr, scratch

DATA   mapvalue /
1      VSTA$K_MAP_DST,
2      VSTA$K_MAP_NOTSRC,
3      VSTA$K_MAP_SRC,
4      VSTA$K_MAP_NOTDST,
5      VSTA$K_MAP_SRC_AND_DST,
6      VSTA$K_MAP_NOTSRC_AND_DST,
7      VSTA$K_MAP_SRC_AND_NOTDST,
8      VSTA$K_MAP_NOTSRC_AND_NOTDST,
9      VSTA$K_MAP_SRC_OR_DST,
1      VSTA$K_MAP_NOTSRC_OR_DST,
1      VSTA$K_MAP_SRC_OR_NOTDST,
2      VSTA$K_MAP_NOTSRC_OR_NOTDST,
3      VSTA$K_MAP_SRC_XOR_DST,
4      VSTA$K_MAP_NOT_SRCXORDST,
5      VSTA$K_MAP_BLACK,
6      VSTA$K_MAP_WHITE /

DATA   maptext /
1      'VSTA$K_MAP_DST              ',
2      'VSTA$K_MAP_NOTSRC           ',
3      'VSTA$K_MAP_SRC              ',
4      'VSTA$K_MAP_NOTDST           ',
5      'VSTA$K_MAP_SRC_AND_DST      ',
6      'VSTA$K_MAP_NOTSRC_AND_DST   ',
7      'VSTA$K_MAP_SRC_AND_NOTDST   ',
8      'VSTA$K_MAP_NOTSRC_AND_NOTDST',
9      'VSTA$K_MAP_SRC_OR_DST       ',
1      'VSTA$K_MAP_NOTSRC_OR_DST    ',
1      'VSTA$K_MAP_SRC_OR_NOTDST    ',
2      'VSTA$K_MAP_NOTSRC_OR_NOTDST ',
3      'VSTA$K_MAP_SRC_XOR_DST      ',
4      'VSTA$K_MAP_NOT_SRCXORDST    ',
5      'VSTA$K_MAP_BLACK            ',
6      'VSTA$K_MAP_WHITE            ' /

DATA   bug/
1 '00000000'x,'00000000'x, '00000000'x,'00000000'x,
1 '00000000'x,'00000000'x, '00000000'x,'00000000'x,
1 '00000000'x,'00000000'x, '00000000'x,'00000000'x,
1 '00000000'x,'00000000'x, '00000000'x,'00000000'x,
1 '00000000'x,'00000000'x, 'C0000000'x,'00000003'x,
1 'E0000000'x,'00000007'x, 'F0000000'x,'0000000F'x,
1 'F0000000'x,'0000000F'x, 'F0000000'x,'0000000F'x,
2 'F8000000'x,'0000001F'x, 'F8000000'x,'0000001F'x,
2 'F8000000'x,'0000001F'x, 'F8C00000'x,'0000031F'x,
3 'F9A00000'x,'0000059F'x, 'FF300000'x,'00000CFF'x,
3 'FE180000'x,'0000187F'x, 'FE0C0000'x,'0000307F'x,
4 'FC060000'x,'0000603F'x, 'FE420000'x,'0000427F'x,
4 'FFA00000'x,'000005FF'x, 'FF300000'x, '00000CFF'x,
5 'FE180000'x,'0000187F'x, 'FE080000'x,'0000107F'x,
5 'FE0C0000'x,'0000307F'x, 'FEC40000'x,'0000237F'x,
```

```
     6 'FF440000'x,'000022FF'x, 'FE400000'x,'0000027F'x,
     6 'FE600000'x,'000006?F'x, 'FE200000'x,'00000047F'x,
     7 'FC300000'x,'00000C3F'x, 'FC100000'x,'000008?F'x,
     7 'FC100000'x,'000008?F'x, 'FC180000'x,'000018?F'x,
     8 'F8080000'x,'00001?1F'x, 'F0080000'x,'00001?0F'x,
     8 '00000000'x,'00000000'x,'00000000'x,'00000000'x,
     8 '00000000'x,'00000000'x,'00000000'x,'00000000'x,
     8 '00000000'x,'00000000'x,'00000000'x,'00000000'x,
     8 '00000000'x,'00000000'x,'00000000'x,'00000000'x,
     8             '00000000'x,'00000000'x /
!
! Create a pasteboard and 4 virtual displays.
!
      CALL vs_init4( 0, 16, vd_ids, vd_x_size, vd_y_size)
!
! Draw a white bar on one display, and a halftone 8 bar on another.
!
      CALL draw_bar(vd_ids(1), 49, 49, 16, xoff(1), yoff(1) )
      CALL draw_bar(vd_ids(2), 49, 49,  8, xoff(1), yoff(1) )
! Copy the bars from the displays to bitmaps.

      bar_address = %LOC(bar1)    ! bitmap descriptor
      bar_descr(3) = 64           ! x
      bar_descr(4) = 49           ! y
      bar_descr(5) = 1            ! z
      status = VSTA$COPY_AREA(
      1             VSTA$K_SRC_VD, vd_ids(1), 0, 0,
      2             VSTA$K_MSK_NONE, ,
      3             VSTA$K_DST_BITMAP, bar_descr, ?, 0,
      4             VSTA$K_MAP_SRC, , ,
      5             0, , 1, 5, , , )
      IF ( .NOT. status ) CALL show_error(status)

      bar_address = %LOC(bar2)
      status = VSTA$COPY_AREA(
      1             VSTA$K_SRC_VD, vd_ids(2), 0, 0,
      2             VSTA$K_MSK_NONE, ,
      3             VSTA$K_DST_BITMAP, bar_descr, ?, 0,
      4             VSTA$K_MAP_SRC, , ,
      5             0, , 1, 5, , , )
      IF ( .NOT. status ) CALL show_error(status)
!
! Draw two bugs.  Generate a bitmap descriptor for the bug bitmap
! and copy the bitmap next to the bars at the top of the virtual
! displays.
!
      bug_address = %LOC(bug)
      bug_descr(3) = 64
      bug_descr(4) = 49
      bug_descr(5) = 1
      status = VSTA$COPY_AREA(
      1             VSTA$K_SRC_BM, bug_descr, 0, 0,
      2             VSTA$K_MSK_NONE, ,
      3             VSTA$K_DST_VD, vd_ids(1), 43, 0,
      4             VSTA$K_MAP_SRC_OR_DST, , ,
      5             0, , 1, 5, , , )
      IF ( .NOT. status ) CALL show_error(status)
```

```
      status = VSTA$COPY_AREA(
     1             VSTA$K_SRC_BM, bug_descr, 0, 0,
     2             VSTA$K_MSK_NONE, ,
     3             VSTA$K_DST_VD, vd_ids(2), 43, 0,
     4             VSTA$K_MAP_SRC_OR_DST, , ,
     5             0, , 1, 5, , , )
      IF ( .NOT. status ) CALL show_error(status)
!
! Label the columns.
!
      doff(1) = 15
      doff(2) = 0
      CALL print_line (vd_ids(3), doff, 'src')
      doff(1) = 355
      CALL print_line (vd_ids(3), doff, 'src')
      doff(1) = 65
      CALL print_line (vd_ids(3), doff, 'dst')
      doff(1) = 405
      CALL print_line (vd_ids(3), doff, 'dst')
      doff(1) = 155
      CALL print_line (vd_ids(3), doff, 'Mapping Function')
!
! Copy bars onto bugs on scratch bitmap using all the map functions,
! then copy scratch bitmap to displays at appropriate offsets.
!
      doff(1) = 5
      DO i = 2, 16
          offy = yoff(i)
          DO j = 1, 2
             offx = xoff(i)
!
! Copy bug to scratch bitmap.
!
      CALL copy_bitmap (bug_descr, VSTA$K_MAP_SRC)
!
! Copy either white bar or halftone bar to scratch bitmap using one
! of the map functions.
!
             bar_address = %LOC(bar1)
             IF ( j .eq. 2 ) bar_address = %LOC(bar2)
             CALL copy_bitmap (bar_descr, mapvalue(i) )
!
! Copy scratch bitmap to left display or right display at appropriate
! destination offset.
!
             status = VSTA$COPY_AREA(
      1             VSTA$K_SRC_BM, scratch_descr, 0, 0,
      2             VSTA$K_MSK_NONE, ,
      3             VSTA$K_DST_VD, vd_ids(j), 16, offy,
      4             VSTA$K_MAP_SRC, , ,
      5             0, , 1, 5, , , )
             IF ( .NOT. status ) CALL show_error(status)
             doff(2) = (i-1)*50+ 10
!
! Print the symbol for the map function.
!
             CALL print_line (vd_ids(4), doff, maptext(i))
          ENDDO
      ENDDO
```

```
!
! Pause for user to see screen.
!
      CALL wait
!
! Make the center display white using a constant source and a rectangle
! mask the same size as the display.
!
  .   mask(1) = 235
      mask(2) = 800
      status = VSTA$COPY_AREA(
     1             VSTA$K_SRC_CONST, 16, 0, 0,
     2             VSTA$K_MSK_RECTANGLE, mask,
     3             VSTA$K_DST_VD, vd_ids(4), 0, 0,
     4             VSTA$K_MAP_SRC, , ,
     5             0, , 1, 5, , , )
      IF ( .NOT. status ) CALL show_error(status)

!
! Use fill and flood.
!
      CALL fill_polygon (vd_ids(4), 235, 800, 16, 0, 8)
!
! Pause for user to see illustration.
!
      CALL wait
      CALL EXIT
      END



!-------------------------------------------------------------------------
      SUBROUTINE VS_INIT4 ( screen_color, text_color,
     1                      vd_ids, vd_x_size, vd_y_size )
!-------------------------------------------------------------------------
!
! Function:
!
!     Create a pasteboard with 4 virtual displays.
!
! Input Args:
!
!     screen_color = screen color of the virtual displays (0 - 16)
!     text_color = text color within the virtual displays (0 - 16)
!
! Output Args:
!
!     vd_ids = array of channel numbers of virtual displays
!     vd_x_size = x size of mapping virtual displays in pixels
!     vd_y_size = y size of mapping virtual displays in pixels
!
      IMPLICIT NONE
!
! Include the Vaxstation defined symbols.
!
      INCLUDE 'SYS$LIBRARY:VSTAGBL.FOR'
```

```
!
! Argument declarations.
!
      INTEGER*4     screen_color, text_color
      INTEGER*4     vd_ids(4)
!
! Routine wide declarations.
!
      INTEGER*4     i, status, VSTA$CREATE_PASTEBOARD,
     1              VSTA$CREATE_DISPLAY, VSTA$PASTE_DISPLAY,
     2              VSTA$CREATE_WINDOW
      CHARACTER*2   ichar
!
! VSTA$CREATE_PASTEBOARD declarations.
!
      INTEGER*4     z_value, n_lines, line_height,
     1              x_size, y_size, pasteboard_id
!
! VSTA$PASTE_DISPLAY declarations.
!
      INTEGER*4     paste_x_offset, paste_y_offset
!
! VSTA$CREATE_WINDOW declarations.
!
      INTEGER*4     window_id
!
! VSTA$CREATE_DISPLAY declarations.
!
      INTEGER*4     vd_x_offset, vd_y_offset,
     1              vd_x_size, vd_y_size
!
! Define pasteboard and window data.
!
      DATA   z_value       /1/    ! for VS100
      DATA   n_lines       /0/    ! number of text text on VD
      DATA   line_height   /0/    ! line height in pixels
      DATA   paste_x_offset/0/    ! pasteboard x offset
      DATA   paste_y_offset/0/    ! pasteboard y offset
!
! Create a pasteboard.
!
      x_size = 438
      y_size = 824
      status = VSTA$CREATE_PASTEBOARD
     1      (x_size, y_size, pasteboard_id)
      IF ( .NOT. status ) CALL show_error(status)
!
! Create window into pasteboard (viewport is attached automatically).
!
      status = VSTA$CREATE_WINDOW
     1      ( x_size, y_size, pasteboard_id, paste_x_offset,
     2      paste_y_offset, window_id,
     3      'VSTA$COPY_AREA Map Functions' )
      IF ( .NOT. status ) CALL show_error(status)
!
! Set size of the vds that will display the mapping functions.
!
      vd_x_size = 99
      vd_y_size = 798
```

```
!
! For each virtual display that is going to display mapping:
!
      DO i = 1, 2
!
!   1)  calculate its offsets
!
          vd_x_offset = (i-1) * 338
          vd_y_offset = 22
!
!   2)  create it
!
          status = VSTA$CREATE_DISPLAY
     1      (vd_x_size, vd_y_size, z_value, screen_color,
     2      text_color, n_lines, line_height, vd_ids(i) )
          IF ( .NOT. status ) CALL show_error(status)
!
!   3)  paste it to left or right side of pasteboard
!
          status = VSTA$PASTE_DISPLAY
     1      (vd_ids(i), pasteboard_id,
     2      vd_x_offset, vd_y_offset )
          IF ( .NOT. status ) CALL show_error(status)
      ENDDO
!
! Create a header vd.
!
!   1)  create it
!
      status = VSTA$CREATE_DISPLAY
     1      ( x_size, 20, z_value, screen_color,
     2      text_color, n_lines, line_height,
     3      vd_ids(3), , ,)
      IF ( .NOT. status ) CALL show_error(status)
!
!   2)  paste it to pasteboard
!
      status = VSTA$PASTE_DISPLAY
     1      (vd_ids(3), pasteboard_id,
     2      0, 0 )
      IF ( .NOT. status ) CALL show_error(status)
!
! Create a vd for center
!
!   1)  create it.
!
      status = VSTA$CREATE_DISPLAY
     1      ( 235, 800, z_value, screen_color,
     2      text_color, n_lines, line_height, vd_ids(4) )
      IF ( .NOT. status ) CALL show_error(status)
!
!   2)  paste it to pasteboard.
!
      status = VSTA$PASTE_DISPLAY
     1      (vd_ids(4), pasteboard_id, 101, 20 )
      IF ( .NOT. status ) CALL show_error(status)
```

```
!
! End of routine.
!
      RETURN
      END


!-------------------------------------------------------------------------
      INTEGER function xoff *4 (ind)
!-------------------------------------------------------------------------
      IMPLICIT NONE
      INTEGER*4     ind
      INTEGER*4     i,j,k

      xoff = 50 * (ind-1)/16

      RETURN
      END


!-------------------------------------------------------------------------
      INTEGER function yoff *4 (ind)
!-------------------------------------------------------------------------
      IMPLICIT NONE
      INTEGER*4     ind
      INTEGER*4     i,j,k

      yoff = 50 * ( mod ( (ind-1), 16 ) )
      RETURN
      END


!-------------------------------------------------------------------------
      SUBROUTINE show_error(status)
!-------------------------------------------------------------------------
!
! Function:
!
!     Show error block.
!
! Input:
!
!     status = status from a VSTA$ function call
!
!
      IMPLICIT NONE
!
! Input declarations.
!
      INTEGER*4     status
!
! VSTA$GET_ERROR_BLOCK declarations.
!
      INTEGER*4     s, VSTA$GET_ERROR_BLOCK
```

```
!
! SYS$GETMSG declarations.
!
      INTEGER*4     ilen
      CHARACTER*80  mesbuf
!
! Routine declarations
!
      CHARACTER*80  sp80
      DATA   sp80/' '/
!
! Call VSTA$GET_ERROR_BLOCK.
!
      s = 42
      ilen = 0
      s = VSTA$GET_ERROR_BLOCK()
!
! If unable to get error block, call SYS$GETMSG.
!
      IF (.NOT.s) THEN
          PRINT*,'could not even get error block'
          mesbuf = sp80           ! initialize message buffer
          CALL SYS$GETMSG(%VAL(status), ilen, mesbuf, %VAL(15), )
          PRINT '(z10)',status
          PRINT '(a80)',mesbuf
      ENDIF
!
! Exit.
!
      CALL exit
      RETURN
      END



!-----------------------------------------------------------------------
      SUBROUTINE draw_bar( vd_id, vd_x_size, vd_y_size, color,
     1                     offx, offy )
!-----------------------------------------------------------------------
!
! Function:
!
!      Draw a bar on the display whose channel number is vd_id
!          and whose dimensions are  vd_x_size  by  vd_y_size
!          in the specified color.
!
      IMPLICIT NONE
!
! Input declarations.
!
      INTEGER*4     vd_id, vd_x_size, vd_y_size, color, offx, offy
!
! Routine declarations.
!
      INTEGER*4     status, proj45,
     1              line_width, average_dimension,
     2              screen_middle_x, screen_middle_y, circle_radius
```

```
!
! Check input for bad x and y dimensions.
!
      IF ( ( vd_x_size .lt. 1 ) .or. ( vd_y_size .lt. 1 ) ) THEN
          PRINT*,' Bad vd dimensioning '
          STOP
      ENDIF
!
! Calculate location, radius, and line width of circle.
!
      screen_middle_x = vd_x_size / 2
      screen_middle_y = vd_y_size / 2
      average_dimension = ( vd_x_size + vd_y_size ) / 2
      circle_radius = .425 * float ( average_dimension )
      line_width = .05 * float ( average_dimension )
      IF ( circle_radius .LT. 1 ) circle_radius = 1
      IF ( line_width .LT. 1 ) line_width = 1
!
! Draw a circle centered at middle of display.
!
      CALL draw_circle ( vd_id, line_width, color, circle_radius,
      1                    screen_middle_x, screen_middle_y,
      2                    offx, offy )
!
! Draw a diameter of the circle at 45 degrees.
!
      proj45 =  float(circle_radius) / 1.414159
      CALL draw_line ( vd_id, line_width, color,
      1              ( screen_middle_x - proj45 ),
      2              ( screen_middle_y + proj45 ),
      3              ( screen_middle_x + proj45 ),
      4              ( screen_middle_y - proj45 ),
      5             offx, offy )
!
! End of routine.
!
      RETURN
      END



!-----------------------------------------------------------------------
      SUBROUTINE draw_line ( vd_id, width, color, x1, y1, x2, y2,
      1                    offx, offy )
!-----------------------------------------------------------------------
!
! Function:
!
!     Draw a line on the display whose channel number is  vd_id, fro
!         point (x1,y1) to point (x2,y2) with width and color
!         specified.
!
      IMPLICIT NONE
!
! Include the VAXstation native mode graphics symbol table.
!
      INCLUDE 'SYS$LIBRARY:VSTAGBL.FOR'
```

```
!
! Input declarations.
!
      INTEGER*4      vd_id, width, color, x1, y1, x2, y2, offx, offy
!
! Routine declarations.
!
      INTEGER*2      pathlist(6), mask(2), pattern_block(3),
     1               pattern_state(2)
      INTEGER*4      status, VSTA$DRAW_CURVE, masktype
      PARAMETER      flag_move = 2
      DATA   pattern_block / 3 * 0 /,
     1       pattern_state / 2 * 0 /
!
! Fill a pathlist for a line.
!
      pathlist(1) = x1
      pathlist(2) = y1
      pathlist(3) = flag_move
      pathlist(4) = x2
      pathlist(5) = y2
      pathlist(6) = 0
!
! Make the mask a square of dimension width to draw a wide line.
!
      mask(1) = width
      mask(2) = width
!
! Draw the line.
!
      status = VSTA$DRAW_CURVE(
     1             VSTA$K_SRC_CONST, color, 0, 0,
     2             VSTA$K_MSK_RECTANGLE, mask,
     3             VSTA$K_DST_VD, vd_id, offx, offy,
     4             VSTA$K_MAP_SRC, , ,
     5             2, pathlist,
     6             VSTA$K_PTN_SINGLE_SRC, VSTA$K_PTN_NO_UPDATE,
     7                    pattern_block, pattern_state,
     8              , , , , 0, , 1, 5)
      IF ( .NOT. status ) CALL show_error(status)
!
! End of routine.
!
      RETURN
      END
```

```
!-------------------------------------------------------------------
      SUBROUTINE wait
!-------------------------------------------------------------------
!
! Function:
!
!      to wait until user presses <RETURN>
!
      CHARACTER*10  more

      PRINT*,' more?'
      accept'(a)',more

      RETURN
      END
```

Glossary

Bitmap: A data structure consisting of a rectangular array of pixel values. The specification of a bitmap includes its starting address in memory and the size of the rectangular array of pixels that it represents, including height, width, and depth (that is, the number of bit planes or bits/pixel).

Clipping Rectangle: A rectangle used to constrain an operation on a set of pixels in a bitmap. The intersection of a clipping rectangle with a destination bitmap rectangle forms a restricting boundary on the operation being executed.

Display: For the purposes of this document, a physical device consisting of a high-resolution raster-scan monitor, keyboard, pointing device, control processor, microcode, and firmware. A memory in the display specifies the intensity or color for each dot on the monitor. Each dot is individually addressable from the host.

Firmware: A control program that is loaded into the display (see Microcode).

Font: A collection of logically related bitmaps, addressed by index, usually defining the symbols of a character set.

Frame Buffer: The bitmap memory used to store the current value of each pixel from which the physical display monitor is refreshed.

Halftone: A rectangular pattern used to tile a destination bitmap. That is, the halftone is replicated in the destination along its width and height to fill the destination. Halftones are used to supply levels of grey or texture on a system without shading, as in newspaper printing.

Microcode: A control program which is fixed in hardware (that is, not loadable), usually in ROMs or PROMs.

Pixel: A single picture element or addressable point on a display. Each pixel has a value, represented by one or more bits, that describes its state (that is, the intensity or color of that point).

Point: The address of a pixel in a two-dimensional Cartesian coordinate system. The coordinates are specified as 16-bit signed x and y components. Within a bitmap, pixels are addressed relative to the upper left corner.

Pointer: A small image that is superimposed on the screen to indicate the current position of the pointing device. The pointer is automatically moved by the display hardware to reflect pointing device movement. The pointer image does not interfere with the current state of the frame buffer.

Rectangle: A rectangular array of pixels within a bitmap. A rectangle is specified by the coordinates of its origin (upper left corner) within the bitmap and its extent (width and height).

Index

VAXstation
Native Graphics Procedures
AA-AG30A-TE

Reader's Comments

Note: This form is for document comments only. Digital will use comments submitted on this form at the company's discretion.

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

_____

_____

_____

_____

_____

_____

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

_____

_____

_____

_____

_____

_____

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

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

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Do Not Tear – Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Software Publications Group MR01-2/L12
DIGITAL EQUIPMENT CORPORATION
200 FOREST STREET
MARLBORO, MA 01752

Do Not Tear – Fold Here and Tape