COMPUTE!'s
Technical Reference Guide

ATARI ST

VOLUME ONE

Sheldon Leemon

A practical tutorial and reference to the Virtual Device Interface, the ST's graphics routines. Includes practical program examples written in C, machine language, and BASIC. For the intermediate-to-advanced-level Atari ST programmer.

COMPUTE!'s
Technical Reference Guide
ATARI ST
VOLUME ONE: VDI

Sheldon Leemon

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

Foreword

The Atari ST is a powerful personal computer. So powerful, in fact, that using it to best effect can be a difficult task—even if you have all the available Atari documentation. That's why you'll find *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI* so valuable. Clear and concise, with program examples at every turn, it's the most complete guide to programming graphics on the Atari ST.

Filled with programs written in C, machine language, and BASIC, this reference guide and tutorial covers everything you need to access program the advanced graphics capabilities of the ST's *Virtual Device Interface*, or *VDI*.
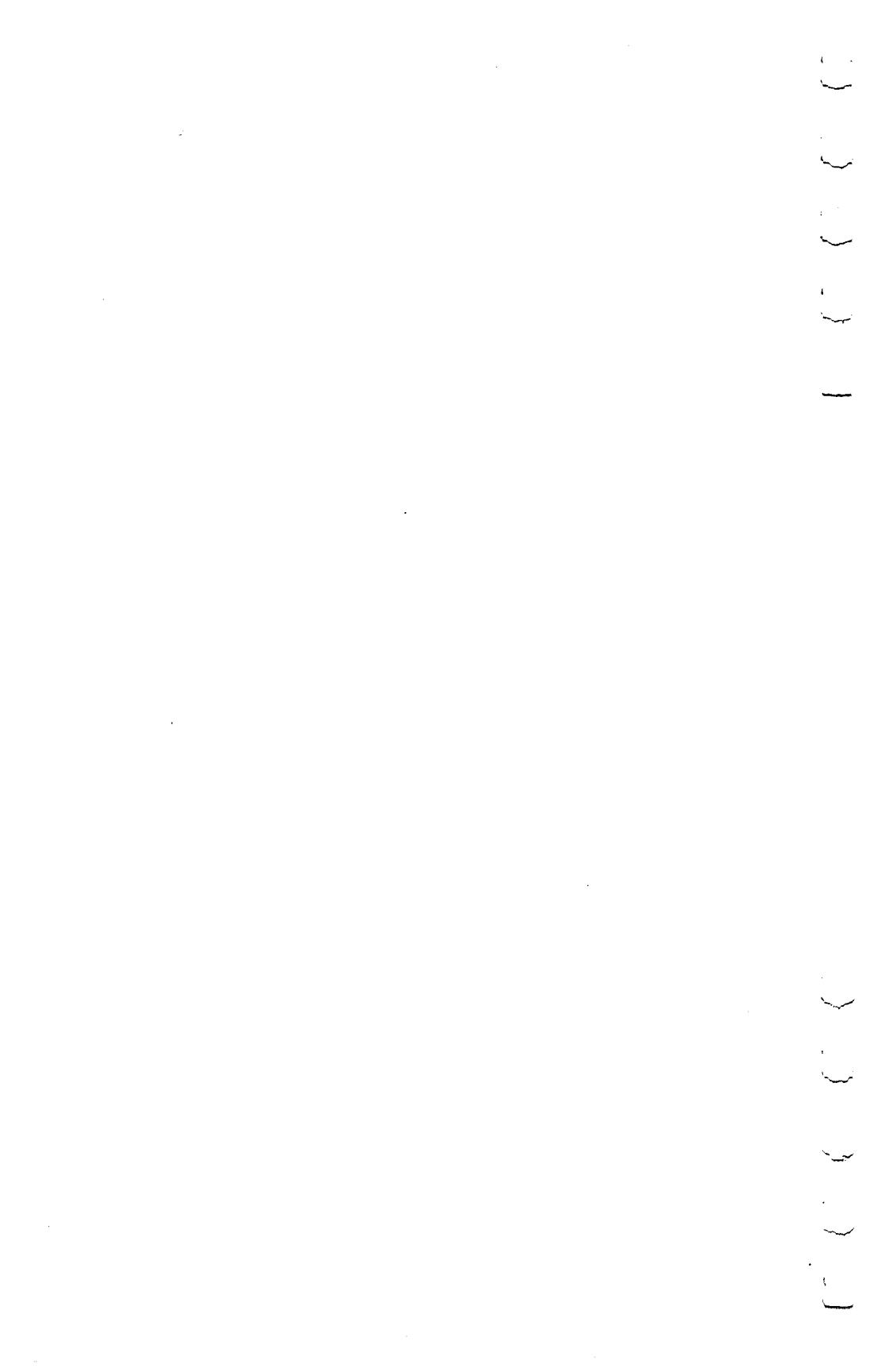
The first sections explain—in plain English, not jargon-filled computerspeak—VDI and GEM and how to set up a graphics environment using VDI functions. Later chapters illustrate how to use VDI functions to draw points and lines, fill areas, and move shapes around the screen.

Program after program shows you how to get your own ST creations to do what you want them to do. You'll see how to read the mouse pointer and other input devices with VDI functions.

COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI devotes an entire chapter to *text* on the ST—after all, text on this computer is just another form of graphics. Demonstration programs show how to align and rotate graphic text strings and discussions on how letters are formed.

The latter half of this book is a complete reference to VDI functions. You will find everything you need to know about each function in one place—a summary of each function, its opcode, C binding, and more. We've even included two indices to the VDI functions, so finding the right function is easier and faster.

COMPUTE! Books is the leading publisher of programs and information for the Atari ST. *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI* is yet another example of the high quality you've come to expect in any guide to personal computing from COMPUTE!.

Chapter 1

VDI and the GEM Graphics Environment

When we think of the Graphics Environment Manager (GEM) operating system, the first thing that comes to mind is the mouse-driven user interface, with its drop-down menus and icons. But there is another side to GEM which is of considerable interest to programmers. This part of GEM is known as the *Virtual Device Interface, or VDI.*

The function of the VDI, as its name suggests, is to provide a uniform, device-independent graphics interface that allows a programmer to design graphics output for a program without necessarily knowing the operational details about the computer on which the program is run, or about the hardware device (screen, printer, plotter, and so on) used to produce the output. This interface is based on previous graphics software interfaces and on the work of a computer industry standards committee. If you keep in mind that it was written as an attempt to be a generalized standard for all kinds of computers, and was not written specifically to support the graphics capabilities of the ST, you may better understand the reasoning behind its implementation.

The VDI implements this device-independent interface in two ways. First, it supplies a wide array of basic graphics functions. These functions include drawing primitives (the fundamental commands used to draw line figures and filled shapes); attribute settings that control aspects of the figures such as the color, size, and shape; and inquiry commands that enable the program to determine specific information about the graphics environment. They even include input functions that enable the programmer to accept input from the user via the mouse pointer, alphanumeric keys, cursor keys, and function keys.

The VDI also provides the means by which device-specific driver programs may be added to the system. These device-driver programs act as translators. The VDI routes the generalized output commands to the device driver, and the device driver converts these commands into the hardware-specific codes used to create the appropriate output on that particular device.

On the Atari ST computers, the part of the VDI that implements the basic graphic functions on the display screen is included in the *Tramiel Operating System (TOS) ROMs*. The part of the VDI that enables the use of disk-loaded fonts and device drivers, however, is not included as part of the current TOS ROMs and must itself be loaded from disk before these functions can be accessed. This part, known as the *GDOS* (Graphics Device Operating System), is contained in a file called GDOS.PRG, which must be included in the AUTO folder on the system disk used to start the computer if device drivers or software-loaded fonts are to be used. In addition, that disk should contain a text file called ASSIGN.SYS, which provides information about the location of the various device-driver and text-font files that are available.

This books deals with only the VDI portion of GEM, but the reader should be cautioned that the VDI does not operate in isolation from the *AES*, the *Application Environment Services* which form the other half of GEM. Unless you take the appropriate precautions, for example, the graphics functions presented here are quite capable of writing over the menu bar and window borders that are managed by the AES. Also, there is a certain amount of overlap between the two, particularly in the area of the VDI input functions. In a program where you use the AES input functions, you should be careful not to mix in VDI calls that will confuse them.

Using the VDI

You can think of the VDI as a collection of subroutines that you call from your program. In order to pass data to these subroutines and receive data from them, you must allocate storage space in memory for a number of data arrays. The VDI uses information from five different arrays, each of which is made up of a number of 16-bit (two-byte) values. These arrays are:

Array name	Size	Function
contrl	12 words	Control parameters
intin	0–256 words	Input parameters
ptsin	0–256 words	Input coordinates
intout	0–256 words	Output parameters
ptsout	0–256 words	Output coordinates

The array contrl consists of 12 elements, each two bytes in length. The information stored in each of the first seven of these elements is as follows:

Address	Element	Control parameter
contrl	contrl(0)	Command Opcode (operation code)
contrl+2	contrl(1)	Number of coordinate points in ptsin array
contrl+4	contrl(2)	Number of coordinate points in ptsout array
contrl+6	contrl(3)	Number of input parameters in intin array
contrl+8	contrl(4)	Number of output parameters in intout array
contrl+10	contrl(5)	Sub-function ID number
contrl+12	contrl(6)	Device handle (identification number)

The first element of the contrl array is used to pass the opcode. Since all of the VDI routines have a common entry point, there has to be some way to let the VDI know what command you want executed. Therefore, each command is given an identification number called an opcode. A few commands are further broken down into several sub-functions. In order to specify which of these sub-functions you wish to use, a sub-function ID number can be passed in contrl(5). Since the VDI can send output to several devices, you must also identify the device you wish to use by placing its handle in contrl(6). The handle is a device identification number which the system assigns when you successfully open the device. (Opening a graphics device like the screen for output will be covered later.)

The remaining four elements are used to indicate what portion of the other parameter arrays are used by a particular call. Of these, two are set aside for the input arrays ptsin and intin, in which you pass information to the function, and two are used for ptsout and intout, in which the function passes information back to you.

The reason you must specify the size of these arrays is that the number of values passed varies from function to function and can even vary from different calls to the same function. The line drawing command, for example, can draw lines between a number of points at once. Therefore, in order to communicate how many lines are to be drawn, you must specify the number of coordinate pairs that you have placed in the ptsin array before calling that command. The number of points is placed in contrl(1). This number is equal to half the length of the array, since each point must be described by both a horizontal and a vertical coordinate. In a similar fashion, the number of points passed back in the ptsout array from the

VDI command itself is stored in contrl(2). Contrl(3) and Contrl(4) are used to store the length of the intin and intout arrays, respectively.

Elements contrl(7)–contrl(11) aren't used for every command, but, when they are, they pass information that is specific to the command.

Assembly Language VDI Calls

If you're programming at the machine language level, you must explicitly reserve memory space for each of these arrays, and put the proper values in each of the memory locations before calling the command. The first step is reserving space for each of the data arrays:

```
contrl:   .ds.w  12
intin:    .ds.w  128
ptsin:    .ds.w  128
intout:   .ds.w  128
ptsout:   .ds.w  128
```

Since all of the arrays but contrl use a variable number of elements, depending on the particular call, it's best to allocate 128 words to each of them, which should be sufficient for most purposes. If you find that you need more elements, you may, of course, allocate additional space.

In addition to allocating data-array space, you must also define a VDI parameter block. This parameter block contains the beginning address of each of the five data arrays:

```
vpb:   .dc.l   contrl,intin,ptsin,intout,ptsout
```

Next, you must place any input parameters into their correct place in the data arrays. For example, to execute the Clear Workstation command that clears the screen, you would transfer the following values:

```
move #3, contrl       ;Move the Clear Workstation
                      ;opcode (3) to contrl(0).
move #0, contrl+2     ;Move the length of ptsin
                      ;array (0) to contrl(1).
move #0, contrl+6     ;Move the length of intin
                      ;array (0) to contrl(3).
move gh, contrl+12    ;Move the graphics handle
                      ;to contrl(6).
```

Now you're ready to call the VDI. First, place the address of the VDI parameter block into register d1. Next, move the VDI identifier code (115 or $78) into register d0. Finally, call the VDI with a trap 2 instruction. This initiates a software-generated exception (similar to a hardware interrupt) that causes execution of an exception-handler routine. In this case, the routine executed is the one whose address is pointed to by the long word beginning at location 136 ($88). This routine is the one used to handle all GEM VDI and AES calls. (AES calls are identified by placing a value of 200 or $C8 into register d0.) The sequence for making a VDI call looks like this:

```
move.l    #vpb,d1    ;Move address of VDI
                     ;parameter block to d1.
moveq.l   #73, d0    ;Move VDI identifier
                     ;($73) into d0
trap #2              ;Call GEM entry point.
```

Please note that the procedures outlined above just cover the steps required to make the VDI call itself. Before you get to that stage, you must take some preparatory steps to set up both the program environment (for example, allocating stack space) and the graphics environment (for example, opening a GEM output workstation). These steps will be outlined in the next chapter and illustrated in an example program.

ST BASIC VDI Calls

The fundamental strategy for making VDI calls from ST BASIC is similar to that used when making such calls from assembly language programs, with the exception that BASIC takes care of much of the preparatory work.

Since the BASIC interpreter program must use VDI calls, it already has set aside memory for the data arrays contrl, ptsin, intin, ptsout, and intout. BASIC assigns the starting address of each of these arrays to a reserved variable of the same name. Thus, the starting address of the contrl array is found in the variable named contrl, the starting address of ptsin in the variable ptsin, and so on. By using the PEEK and POKE commands, you may access the various elements of these data arrays. Remember that each element is two bytes long, so you must multiply the element number by two to get the proper offset for the POKE statement. The following short program shows how to clear the screen with the the Clear Workstation

call from BASIC. If you type it in and run it, you'll see that Clear Workstation erases everything on the screen, including window borders and the menu title bar.

```
10 REM POKE contrl(0) with Clear Workstation opcode (3)
15 POKE contrl,3

20 REM POKE contrl(1) with length of ptsin array (0)
25 POKE contrl+2,0

30 REM POKE contrl(3) with length of intin array (0)
35 POKE contrl+6,0

40 VDISYS(1)
```

Although this program is similar to the assembly language version shown above, you'll notice a couple of differences. First, we didn't have to POKE a value for the graphics handle into contrl(6) (contrl+12). That's because we're using the same display device as BASIC, and BASIC has already put the graphics handle there for us. The second difference is the use of the VDISYS call. The BASIC statement VDISYS(1)—the one being a dummy value that could be any number—performs the same tasks as the three lines of assembly code that place the address of the parameter block into register d1 and the VDI identifier code into proper values in the d0 register, and then execute the TRAP #2 statement.

The original version of ST BASIC contains several built-in commands that perform the same functions as VDI calls without the hassle of POKEs. Although not released at the time of this writing, the revised MCC BASIC promises to include even more graphics commands. Nevertheless, BASIC programmers can still benefit from learning about the VDI. A familiarity with the structure and function of the VDI calls gives a better understanding of how the BASIC graphics commands work and how they interact. Even the enhanced version of BASIC does not include keywords for all of the VDI functions. Learning how to access VDI calls directly from ST BASIC provides the means for using all of the tools provided by the GEM VDI, not just those that have been implemented by BASIC.

Calling the VDI Routines from C
It's much easier to make VDI function calls from C than from either assembly language or BASIC. That's because C compiler packages for the ST include one or more function libraries

known as GEM bindings. These bindings are object-code library files that define a separate, named function for each VDI call. When the C program is linked to the proper library files, it can call VDI functions as if they were part of the C language.

You still must allocate storage space for the data arrays, by making the following global array declarations at the beginning of this program:

```
int contrl[12],
   intin[128],
   ptsin[128],
   intout[128],
   ptsout[128];
```

But you're not responsible for placing data directly into these arrays. Instead, input parameters are passed to the binding functions as part of the function call. For example, you could execute the Clear Workstation call from C with the following call:

```
v_clrwk(handle);
```

The function defined as v_clrwk in the library takes the parameter handle that is passed to it and puts it in contrl(6). It also puts a zero in contrl(1) and (3), and places the command opcode (3) in control(0). It then loads registers d0 and d1 with the proper values, and executes a TRAP #2 instruction. In short, it takes over all of the repetitive steps associated with making VDI calls, allowing the programmer to concentrate on the essential aspects of the function.

Because it's easy to make GEM calls from C, and because the language produces programs that are relatively small in size and quick in execution for a high-level language, it has become the language of choice for software development on the ST. Therefore, most of the examples in this book will be written in C. On occasion, however, we will include assembly language and BASIC examples as well, to show how the C examples may be translated to these other environments. We will use the C function names as they appear in the official Digital Research GEM bindings, since they have been adopted by the manufacturers of other C compilers as well.

The C programs in this book are designed to work specifically with the Alcyon C compiler (the one officially supported by Atari) and with Megamax C, which also provides a very

complete development environment. For these compilers, the int data type refers to a 16-bit word of data. Some other compilers, such as the Lattice C compiler, use a 32-bit integer as the default data type. You should substitute "short" for each reference to "int" when compiling the programs in this book with such compilers. For the sake of simplicity, we have not used the portability macros such as WORD, which use the C preprocessor to define a 16-bit data type that will be valid for any compiler, but you're free to do so if you find it more convenient.

Chapter 2

Setting Up the Graphics Environment

Before you can begin using VDI calls, you must
first prepare a graphics output environment by opening a
GEM workstation. When you open a workstation, the GDOS
loads a device driver file (if necessary), initializes the output
device, reserves environment space for storing graphics
settings associated with that workstation, and returns a device
identification number, or handle, which is used to identify the
output device when making VDI calls. For graphics output de-
vices *other than the display screen,* the call to use is Open
Workstation, which opens a physical device workstation. The
C format for this call is

```
int input[12];
int output[57]
int handle;

v_openwk(input, &handle, output)
```

The array, input, consists of twelve words of data that
you pass to the VDI. The first of these, the Device ID number,
is used to let the GDOS know what device driver file it must
load from disk. As explained in the previous chapter, the
ROM part of the VDI doesn't know how to talk to any graph-
ics output device except the computer's own display screen. In
order to communicate with any other device, the GDOS must
first load a device driver file to translate its graphics com-
mands into a format that the output device recognizes.

Loading the device driver is the job of the Open Work-
station command, but in order for it to successfully perform
that task, several conditions must be satisfied. First, the GDOS
extensions must have been loaded into computer memory by
running the GDOS.PRG program. This should be done by in-
cluding that file in the AUTO folder on the boot disk. At the
time the GDOS.PRG program is run, there must be a file
called ASSIGN.SYS in the root directory of that disk. This is a
text file that tells the GDOS what drivers' IDs are available,
what the driver files are called, and what text-font files are

available for that graphics device. The format for each entry in the assign.sys file is

ID_number(flag) filename
(fontfilename)

where *ID_number* is the device ID number, *flag* is an optional letter that can be added to the ID number to give special loading instructions for the driver, *filename* is the name of the actual file containing the device driver, and *fontfilename* is the name of an optional font file that the device can load with the vst_load_font call. If there is more than one font file available for the device, additional font files may be listed below the ID line, one filename per line.

Although technically you're free to assign any ID number from 0 to 32767 to any device, it's recommended that you use numbers in the following ranges for these devices:

Device	ID Range
Display Screen	01–10
Plotter	11–20
Printer	21–30
Metafile	31–40
Camera	41–50
Tablet	51–60

The two flags that can be added to the device number are the letters *r* and *p*. An *r* indicates that the driver is resident, which means that it's loaded at the time the GDOS is booted and stays in RAM until the computer is reset. A *p* signifies that the driver is maintained permanently in ROM, like the screen driver that is contained in the Operating System ROM. If neither of the flags follows the ID number, it is assumed that the device driver file will be loaded when the physical workstation is opened. A typical entry in the assign.sys file looks like this:

21 fx80.sys
EPSHSS14.FNT

This describes a device driver file for the Epson FX-80 printer. The printer ID number is 21, the driver itself is contained in a file called fx80.sys, and there is one text font available for use by this driver, in a file called epshss14.fnt (a single-height, single-width font 14 points high).

When opening a workstation for this device, the device

number would be given as the first input parameter (intin(0)). The rest of the input array parameters are used to specify the initial default graphics settings for the workstation, and will be discussed more fully later on.

The other two parameters associated with the Open Workstation call, *handle* and *output*, are used by the function to return information about the workstation that was just opened. The most important item is the workstation ID number, or handle, that is returned in the variable handle. This ID must be included as part of the input to all of the other VDI functions, to indicate the device to which graphics output is to be sent. If the VDI can't open the device, it returns a handle number of zero. Up to 16 workstations may be open at one time, and it is possible to open more than one physical workstation for devices other than the display screen.

In addition to supplying the handle, v_opnwk fills the array output with 57 items of information about the workstation that was just opened. All of the input and output parameters will be covered in the discussion of virtual workstations, below.

Virtual Workstations

The Open Workstation call is used for all devices except the display screen. The display device has a unique role in the GEM system. It's the primary means of communicating with the user, so it's the one graphics device that has to be open almost all the time.

The display device is the only device for which there is a device driver built into the TOS ROMs. And since it has to be shared by the Desktop, application programs, and desk accessories, it's the only graphics device that must display information from more than one program at a time.

In order to allow multiple users to access a single display, GEM uses *pseudo-devices* called *virtual screen workstations*. Once a physical workstation has been opened, the physical device can be subdivided into one or more virtual workstations, each of which has complete access to the display screen. Each virtual workstation maintains its own set of graphics settings, so that if one application makes a change that affects how its drawing takes place, that change will not affect the other workstations as well. In fact, a single application can open more than one virtual screen workstation at a time.

The format of the Open Virtual Screen Workstation function is identical to that of the Open Workstation function:

```
int input[12];
int output[57]
int handle;
```

v_openvwk(input, &handle, output)

Input array. The input array consists of twelve words of data that you pass to the VDI to specify the initial default graphics settings for the workstation. It should be noted that unless you load the GDOS extensions with the GDOS.PRG program, or have a version of the TOS ROMs with the GDOS built in, none of these input values will affect the initial workstation settings, which will always be set to their default values. We'll only briefly mention these settings here, since most duplicate the function of individual graphics setting commands that will be discussed at length in the chapters on line drawing, filled shapes, and color. They are

Element	Contents	Comparable VDI function
input[0]	Device ID number	
input[1]	Line drawing pattern	vsl_type
input[2]	Line pen number	vsl_color
input[3]	Marker type	vsm_type
input[4]	Marker pen number	vsl_color
input[5]	Text font	vst_font
input[6]	Text pen number	vst_color
input[7]	Fill pattern type	vsf_interior
input[8]	Fill pattern index	vsf_style
input[9]	Fill pen number	vsf_color
input[10]	NDC to RC transformation flag	

The first and last of these parameters require a little further explanation. The first, Device ID number, is used somewhat differently for virtual screen workstations. That's because on the ST there isn't one standard type of screen display device that's always used. Instead, the user is allowed to choose from three different types of displays. The monochrome screen offers a resolution of 640 X 400 pixels, but only two colors (black and white). The color monitor uses either a medium resolution mode of 640 X 200 pixels, with a maximum of 4 colors on screen at once, or a low-resolution mode of 320 X 200 pixels,

with a maximum of 16 colors. In either color mode, each color may be selected from any of the 512 available.

While your program cannot dictate which of the three displays is to be used, it should be able to load a set of text fonts that is appropriate for the current display. Therefore, in the assign.sys file, there are several IDs assigned to the screen device. These assignments, which are specific to the Atari version of GEM only, are

Device Number	Screen Type
01	Default
02	Lo-res color
03	Medium-res color
04	Hi-res monochrome
05–10	Reserved for Atari expansion

Therefore, a typical assign.sys file is shown in Program 2-1.

Program 2-1. Typical assign.sys

```
path = c: \ drivers  ;Optional path designation for
                     ;device driver and font files
;
01p screen.sys       ;The default setting, provided
                     ;for compatibility with pre-GDOS
                     ;applications
;
02p screen.sys       ;Using device ID 2 makes available
LOWRES10.FNT         ;these lo-res fonts only
LOWRES14.FNT
LOWRES18.FNT
;
03p screen.sys       ;Using device ID 3 makes available
MEDRES10.FNT         ;these medium-res fonts only
MEDRES14.FNT
MEDRES18.FNT
;
04p screen.sys       ;Using device ID 4 makes available
HIRES10.FNT          ;these hi-res fonts only
HIRES14.FNT
HIRES18.FNT
21 fx80.sys          ;Epson printer driver and fonts
EPSHSS10.FNT
EPSHSS20.FNT
EPSHSS28.FNT
;
31 meta.sys          ;Meta-file driver
```

17

As you can see, an optional path statement may be used to designate a path name for the device drive and text-font files. This path specification must be given at the beginning of the file, before any device ID assignments are made. The path-name can only be 64 characters long. It doesn't matter whether the names are entered in upper- or lowercase letters. Next come the device IDs for the screen drivers. A filename of screen.sys is given for each entry so it will conform to the standard format, but the GDOS doesn't try to read in a device driver file of that name because the p flag after the device number tells it that this driver is permanently installed in ROM. A device number of 1 is used to specify the default screen driver. This indicates that you don't care about match-ing the disk-based text fonts to the screen resolution. Device numbers 2, 3, and 4 are used for the low, medium, and high-resolution screens, respectively. In order to open the virtual workstation with the proper ID, however, you first must deter-mine what display is in use. You can use the XBIOS (Extended Basic Input/Output System) command 4 to determine the cur-rent screen resolution. From C, you can use getrez, a macro defined in the file "osbinds.h", to call this function. To find the proper ID number, use the statement

ID = getrez( ) + 2;

Since getrez returns the number 0 for lo-res, 1 for medium-res, and 2 for hi-res, all you have to do is add 2 to the value returned to get the right ID number. Assembly language pro-grammers can perform the getrez call using the following code:

```
move.w    #4,-(sp)    * push command number on the stack
trap      #14         * call XBIOS
addq.l    #2,sp       * pop command number off the stack
```

The resolution will be returned in register d0.

The last Open Virtual Workstation input parameter, the Normalized Device Coordinate (NDC) to Raster Coordinate (RC) transformation flag, allows you to specify which coordi-nate system you'll use for drawing. The RC system is the one used most commonly on microcomputers. Under this system, the screen is divided into rows and columns of dots, which represent every point that can be plotted on the display screen. The dots in the top row have a vertical, or y, coordi-nate of zero.The vertical coordinate number increases as you

move toward the bottom row of the screen, which has a y co-ordinate of 199 on the color screen and 399 on the mono-chrome display. Likewise, the leftmost column has a horizontal or x coordinate of zero, which increases as you move toward the rightmost column, where the x coordinate is 639 on medium- or high-resolution displays, and/or 319 on low-resolution displays. (See Figure 2-1.)

But the GDOS also supports another system known as Normalized Device Coordinates (NDC). Since almost every graphics output device has a different maximum horizontal and vertical resolution, it's difficult to write a single program that will work with many different types of devices. That's where the NDC system comes in. It attempts to offer the pro-grammer a system in which graphics drawn on one computer screen or printer will look the same when drawn on other computers screens or printers of varying resolutions. When you use NDC coordinates, you send all of your graphics out-put to an imaginary display that is 32,768 pixels wide by 32,768 pixels high. These pixels are grouped differently than they are under the Raster Coordinate system, since the vertical axis starts at the bottom of the screen (0) and moves up to the top row (32767). As in the RC system, the left-hand column is zero, and x coordinate numbers increase as you move the rightmost column (32767). (See Figure 2-2.)

Figure 2-1.
Raster Coordinate (RC) System
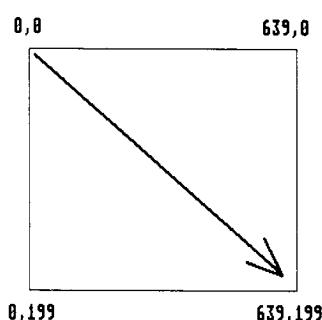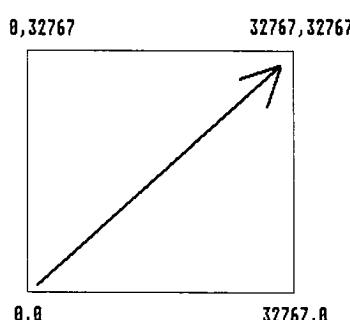
Figure 2-2.
Normalized Device Coordinate (NDC) System

0,0   639,0

0,199   639,199

0,32767   32767,32767

0,0   32767,0

19

The GDOS takes the graphics output that you send to this enormous display and scales it down in proportion to the more modest dimensions of your actual output device. For example, let's say that you order the VDI to draw a box whose upper left corner is at point 8192,24576 and whose lower right corner is at point 24576,8192 in the NDC system. This box follows the outline of the display, a quarter of the way in from each edge. If you're using the ST's medium-res color screen for your display, GDOS transforms the normalized coordinates to raster coordinates, and draws a box from 160,50 to 480,150. But if you're using the lo-res screen, the box would go from 80,50 to 240,150, and in hi-res it would go from 160,100 to 480,300. This means that with one command, you can create the same size box on each of the ST's three display screens.

You may find, however, that normalized coordinates may not be as useful as they may seem at first. For one thing, they slow down all of the graphics operations. Even though the ST computers are fast, they still aren't so fast that the extra step of translating normalized coordinates to raster coordinates can't cause some appreciable delay in complex drawing operations. Secondly, the disparity between the resolution of the normalized display and that of real-world graphics devices is so great that something is bound to be lost in the translation. With only 1 RC pixel for every 8000 NDC pixels, there is no way that very complex drawings can be accurately reproduced on screens of varying resolutions. Realistically, most ST programmers will want to write applications specifically for the ST and will want to know exactly where every pixel will be drawn. So while NDCs are a nice idea, the RC system will probably still be the one used most often, at least until that computer with the 32,768 × 32,768-pixel display comes along.

The handle. Handles pass a lot of information when they are called. The most important item is the workstation ID number, or handle, that is returned in the variable handle. This ID is included as part of the input to all of the other VDI functions, to indicate the device to which the graphics output is to be sent. For v_opnvwk only, *handle* is used as both an input and an output parameter. In order to get back the new handle for your virtual workstation, you should first pass the handle of the screen device that GEM has already opened.

You can get this handle by using the call

**int handle,
    charw, charh,
    boxw, boxh;**

handle = graf_handle(&charw, &charh, &boxw, &boxh);

Again, note that unless you load the GDOS extension by running the GDOS.PRG program, this step really doesn't accomplish anything, since it appears that the input value placed in handle is ignored by the TOS ROMs. But even if you aren't running GDOS.PRG from the AUTO folder of your boot disk, it pays to follow the GEM guidelines, since, otherwise, your program will not work on computers that do have the GDOS resident. Besides, this call returns some interesting information about the size of the default text font. The width and height of the actual characters are returned in charw and charh, while the width and height of the text box, the cell in which each character is placed, are returned in boxw and boxh. We'll go deeper into the formation of text characters in the chapter that deals with text.

Output array. In addition to supplying a handle for your new workstation, the v_opnwk and v_opnvwk calls fill the array output with 57 varieties of information about the workstation that was just opened. This information is really provided to aid in creating more portable GEM programs that may be run on other types of computers and on all kinds of output devices, but it is still of interest to the programmer who works exclusively with the ST screen device. Information given about the workstation includes:

output[0]	Maximum horizontal coordinate value (in points or pixels)
output[1]	Maximum vertical coordinate value (in points or pixels)
output[2]	Device Coordinate units flag (1 = device doesn't support precise scaling)
output[3]	Width of one pixel in microns (1/1000 millimeter) For display screens, horizontal component of aspect ratio
output[4]	Height of one pixel in microns (1/1000 millimeter) For display screens, vertical component of aspect ratio
output[5]	Number of text font heights (0 = continuous scaling)
output[6]	Number of line patterns
output[7]	Number of line widths (0 = continuous scaling)

output[8]   Number of marker patterns
output[9]   Number of marker sizes
             (0 = continuous scaling)
output[10]   Number of text fonts supported by the device
output[11]   Number of pattern fill styles
output[12]   Number of crosshatch fill styles
output[13]   Number of drawing-pen colors available
             (the number of colors that can be displayed by the de-
             vice at the same time)
output[14]   Number of Generalized Drawing Primitives (GDPs)
             (how many of the 10 basic drawing commands are
             supported)
output[15]   This part of the array holds a sequential list of code
     to   numbers for the first 10 GDPs supported.
output[24]   Each element holds one of the following code numbers:
                  1 = Filled Rectangle or Bar (v_bar)
                  2 = Circle Segment or Arc (v_arc)
                  3 = Filled Pie Slice (v_pieslice)
                  4 = Filled Circle (v_circle)
                  5 = Filled Ellipse (v_ellipse)
                  6 = Elliptical Arc (v_ellarc)
                  7 = Filled Elliptical Pie Slice (v_ellpie)
                  8 = Rounded Rectangle (v_rbox)
                  9 = Filled Rounded Rectangle (v_rfbox)
                 10 = Justified Graphics Text (v_justified)
                 −1 = End of list
output[25]   This part of the array holds a sequential list of code
     to   numbers showing what category of graphics operation is
output[34]   performed by of each of the supported GDPs. This indi-
             cates what kind of graphics settings affects each of the
             supported commands. Each element holds one of the fol-
             lowing code numbers:
                  0 = Line drawing
                  1 = Marker drawing
                  2 = Graphics text
                  3 = Filled area
                  4 = no setting
output[35]   Color availability flag
                  0 = device is not capable of color output
                  1 = device is capable of color output
output[36]   Text rotation availability flag
                  0 = device is not capable of text rotation
                  1 = device is capable of text rotation
output[37]   Area fill availability flag
                  0 = device is not capable of area fill operations
                  1 = device is capable of area fill operations

output[38] Cell array function availability flag
         0 = device can not perform the cell array function
         1 = device can perform the cell array function
output[39] Total number of color choices available in the palette
         0 = more than 32,767 colors available
         1 = monochrome
         2–32767 = actual number of colors available
output[40] Input devices available for the locator function
         1 = keyboard only
         2 = keyboard and other device (such as mouse)
output[41] Input devices available for the valuator function
         1 = keyboard
         2 = other device
output[42] Input devices available for the choice function
         1 = function keys on keyboard
         2 = some other keypad
output[43] Input devices available for the string input function
         1 = keyboard
output[44] Workstation type
         0 = output only
         1 = input only
         2 = input and output
         3 = reserved for future use
         4 = metafile output

output[45] Minimum character width
output[46] Minimum character height
output[47] Maximum character width
output[48] Maximum character height
output[49] Minimum line width
output[50] 0
output[51] Maximum line width
output[52] 0
output[53] Minimum marker width
output[54] Minimum marker height
output[55] Maximum marker width
output[56] Maximum marker height

The first two elements of the array give the maximum horizontal and vertical coordinates, assuming that the coordinates start at point 0,0. For the ST monochrome screen the horizontal value is 639 and the vertical value is 399, indicating a resolution of 640 × 400 pixels. The horizontal value for the color medium-res screen is 639, and for the lo-res screen it's 319. The vertical value for both color screens is 199. Element 2

contains a flag indicating whether the device is capable of pre-
cise scaling or only approximate values like a film recorder.
This flag is set to 0, indicating precise scaling, for all ST
screens.

The next two elements contain the width and height of
one pixel in microns (a micron equals 1/1000 millimeter).
While such measurements are more accurate for printers and
plotters than display screens which vary considerably from
unit to unit and model to model, they can be used to deter-
mine the aspect ratio, which is the ratio of the width to the
height. For the hi-res screen the pixel width value is 372. The
medium-res width is 169, while the lo-res width is twice that,
or 338. The pixel height is 372 in all three modes. Thus, while
the hi-res pixels are square, and the lo-res ones almost so, the
medium-res pixels are less than half as wide as they are tall.
The aspect ratio comes in handy when you try to draw boxes
that look square and circles that look round on the screen.
Since each pixel in medium-res is tall and skinny, a box that is
100 pixels wide by 100 pixels high will appear to be tall and
skinny as well. In order to get the box to look square, you
must multiply the width by the aspect ratio (work_out[3]/
work_out[4]) to get the height. In medium-res mode, a box
that is 100 pixels wide should only be 45 pixels high if it is to
appear square. The VDI does this kind of scaling for you auto-
matically when it draws a circle using one of the circle
functions.

The next 9 values and the last 12 give some information
about the kinds of graphics settings available. The VDI is ca-
pable of drawing text in a number of different sizes (or
heights, to be more precise), and element 5 tells exactly how
many are available (on the ST the default number is 3 in all
resolution modes).

Elements 45 and 46 give the minimum text character size
(5 pixels wide by 4 pixels high on the ST), and elements 47
and 48 give the maximum text character size (7 pixels wide by
13 pixels high).

Element 6 tells how many types of patterned lines can be
drawn. (On the ST this value is 7.) Lines can be drawn one
pixel wide or several pixels wide, and the number of available
line widths are stored in element 7. On the ST, this value is 0,
indicating that line widths may be continuously scaled from
the minimum value to the maximum (though even numbers

will be rounded down to the next lower odd value). Element 49 gives the minimum line width (1 pixel), while element 51 holds the maximum line width (40 pixels on the ST).

The next two values give the number of marker types and sizes. Markers are graphics objects that can look like a dot, a diamond, an asterisk, or other shapes, and the number of marker types indicates the number of shapes available. (On the ST, the standard set of six shapes can be used.) On the ST, these markers can be drawn in any of eight sizes. Elements 53 and 54 give the minimum marker size (15 pixels wide by 11 pixels tall), and elements 55 and 56 hold the maximum marker size (120 pixels wide by 88 pixels tall).

Element 10 shows the number of text fonts that are resident. On the ST, only one system font is available unless you load additional fonts from disk.

The next two values show the number of pattern types available for filling shapes with colored patterns. On the ST there are 24 pattern fill types and 12 crosshatch styles.

Finally, element 13 shows the number of drawing pens that are available. This number corresponds with the number of hardware color registers used for a particular mode and determines the maximum number of different colors that can be displayed on screen at one time. On the ST this value is 2 for the monochrome screen, 4 for the medium-res color screen, and 16 for the lo-res color screen. Since each of the ST's three display screens has a different maximum number of colors that it can display, the value found in element 13 can be used to determine what display mode is in use. Element 13 should not be confused with element 39, which holds the total number of colors available. For both of the color modes, this value is 512, which means that any of the drawing pens (color registers) can hold any of 512 possible values at one time. For the monochrome screen, this value is 2, since only black and white are displayed.

Next comes information about the types of basic drawing operations that may be performed. For historical reasons having to do with the older graphics systems from which the VDI evolved, these operations are known as Generalized Drawing Primitives (GDPs). Element 14 shows how many of the ten GDPs are supported. (On the ST, all ten are supported.) Elements 15–24 contain a list of the code numbers for the supported operations. Since the ST supports all ten, these elements

hold the numbers 1 through 10. Elements 25 to 34 contain a list of code numbers showing the type of drawing operation (line drawing, area fill, text, and so on) performed by each of the ten operations. The operation type of a graphics output function determines which group of graphics settings will affect it. On the ST, the values for these elements are

Element	Code	Function	Element	Code	Type
15	1	Filled Bar	25	3	Fill
16	2	Arc	26	0	Line
17	3	Filled Pie	27	3	Fill
18	4	Filled Circle	28	3	Fill
19	5	Filled Ellipse	29	3	Fill
20	6	Elliptical Arc	30	0	Line
21	7	Filled Elliptical Pie	31	3	Fill
22	8	Rounded Rectangle	32	0	Line
23	9	Filled Rounded Rectangle	33	3	Fill
24	10	Justified Text	34	2	Text

The next four values are flags that indicate whether the device supports a certain VDI function or not. Zero in one of these elements means that the device doesn't support the function, while 1 indicates that it does. The flag in element 35 indicates whether the device supports color output. This flag shows a 1 for either resolution of color display, and a 0 for the monochrome screen. The next flag shows whether text rotation is available. (It is on the ST, though we will see later on that text characters may only be turned in 90 degree increments.) The flag in element 37 shows whether the device supports area filling. (All ST screens do.) Finally, element 38 indicates whether the device supports the Cell Array function, in which a rectangular area of the drawing surface may be broken down into smaller rectangular color zones. The ST screens do not support this function.

In addition to all of the output functions, the VDI supports a number of input functions as well, which allow a program to receive feedback from the user. You can tell whether a particular device supports these functions by looking at the value in element 44, which tells whether it supports input and/or output functions. On the ST, the virtual screen workstation includes the keyboard and mouse as well as the screen, and therefore supports both input and output.

Elements 40–43 give information about the hardware devices used for the input functions. Element 40 specifies the input devices available for the locator function, which allows the

user to choose a point on the screen. On the ST, the value
here is 2, which indicates that either the keyboard or the
mouse may be used for this function. Element 41 shows the
input devices available for the valuator function, which allows
the user to move a value setting higher or lower within the
range of 1-100. On the ST, this value is 1, which means that
the keyboard (and more specifically, the arrow and shift keys)
are used for this function. Element 42 deals with the hardware
device used for the choice function, which lets the user select
from a number of options. The value here on the ST is 1,
which indicates that the ten function keys are used to return a
number from 1 to 10, indicating the choice that was selected.
Element 43 indicates what device is available for text character
string input. On the ST the value here is 1, indicating the key-
board (the only choice, actually) as the device used.

Extended Inquire
The VDI includes another function that can supply your pro-
gram with all of the same information that is returned by the
Open Workstation calls, plus a number of additional facts
about the graphics environment. This function is called *Ex-
tended Inquire,* and it uses the following format:

int handle, flag, output[57];

vq_extnd(handle, flag, output);

where *handle* is the workstation ID number, *output* is a pointer
to the array where the information is returned, and *flag* indi-
cates whether you want the function to return the same output
values as the Open Workstation calls (flag = 0), or the ex-
tended information (flag = 1). When you request the extended
information, the values returned in the output array have the
following meanings:

output [0]   Screen type
                0 = no screen
                1 = separate character and graphics controllers
                using separate screens
                2 = separate character and graphics controllers
                sharing a common screen
                3 = common character and graphics controller
                using separate graphics memory storage
                4 = common character and graphics controller
                using common graphics memory storage

output [1]   Number of available background colors
output [2]   Number of graphics text special effects
output [3]   Raster scaling flag
                        0 = scaling not supported
                        1 = scaling supported
output [4]   Number of color planes used for raster (number of bits per pixel)
output [5]   Color register lookup table flag
                        0 = lookup table not supported
                        1 = lookup table supported
output [6]   Number of 16 × 16 raster operations per second
output [7]   Contour fill flag
                        0 = contour fill not supported
                        1 = contour fill supported
output [8]   Text rotation flag
                        0 = text rotation not supported
                        1 = text may be rotated in 90 degree steps
                        2 = text may be rotated any angle
output [9]   Number of drawing modes
output [10]   Input modes flag
                        0 = no input modes supported
                        1 = request mode supported
                        2 = sample and request modes supported
output [11]   Text alignment flag
                        0 = text alignment not supported
                        1 = text alignment supported
output [12]   Inking flag (for plotters)
                        0 = device cannot ink
                        1 = device can ink
output [13]   Rubberbanding flag
                        0 = no rubberbanding
                        1 = rubberband lines only
                        2 = rubberband lines and rectangles
output [14]   Maximum vertices in ptsin (for Polyline, Polymarker, and Area Fill)
output [15]   Maximum size of intin
output [16]   Number of mouse buttons
output [17]   Is line pattern used for wide lines?
                        0 = no patterned drawing of wide lines
                        1 = wide lines can be drawn with patterns
output [18]   Drawing modes for wide lines
output[19]–[56]   Reserved for future use

On the ST, output [0] always shows the screen type to be 4, since graphics and text are bitmapped on the same screen in both color and monochrome modes. The number of available

background colors shown in output [1] is 1 for monochrome systems and 512 for color systems. The number of graphics text special effects reported in output [2] on the ST is 31. Output[3] shows that raster scaling is not supported. Output [4] shows that there is one color plane in monochrome mode, two color planes in medium-res mode, and four color planes in lo-res mode; this means that the monchrome screen can only display 2 colors at one time, while medium-res can display 4, and lo-res 16. Output [5] shows that a color-lookup table is not supported in monochome mode, but is supported in the color modes. (The color lookup table, which assigns VDI color index numbers to hardware color registers, is discussed in Chapter 3.)

The performance factor in output [6] shows that 1000 16 × 16 pixel raster operations can be performed in one second on the ST. Output [7] shows that the ST has flood-fill capability. Text characters may be rotated in 90 degree increments, according to the value in output [8]. There are four drawing modes available on the ST, as shown by output [9]. The GEM input pseudodevices work in both sample and request mode on the ST, per output [10]. Output [11] shows that text may be aligned. Output [12] shows that the screen device cannot ink, and output [13] shows that it cannot draw rubberband lines.

The maximum number of vertices for the Polyline, Polymark, or Area Fill functions is 128 on the ST, according to output [14]. (This means that the maximum size of the ptsin array should be 256.) There is no maximum size for the intin array, though, according to output [15]. Output [16] shows that the ST mouse has two buttons. Output [17] and output [18] show that wide lines cannot be drawn with line patterns and that they have no special writing modes.

Other Workstation Functions

The Clear Workstation function initializes the device to a state in which there is no graphics output. For the screen, this means setting every pixel to the background color. For a printer or plotter, the print buffer is cleared and a form feed is sent to advance to a new page. This function is performed automatically whenever a physical (but not a virtual) workstation is

opened. The format for the C version of this function is

v—clrwk(handle);

where *handle* is the graphics handle for the workstation.

Another function, Update Workstation, can be used with devices like printers, which don't execute graphics commands as soon as the program sends them, but accumulates data in a buffer first. This function is used to immediately execute any commands that are waiting in the buffer. It has no effect on a device like the screen, which always executes output commands immediately. The C command looks like this:

v—updwk(handle)

The final two workstation commands are Close Workstation and Close Virtual Screen Workstation. These de-allocate the workspace used to keep track of the device settings, and prevent further output to the device. You should always remember to close any devices that you have opened before exiting your program. The syntax for these functions is

v—clswk(handle);

and

v—clsvwk(handle);

A C Program Shell

Since most of the subsequent example programs use virtually the same program code to initialize the graphics environment, it would be repetitive to include the text of that code in every example. Instead, we include the steps necessary to open the screen device and get its identification handle below, in the form of a short program *shell*, Program 2-3. All Program 2-3 does is open the virtual workstation, call a function named demo, wait for somebody to press a mouse button (to give the viewer time to see the graphics display), and close the workstation. Since the function demo is not defined in this program, it will not link properly unless you add that function yourself. The way we'll do that in our examples is to use the C #include operator to include the file work.c at the beginning of most of our sample programs, and name the main function of the sample program demo( ). For example, to create a program that does absolutely nothing but wait for a mouse button

press, you could define an empty function for demo( ), as shown in Program 2-2.

Program 2-2. dummy.c

```
#include "work.c"
demo(){}
```

Keeping the initialization code in a reusable file will shorten our sample listings substantially, and will save you a lot of retyping. But be sure that the file work.c is stored where your compiler can find it, either in the same disk and directory as your standard header files or in the same disk and directory as the source code file.

Program 2-3. shell.c

```
/* Global variables -- For VDI bindings, etc. */

int contrl[12],
    intin[128],
    ptsin[128],
    intout[128],
    ptsout[128];

int handle;

int work_in[12],
    work_out[57];


/* Initialization starts here */

main()
{
    int x, nul, button=0;

/* Initialize the GEM application */

    appl_init();

/* Initialize input array, get the physical workstation handle,
   and open the Virtual Screen Workstation */

    for (x=0, work_in[10]=2; x<10; work_in[x++]=1);
    handle = graf_handle(&nul, &nul, &nul, &nul);
    v_opnvwk (work_in, &handle, work_out);
    v_clrwk(handle);

/* perform the graphics demos */

    demo();

/* Wait until the mouse button is pushed, then close the virtual
   workstation, and exit from the application */

    while(button==0)vq_mouse(handle,&button,&nul,&nul);
    v_clsvwk(handle);
    appl_exit();
}
```

Program 2-3 has a few function calls that haven't been discussed yet. For example, the first and last commands used are appl_init( ) and appl_exit( ). These are non-VDI GEM functions that are used at the beginning and end of every GEM program, the former to initialize the application and the latter to exit from it. Also, we used the vq_mouse function to check for a mouse button press. This is a VDI input function, and we will discuss it in more detail in Chapter 8. The rest of the program, however, should be familiar from the material covered above.

After you compile and link the sample programs (and use the relmod utility if you are using Alcyon C), you'll end up with a GEM program whose name ends in .PRG. When you run this program, the mouse pointer will be visible, and may disrupt part of the drawing when you move it. To avoid this, you can either be careful not to move the pointer, and just click a button when you wish to exit, or you can rename the program with the extender .TOS. When you run a TOS program, the mouse pointer becomes invisible, and the screen is cleared automatically. Later, in the chapter on input commands, you'll see how to make the mouse pointer invisible before using graphics commands. You can then modify the shell program to incorporate this feature.

An Assembly Language Program Shell

Setting up a bare-bones assembly language program is more involved than just translating the corresponding shell.c program. For one thing, C programs usually link in a startup file at the beginning of the program to take care of such maintenance chores as allocating a chunk of RAM for a program stack, setting the stack pointer to the address of that stack, and returning any unused RAM to the pool of free memory. Programs written in Alcyon C link in the file appstart.o or gemstart.o at the beginning to take care of these tasks.Megamax C programs get the necessary code from a library module called init.o, the source code for which is supplied in a file called init.c (which uses the inline assembly commands). But assembly language programmers must provide the equivalent functions for each of their programs themselves.

The other problem is that not all assemblers have a .include directive, so we won't be able to include the text of our

shell program in each of our demo programs. Instead, we'll as-
semble the shell program separately, and link the resulting ob-
ject file with the demo program object files. Since our shell
program refers to the demo subroutine in the demo program
file, and the demo programs refer to the VDI data arrays de-
fined in the shell program, we'll use the .xdef and .xref
directives to help resolve these external references. The .xref
directive tells the assembler that the symbol is defined in an-
other object file, while the .xdef tells it that this symbol will be
used by another object file. All of the assembly language ex-
amples in this book have been created with the assembler that
comes as part of the Alcyon C compiler, so bear in mind that
the assembler directives used may be slightly different than
those of other assemblers.

Program 2-4 is the assembler shell program, shell.s.

Program 2-4. shell.s

```
******************************************************************
*                                                                *
*   Shell program to be linked with all assembly language programs. *
*                                                                *
******************************************************************

*** Program equates

bpadr    =      4   * Stack offset to base page address
codelen  =     12   * Base page offset to Code segment length
datalen  =     20   * Base page offset to Data segment length
bsslen   =     28   * Base page offset to BSS  segment length
stk      =  $400   * size of our stack (1K)
bp       =  $100   * size of base page

setblk   =   $4a   * command number of SETBLOCK function
aescode  =   $c8   * command number for AES call
vdicode  =   $73   * command number of vdi call

*** External references

.xref    demo      * import the external demo subroutine
.xdef    vdi       * export the vdi subroutine call,
.xdef    vwkhnd    * the virtual workstation handle,
.xdef    contrl0   * and all of the VDI data arrays
.xdef    contrl1
.xdef    contrl2
.xdef    contrl3
.xdef    contrl4
.xdef    contrl5
.xdef    contrl6
.xdef    contrl7
.xdef    contrl8
.xdef    contrl9
.xdef    contrl10
.xdef    contrl11
.xdef    intin
.xdef    intout
.xdef    ptsin
.xdef    ptsout
```

33

```
*** Program starts here.  Get base page address in a5

    .text
    move.l    a7,a5          * save a7 so we can get the base page address
    move.l    bpadr(a5),a5   * a5 = basepage address

*** Calculate the total amount of memory used by
*** our program (including stack space) in d0

*                            * total memory used =
    move.l    codelen(a5),d0 *   length of code segment
    add.l     datalen(a5),d0 * + length of data segment
    add.l     bsslen(a5),d0  * + length of uninitialized storage segment
    add.l     #stk+bp,d0     * + (size of the base page + our stack)

*** Calculate the address of our stack
*** and move it to the stack pointer (a7)

*                            * stack address =
    move.l    d0,d1          *   size of program memory
    add.l     a5,d1          * + program's base address,
    and.l     #-2,d1         * pick off odd bit to make sure that the
*                            * stack starts on a word boundary (it must).
    move.l    d1,a7          * set stack pointer to our stack
*                            * which is stk bytes above end of BSS

*** Use the GEMDOS SETBLOCK call to reserve the area of memory
*** actually used for the program and stack, and release the
*** rest back to the free memory pool.

    move.l d0,-(sp)          * push the size of program memory
*                            * (first SETBLOCK parameter) on the stack.
    move.l a5,-(sp)          * push the beginning address of the
*                            * program memory area (2nd SETBLOCK parameter).
    clr.w  -(sp)             * clear a dummy place-holder word
    move   #$4a,-(sp)        * finally, push the GEMDOS command number
*                            * for the SETBLOCK function
    trap   #1                * call GEMDOS
    add.l  #12,sp            * and clear our arguments off the stack.

*** Initialize the application with appl_init

    move.l    #0,resv1       * clear global variables
    move.l    #0,resv2
    move.l    #0,resv3
    move.l    #0,resv4
    move      #10,contrl0    * command = appl_init
    move      #0,contrl1     * no integer input parameters
    move      #1,contrl2     * 1 integer output parameter
    move      #0,contrl3     * no address input parameters
    move      #0,contrl4     * no address output parameters
    jsr       aes            * do the call

*** Get the physical screen device handle from graf_handle

    move      #77,contrl0    * command = graf_handle
    move      #0,contrl1     * no integer input parameters
    move      #5,contrl2     * 5 integer output parameters
    move      #0,contrl3     * no address input parameters
    move      #0,contrl4     * no address output parameters
    jsr       aes            * do the call

*** Open the Virtual Screen Workstation (v_opnvwk)

    move      #100,contrl0   * opcode to contrl(0)
    move      #0,contrl1     * no points in ptsin
```

34

```
    move      #11,contrl3      * 11 integers in intin
    move      intout,contrl6 * physical workstation handle to contrl(6)

    movea.l   #intin,a0
    move      #9,d0
initloop:
    move.w    #1,(a0)+         * intin(0)-intin(9) = 1
    dbra      d0,initloop

    move      #2,intin+20      * intin(10) = 2 (Raster Coordinates)


    jsr       vdi
    move      contrl6,vwkhnd * save virtual workstation handle

*** Clear Virtual Workstation

    move      #3,contrl0       * clear work opcode
    move      #0,contrl1       * nothing in ptsin
    move      #0,contrl3       * nothing in intin

    jsr       vdi



*** Do our demo program

    jsr demo

*** Wait for a mouse button push (vq_mouse)

wait:
    move      #124,contrl0     * opcode for Query Mouse Button (vq_mouse)
    move      #0,contrl1
    move      #1,contrl2       * x,y coordinates of mouse returned in ptsout
    move      #0,contrl3
    move      #1,contrl4       * button status returned in intout
    move      vwkhnd,contrl6

    jsr       vdi            * check the button
    cmpi      #0,intout      * if = 0, no button pushed
    beq       wait           * so try again

*** Close Virtual Screen Workstation  (v_clsvwk)

    move      #101,contrl0     * opcode to contrl(0)
    move      #0,contrl1       * no points in ptsin
    move      #0,contrl3       * no integers in intin
    move      vwkhnd,contrl6 * virtual workstation handle

    jsr       vdi

*** Finish the application (appl_exit)

    move      #19,contrl0      * opcode to contrl(0)
    move      #1,contrl2       * 1 integer returned in inout
    move      #0,contrl1
    move      #0,contrl3
    move      #0,contrl4

    jsr       aes

*** Exit back to DOS

    move.l #0,(a7)        * Push command number for terminate program
    trap   #1             * call GEMDOS.  Bye bye!
```

```
***    Make AES function call
***    (after setting parameters)

aes:
   move.l    #apb,d1
   move.w    #aescode,d0
   trap    #2
   rts

*** Make VDI function call
*** (after setting parameters)

vdi:
   move.l    #vpb,d1
   move.w    #vdicode,d0
   trap    #2
   rts


*** Storage space for AES and VDI call parameters

.data
.even


contrl:
contrl0:    .ds.w 1
contrl1:    .ds.w 1
contrl2:    .ds.w 1
contrl3:    .ds.w 1
contrl4:    .ds.w 1
contrl5:    .ds.w 1
contrl6:    .ds.w 1
contrl7:    .ds.w 1
contrl8:    .ds.w 1
contrl9:    .ds.w 1
contrl10:   .ds.w 1
contrl11:   .ds.w 1


global:
version:    .ds.w 1
count:      .ds.w 1
id:         .ds.w 1
private:    .ds.l 1
tree:       .ds.l 1
resv1:      .ds.l 1
resv2:      .ds.l 1
resv3:      .ds.l 1
resv4:      .ds.l 1

intin:      .ds.w 128    * intin and intout used by both also
intout:     .ds.w 128
addrin:     .ds.w 128
addrout:    .ds.w 128
ptsin:      .ds.w 128
ptsout:     .ds.w 128

vwkhnd     .ds.w 1

*** The AES and VDI parameter blocks hold pointers
*** to the starting address of each of the data arrays

apb:  .dc.l contrl,global,intin,intout,addrin,addrout
vpb:  .dc.l contrl,intin,ptsin,intout,ptsout

.end
```

The first part of Program 2-4 requires a bit of explanation. When GEM starts an application program (but not a desk accessory), it allocates all of the system memory to that program. Therefore, if the program wishes to use the system memory-management calls, or any of the AES calls that themselves allocate memory, it must first de-allocate all of the memory it isn't actually using at startup time. The way to do this is with the XBIOS function, SETBLOCK. SETBLOCK is used to reserve a specific area of memory for the program, and return the remaining RAM area to the Operating System's free memory pool. In order to execute this command, you must pass the starting address of the area that you wish to reserve and the size of the area. Please remember that it's only necessary to free memory when you start an application program, not a desk accessory.

Finding the starting address of program memory isn't too difficult, since when you start the program, the second word on the stack points to that location. Finding the size of the program requires a little more knowledge of how program storage space is allocated. The memory area in which a program resides is known as the Transient Program Area (TPA).

At the beginning of the TPA is a 256-byte segment known as the basepage. The basepage contains information about the size and address of each program segment, as well as the command line that is passed to the program. (These are the extra characters you type in when you run a Tos Takes Parameters program whose name ends in .TTP.) After the basepage comes the actual program code, followed by the data area, and then the BSS (Block Storage Segment), which is used to store uninitialized data. So to find the total size of the program area, we take a look in the basepage to find the size of the code, and add that to the size of the data and BSS segments, along with the size of the basepage itself. Since we need a stack area for the program, it makes sense to add the size of the stack to the end of the program and reserve the combined program and stack area together. Once we calculate this area, we can set the stack pointer to the top of program memory and make the SETBLOCK call. Once that's done, we can get on with whatever it is that our program does.

In order to assemble the shell.s program with the Alcyon assembler, we invoke the as68 assembler with the following command:

as68 -u -l shell.s

This creates an object file called shell.o. Since this program does not contain the demo subroutine, it won't link and run properly. In order to get it to function, the least you must do is to create another object module that contains that subroutine. An example of this is Program 2-5, dummy.s.

Program 2-5. dummy.s

```
*********************************
*   Dummy.s -- a do-nothing demo
*********************************

.xdef demo
.text

demo:
    rts

.end
```

Assemble this file in the same way to create the dummy.o file. Next, use the linker to join the two object modules. The command line to use is

link68 [u] dummy.68k=shell,dummy

This creates the dummy.68k, a relocatable program module that must be modified to run under GEMDOS, using the relmod program:

relmod dummy

This produces the dummy.prg program file that can be executed from the desktop. This program just waits until the user presses a mouse button, and then ends. When you substitute the graphics demo subroutines from the programs in subsequent chapters for the dummy demo routine, the program will draw the graphics demonstration, and then waits for the user to press the mouse button.

As with the C example programs, you may find that the mouse pointer image disrupts the picture when you move it for the first time, because it saves and restores the original background image that appeared before your drawing was displayed. The solution is to either rename the program with a .TOS extender or modify the shell program to turn off the mouse pointer before drawing, as we will demonstrate in a later chapter.

Chapter 3

Drawing Points and Lines

The simplest kind of drawing that you can do
is to color in one spot on the screen at a time. The GEM VDI
provides an extremely flexible graphics type called a marker
which is used to perform this function. At its simplest, the
marker function does just what its name suggests—it marks a
single dot. But the extent of this function doesn't stop there.
The basic marker routine, Polymarker, can be used to mark a
number of points at once. The C language format for this
function is

int handle, count, points[COUNT*2];

v_pmarker(handle, count, points);

where *handle* is the ID number for the graphics workstation,
count is the number of points to mark, and *points* is an array
of integers which holds the x and y coordinates for each of
those points. Since each point has two coordinates, there are
twice as many elements in the points array as there are actual
points to mark. Program 3-1 is a brief program that uses the
Polymarker function to draw 64 dots in an 8 × 8 grid.

Program 3-1. pmark1.c

```
/*****************************************************/
/*                                                 */
/*                                                 */
/*    PMARK1.C -- Demonstrates use of the          */
/*    Polymarker function to draw a series         */
/*    of points on the screen.                     */
/*                                                 */
/*                                                 */
/*****************************************************/

#include "shell.c"

demo()
{
int points [128]; /* max of 64 points */
int x,y;

    for (y=0;y<8;y++) /* for each row */
        for (x=0;x<8;x++) /* for each column */
        {
        points[(16*y)+(2*x)]= 100 + (x*4); /* set points */
        points[(16*y)+(2*x)+1] = 100 +(y*4);
        }
```

```
v_pmarker(handle, 64, points);   /* draw all of points */
}

/* End of Pmark1.c */
```

In addition to drawing dots, the Polymarker function can be used to draw several other marker shapes as well. In our example program, single points were drawn on the screen because when we opened the virtual screen workstation, we specified marker type 1 (a single point) as our default marker type in work_in[3]. But, as we saw from the information returned by the v_openvwk call in work_out[8], there are six marker types available for use on the ST screen. These are as follows.

Marker Number	Shape
1	Point
2	Plus sign
3	Star
4	Square
5	Diagonal Cross
6	Diamond

To change the type of marker currently used for drawing, we use the Set Polymarker Type command. In C, the format for this command is

int handle, markerno, type_set;
type_set = vsm_type(handle, markerno);

where *handle* is the workstation ID number, *markerno* is the marker shape number of the symbol you want to use, and *type_set* is the marker shape number of the symbol that was actually set. If you request a marker type that isn't available (for example, a number greater than 6 when using the screen device), the VDI sets the star symbol as the current marker type.

In addition to using a number of different marker shapes, you may also specify the size, or to be more specific, the height, of the marker you wish to use. The only exception to this is the point marker type, which is always exactly one pixel in size. As we have seen from the values returned by v_openvwk in work_out[9] and work_out[53]-[56], the ST screen offers eight different marker sizes, ranging from 15 pixels wide by 11 high to 120 pixels wide by 88 high. Since

the biggest marker size is eight times as large as the smallest, it stands to reason that each larger marker size is 15 pixels wider and 11 pixels taller than the last. The C language format of the Set Polymarker Height command is

int handle, height;
height_set = vsm_height(handle, height);

where *height* is the marker height that you're requesting, and *height_set* is the height of the marker that is actually set. If we request a marker height that isn't available, the VDI sets the height to the next smaller height that's supported. Since we know the exact sizes of markers that are available on the ST screen, we know that the height requested should be an even multiple of 11 no greater than 88. If we did not know the size of the markers available on another device, however, they could be determined by repeatedly trying a height that is 1 less than the tallest known height, and seeing what value is returned in height_set.

The final marker attribute that can be changed is the color in which it is drawn, assuming, of course, that the program is being run on a color monitor. With a monochrome monitor, the background is usually all white, and all drawing is done in black. But on the color monitor, you can have up to four different colors on screen at one time in medium-resolution mode, and up to sixteen different colors in low-resolution mode. Color selection is controlled by sixteen hardware registers. You may think of these as pens, each filled with a different color of ink. By default, you draw with pen 1, which, unless you change it, contains black "ink." That default drawing pen was set by placing a 1 in work_in[4] at the time the virtual workstation was opened. To draw in another color, you must choose another pen with the Set Polymarker Color Index command,

int color_set,handle, pen;
pen_set = vsm_color(handle, pen);

where *pen* is the number of the drawing pen (hardware color register) that you wish to use, and *Pen_set* shows the actual drawing pen that was selected—which may differ from the one you requested if you asked for an invalid pen number. We'll discuss the default colors of the drawing pens, as well as how to change those colors, in the next chapter.

Program 3-2 shows the five sizable marker types in five

different sizes. If you have a color monitor, some of them will appear in different colors as well:

Program 3-2. pmark2.c

```
/************************************************/
/*                                            */
/*                                            */
/*    PMARK2.C -- Demonstrates use of         */
/*    different sizes and shapes              */
/*    of markers.                             */
/*                                            */
/*                                            */
/************************************************/

#include "shell.c"

demo()
{
int points [2];
int x,y;


    for (y=0,points[1]=3;y<5;y++) /* for each row */
        {
        vsm_height(handle, 11+(y*11)); /* set marker height */
        vsm_color(handle,y+1);         /* set color */
        points[1]+=(11*y);             /* set points */
        for (x=0,points[0]=(8+(y*8))/2;x<5;x++)
            {
            vsm_type(handle, 2+x);     /* change marker shape */
            if (x>0)points[0]+=(12*(y+1));
            v_pmarker(handle, 1, points); /* draw marker */
            }
        }


}

/* End of Pmark2.c */
```

You may have noticed something peculiar about the size and positioning of the markers on the display. In order to line them up with the left edge of the screen, it was necessary to offset them a number of pixels from the left. That's because the x,y coordinate given for a marker that's bigger than a single point specifies the center point of that marker, not its upper left corner. Therefore, a marker placed at 0,0 would have both its top half and its left half cut off by the edge of the screen.

Another thing you may have noticed is that we didn't have to horizontally space the markers an even multiple of 15 pixels apart. That's because the marker size measurement is for the marker's *cell*, which includes not only the actual area filled by the marker, but also some blank space around the
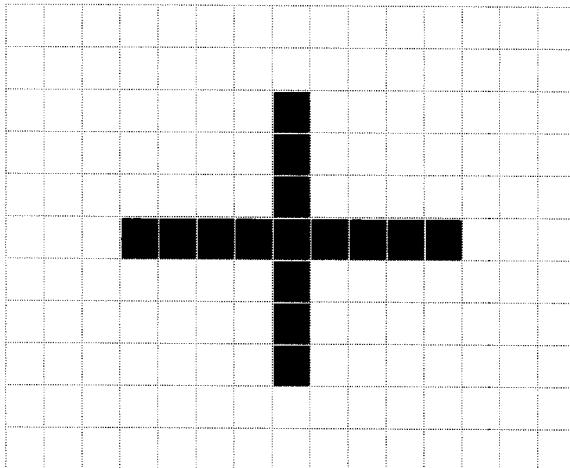
border of the marker. For example, in the 11-pixel-high plus-sign marker (number 2), the horizontal line is actually only 9 pixels long, not 15, and the vertical line is 7 pixels high, not 11 (Figure 3-1). The rest of the cell is taken up by blank pixels, distributed evenly on all four sides of the marker. To find the actual drawing width of any marker, divide the height by 11; multiply the result by 8; then add 1. To find the drawing height, divide the height by 11; multiply that result by 6; then add 1. The mathematical formulas are:

Drawing width = ((height / 11) * 8) + 1
Drawing height = ((height / 11) * 6) + 1

Figure 3-1. Polymarker 2, 11 Pixels High

If you need to know during the course of a program just what the current Polymarker settings are, you can use the VDI function called Inquire Current Polymarker Attributes. The format of the C command is:

int handle,
   settings[4];

vqm_attributes(handle, settings);

The function returns the current polymarker settings in the four elements of the settings array. Element 0 contains the current marker type, element 1 contains the pen number, element 2 holds the drawing mode (we'll explain that one in the

next chapter as well), and element 3 contains the current polymarker height.

Lines

The next step up in complexity from marking points on the screen is drawing straight lines. As with markers, the principal VDI line-drawing command allows you to draw several of these at the same time. For that reason, it's called Polyline. The C syntax for this call is

int handle, count, points[COUNT*2];
v_pline(handle, count, points);

where *count* is the number of endpoints that will be joined, and *points* is an array that contains the *x* and *y* coordinates for each of these points (in the format *x,y,x1,y1,x2,y2*, and so on). Since each point is described by two coordinates, there are twice as many elements in the points array as there are points. Although it takes two points to describe a line, the endpoint of one line is always the beginning point of the next, except for the first line. Therefore, the number of lines drawn by a Polyline command is always one less than the number of points.

Anytime you use the Polyline command to draw a closed polygon, the first point and the last point will be the same. So, in order to draw a square with Polyline, you need five coordinate pairs in the points array, with the first and last pair being exactly the same.

Program 3-3 is a sample program that demonstrates use of the Polyline command.

Program 3-3. pline1.c

```
/************************************************/
/*                                            */
/*                                            */
/*    PLINE1.C -- Demonstrates use of the     */
/*    Polyline function to draw a series      */
/*    of connected lines.                     */
/*                                            */
/*                                            */
/************************************************/

#include "shell.c"

#define REPS 15          /* number of spirals */
#define NEXT points[index++]  /* set next point */
#define STEP 12          /* space between lines */

demo()
{
```

```
int index=0,
    c,
    x,y,xstep,ystep,dx,dy,
    points[8*REPS+2];

    if (work_out[0] == 639)dx=STEP;  /* set full horiz step */
        else dx = STEP/2;            /* except for lo-res */
    if (work_out[1] == 399)dy=STEP;  /* set full vert step */
        else dy =  STEP/2;           /* except for color */

    xstep = work_out[0]-dx;          /* set horiz line length */
    ystep = work_out[1]-dy;          /* set vert line length */

    NEXT = x = dx/2;     /* set first point */
    NEXT = y = dy/2;

 /* For each repetition of the spiral... */
    for (c=0;c<REPS;c++)
        {
        NEXT = (x+=xstep);      /* set four x,y coords */
        NEXT = y;
        NEXT = x;
        NEXT = (y+=ystep);
        NEXT = (x-=(xstep-=dx));
        NEXT = y;
        NEXT = x;
        NEXT = (y-=(ystep-=dy));
        ystep -=dy;             /* & change the line lengths */
        xstep -=dx;
        }

    /* draw all of the lines */
    v_pline(handle, 4*REPS+1, points);

}

/* End of Pline1.c */
```

As you can see, Program 3-3 adapts itself to any type of monitor by reading the horizontal and vertical resolution from work_out[0] and work_out[1], and scaling the length of the lines and the size of the space between lines accordingly. Notice also how the use of macro definitions like NEXT, STEP, and REPS saves a lot of typing and allows us to easily vary the size between the "squirals" and repetitions (though if you want more than 15 repetitions you will have to increase the size of the ptsin array in the shell program). These macro definitions also have value as program documentation.

Patterned Lines

In addition to solid lines, the VDI also lets you draw patterned lines composed of dots and/or dashes. In order to draw these, we must set the line-drawing pattern with the Set Polyline

Line Type command. The C syntax for this routine is

int handle, pattern;
pattern_set = vsl_type(handle, pattern_no);

where *pattern_no* is the number of any of the seven available line-drawing patterns. The number of the pattern that was actually set is returned in the variable *pattern_set*.
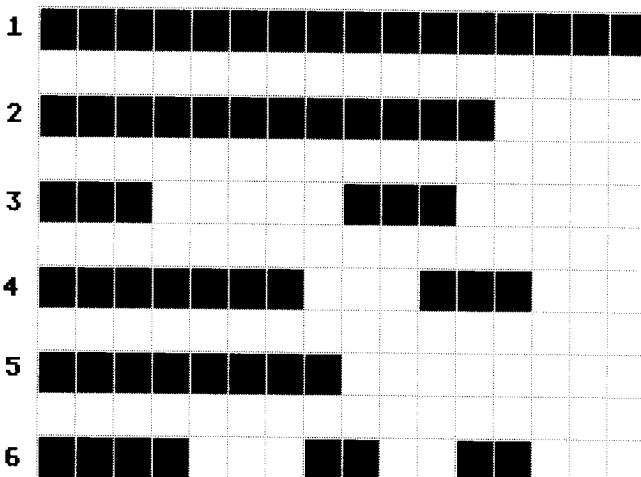
The line-drawing pattern is composed of 16 pixels lined up in a row. Each pixel is either colored in with the line-drawing color (on), or colored in with the background color (off). These patterns can be represented by a single 16-bit binary (base 2) number. For example, a pattern in which an "on" pixel alternates with an "off" pixel can be represented by the binary number 1010101010101010, which has a decimal value of 43690. The Atari ST screen driver supports seven types of line-drawing patterns (as we saw from the value returned in work_out[6] when we opened the virtual screen workstation). These are (see Figure 3-2)

Pattern Number	Binary value	Pattern
1	1111111111111111	Solid line
2	1111111111110000	Long dash
3	1110000011100000	Dotted line
4	1111111000111000	Dash-dot
5	1111111110000000	Dashed line
6	1111000110011000	Dash-dot-dot
7	User-defined style (must be set with vsl_udsty call before vsl_type call)	

The reason that we get a solid line as our default pattern is because we placed a 1 representing pattern number 1 in work_in[3] when we opened our virtual workstation, thus requesting the solid line as our default. As long as the GDOS extension is installed, however, it's possible to specify another value as the default. If the GDOS extension is not installed, the default line type will be the solid line, regardless of the value you put in work_in[3]. Another point to note about these line-drawing patterns is that the VDI makes no attempt to scale them according to the screen display used, so the pattern may look fatter or thinner depending on whether you are in lo-res, medium-res, or hi-res mode. And, since the horizontal resolution varies significantly from the vertical resolution, the pattern of a dotted line that is drawn horizontally looks quite different from that of the same line drawn vertically.

Figure 3-2. The Six GEM VDI Line-Drawing Patterns

Although the VDI supplies six preset line patterns for the sake of convenience, it also provides for a user-defined style, which allows you to choose any of the 65,536 possible combinations of lit and unlit pixels for the line-drawing pattern. Before we select pattern 7, however, we should first tell the VDI which pattern it represents. We do this by calling the Set User-Defined Line Pattern function. In C, this function appears in the following format:

```
int handle, pattern;
vsl_udsty (handle, pattern);
```

where *pattern* is a 16-bit number representing the 16 pixels of the line-drawing pattern. As stated above, the way to translate the on/off patterns into a 16-digit base-2 number is by writing a 1 for every lit pixel and a 0 for every unlit pixel. In binary math, the rightmost digit has a value of 1, and each subsequent digit has a value that is twice that of the digit to its right. Therefore, the binary number 1010 has a decimal value of 10:

Binary digits	1010
Decimal value of each digit	8020
	8+0+2+0 = 10

Line Color

On ST systems that have a color monitor, you can specify
which drawing pen will be used for line drawing, and thus
control the color of the line that is drawn. On the mono-
chrome system, you're limited to pen 0 (white) or pen 1
(black). The VDI command for selecting the pen is Set
Polyline Color Index, which corresponds to the following C
function:

int handle, pen_number
set_color = vsl_color(handle, pen_number);

where *pen_number* is the number of the drawing pen to use.
The pen number is referred to in the GEM literature as the
color index, since it represents an offset from the beginning of
the color lookup table. Pen 0 is the background color, and pen
1 is the default foreground color (black), which we set with
work_in[2] when we opened the virtual workstation. If you se-
lect a color higher than the number of colors available, pen 1
will be set. We'll discuss the other color defaults and how to
change them in the next chapter.

Program 3-4 demonstrates the use of the line-drawing
pattern settings and the color settings with the Polyline
command.

Program 3-4. pline2.c

```
/*******************************************/
/*                                         */
/*                                         */
/*    PLINE2.C -- Demonstrates patterned   */
/*    line drawing and color selection     */
/*                                         */
/*                                         */
/*                                         */
/*******************************************/

#include "shell.c"

#define REPS 4    /* number of polygons to draw */
#define STEP 15   /* distance between them */

demo()
{
int index=0,
    c,
    x,y,
    xmax,xmin,ymax,ymin,dx,dy,
    points[18];

    if (work_out[0] == 639)dx = STEP*2; /* full horiz step */
        else dx = STEP;                 /* except lo-res */
    if (work_out[1] == 399)dy = STEP*3; /* full vert step */
        else dy = (STEP*3)/2;           /* except for color */
```

```
      xmax = work_out[0]-(3*dx);    /* set initial margins */
      xmin = 3*dx;
      ymax = work_out[1];
      ymin = 0;

  /* Set user-defined line drawing pattern */
  /* 0xAAAA = 1010101010101010 binary */
      vsl_udsty(handle, 0xAAAA);

      for (c=0;c<REPS;c++)    /* for each polygon */
      {

      points[12]=points[14] = xmin;    /* set points */
      points[0]=points[10]=points[16] =( xmin+=dx);
      points[1]=points[3]=points[17] = ymin;
      points[5]=points[15] = (ymin+=dy);
      points[4]=points[6] = xmax;
      points[2]=points[8] = (xmax-=dx);
      points[9]=points[11] = ymax;
      points[7]=points[13] =   (ymax-=dy);

      xmin+=dx;  /* shorten offsets */
      xmax-=dx;

       vsl_color(handle, 1+c);  /* change colors */
       vsl_type(handle,1+(c*2)); /* change line pattern */

       v_pline(handle, 9, points);  /* draw the polygon */
       }
 }

/* End of Pline2.c */
```

Line Width

In Program 3-4, the lines that were drawn were all 1 pixel wide. But as you can see by the values returned in work_out[7], work_out[49], and work_out[51] when we opened the virtual workstation, the ST display driver can draw lines in any width from 1 pixel to 40 pixels. The function used to specify the width of the lines that the VDI draws is Set Polyline Line Width, which in C looks like this:

width_set = vsl_width(handle, width)

where *width* is the line width in pixels. In order to keep everything symmetrical, the VDI only uses odd-numbered line widths. If you request a line width that's unavailable, either because it's larger than the maximum width or because it's an even number, the VDI will set the width to the next lower available width. The actual line width that was set by the call is returned in the variable *width_set*.

Although most of the line-drawing settings can be used together, patterned lines cannot be drawn on the ST screen if their width is set to a value greater than one pixel. Whenever

you use thicker lines, they are drawn as solid lines, regardless of the current line-pattern setting.

Line End Styles

The final line-drawing setting is a fairly obscure function that allows you to designate the end style for the line. By default, the ends of a line are squared off, but by using the Set Polyline End Styles command, you may instruct the VDI to round off either end of the line, or to place an arrow head at either end of the line. The arrow head is positioned so that its tip is placed at the last point of the line. Although the GEM literature states that the rounded end is drawn so that the center of the rounded line is positioned at the end of the line, experience on the ST shows otherwise. The rounding is added on to the end of the line, increasing its length by about half its thickness. While arrows can be used with any width of line, the rounding is not really noticeable unless you are drawing a line that is thicker than five pixels. The syntax of the C command used to set the end styles is

int handle, begin_style, end_style;
vsl_ends(handle, begin_style, end_style);

where *begin_style* and *end_style* contain a number code from 0 to 2, indicating what style will be used to draw the beginning and end of the line. The number 0 represents a squared-off end, 1 indicates that an arrow is to be drawn, and a 2 means that the end should be rounded off. If an invalid number is given for either of these, the squared-end style is selected by default.

Program 3-5 shows the use of thickened lines, and of the two stylized types of endpoints.

Program 3-5. pline3.c

```
/*********************************************/
/*                                           */
/*                                           */
/*    PLINE3.C -- Demonstrates drawing       */
/*    lines of various widths and end styles */
/*                                           */
/*                                           */
/*                                           */
/*********************************************/

#include "shell.c"

#define REPS 5   /* number of wide lines to draw */
```

```
demo()
{
int index=0,
    c,
    xmax,xmin,ymax,dy,
    points[6];


    points[1] = 12;   /* set margins and offsets */
    dy = 31;
    xmax = work_out[0]+16;
    ymax = work_out[1]+10;
    xmin = (-2*dy);

  /* make beginning of line rounded, end w/arrowhead */
    vsl_ends(handle,2,1);

    for (c=0;c<REPS;c++)   /* for each line */
    {

    points[0] = points[2] = (xmax-*(dy+4)); /* set points */
    points[4] = ( xmin+=(2*dy+10) );
    points[3] = points[5]= ( ymax-*(dy+18) );

    vsl_width(handle, (dy-*6 ) ); /* change width */
    vsl_color(handle,c+1);         /* change colors */
    v_pline(handle, 3, points);   /* draw it */
    }
}

/* End of Pline3.c */
```

If you ever need to find out from your program what the current line-drawing settings are, you can use the function Inquire Current Polyline Attributes (though this is fairly inefficient, as you should be able to keep track of the settings in the program without having to inquire). The C format for this call is

int handle, settings[4];
vql_attributes(handle, settings);

where *settings* is the address of the array in which the function stores the current line-drawing settings. After the call has been successfully completed, the following information can be found in the various elements of the array:

Element	Setting
[0]	Line pattern
[1]	Line-drawing pen
[2]	Draw mode
[3]	Line width

The function also returns the beginning end style and the ending end style in intout[3] and intout[4], respectively. These

53

values are not transferred to the settings array by the C function bindings.

Line-Drawing GDPs

There are some other VDI functions that draw figures using lines, and these functions share the same line-drawing settings as Polyline. For reasons having to do with the older graphics systems from which they evolved, they're referred to as Generalized Drawing Primitives, or GDPs, for short. The only practical programming difference between GDPs and any other drawing function is that they all have the same opcode, so assembly language and BASIC programmers will have to remember to set both the opcode in contrl(0) and the sub-function ID in contrl(5) before calling one of these functions. C programmers will not have to worry about this, since the bindings take care of this detail for them. The line-drawing GDPs allow you to draw circles, or any part of a circle; ellipses, or any part of an ellipse; and rounded rectangles.

The first of these functions is called Arc, and it allows you to draw any segment of a circle. The C function call is
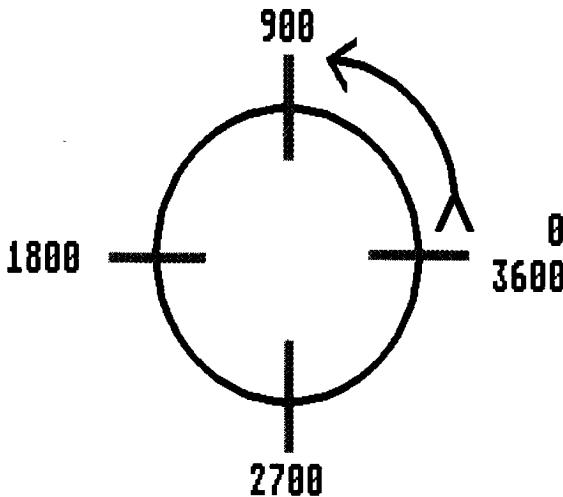
```
int handle, x, y, radius, begin_angle, end_angle;
v_arc(handle, x, y, radius, begin_angle, end_angle);
```

where x and y are the coordinates for the center point of the circle, *radius* is its radius, and *begin_angle* and *end_angle* give the starting and ending points of the arc.

In order to know which beginning and ending angles to specify to draw a particular segment of the circle, we have to understand how the VDI refers to angles. The interior angles of a circle add up to 360 degrees, and since the VDI thinks in tenths of a degree, the points on a circle go from 0 to 3600. The VDI designates the rightmost point on the circle (the three o'clock position) as 0 or as 3600, depending on whether it's used as the starting angle or ending angle. All drawing proceeds in a counterclockwise direction, so the topmost point is designated 900, the leftmost as 1800, and the bottom-most as 2700 (Figure 3-3).

Figure 3-3. Drawing Angles

Thus, to draw the top right quarter of a circle whose center point is at 100,100, and whose radius is 50 pixels, you use the command:

v_arc(handle, 100, 100, 50, 0, 900);

To draw a complete circle, you need only to specify 0 as the starting angle, and 3600 as the ending angle.

Keep in mind that the VDI adjusts the vertical radius of the circle for the aspect ratio of the display screen (the ratio of the width of each pixel to its height), so it always appears to be round. If it did not make this kind of adjustment, a circle that appears round on the low-resolution screen would look tall and thin on the medium-resolution screen. To find the effective vertical radius of the circle, multiply the radius by the value that was in output[3] after you opened the workstation, and divide the result by the value that was in output[4]. Thus, the box that contains a circle with radius r is r units wide and (r * work_out[3] / work_out[4]) units high. An almost identical function allows you to draw any segment of an ellipse. The only difference between a circle and an ellipse is that the vertical radius of a circle is automatically calculated to scale from

the horizontal radius that you supply, so that it always appears to be round, while you supply both the horizontal and vertical radius values for the ellipse, so that it may be oval in shape. The C call for the Elliptical Arc function is:

int x, y, xradius, yradius, begin_angle, end_angle;
v_ellarc(handle, x, y, xradius, yradius, begin_angle, end_angle);

All of the parameters for this call are the same as those used by Arc, except, instead of a single radius, there are variables for both the horizontal (xradius) and vertical (yradius) radii.

The final line-drawing function is called Rounded Rectangle. As its name suggests, it's used to draw rectangles whose corners are rounded. GEM applications often use rounded boxes in dialogs as push buttons, as they give the program a more polished look than do boxes with square corners. Since you cannot control the amount of rounding, however, you'll find that some of the smaller rounded boxes look more like circles.

One thing to note about rounded rectangles is that they're affected by line-drawing settings like line width and pattern, but not by end styles, since, properly speaking, they have neither beginning nor ending points. The C syntax for the rounded rectangle function is

int handle, sides[4]
v_rbox(handle, sides);

where *sides* is an array that gives the coordinates for the four sides of the box. The elements of this array are:

Element	Position
[0]	Left Edge
[1]	Top Edge
[3]	Right Edge
[4]	Bottom Edge

Program 3-6 demonstrates the use of the line-drawing GDPs, and shows how they are affected by the various line-drawing settings.

Program 3-6. gdpline1.c

```
/*******************************************************/
/*                                                    */
/*                                                    */
/*    GDPPLINE1.C -- Demonstrates use of the          */
/*    line-drawing GDP functions, and how             */
/*    they are affected by line settings.             */
/*                                                    */
/*                                                    */
/*******************************************************/

#include "shell.c"

#define STEP 10                 /* space between lines */
int dx, dy;

demo()
{

    if (work_out[0] == 639)dx=STEP; /* set full horiz step */
        else dx = STEP/2;           /* except for lo-res */
    if (work_out[1] == 399)dy=STEP; /* set full vert step */
        else dy =  STEP/2;          /* except for color */

  showarc();    /* do circle demo */
  showellarc(); /* do ellipse demo */
  showrbox();   /* do rounded rectangle demo */
  }

showarc()
{
  int c;

    vsl_width(handle,dx); /* set wide line */
    for (c=1;c<8;c++)
        {
        vsl_color(handle,c); /* change color */
 /* draw concentric semi-circles */
        v_arc(handle,17*dx,20*dy,c*(dx+dy)+dx+dy,0,1800);
        }

}

showellarc()
{
  int c;

    vsl_width(handle,1);       /* line width to 1 */
    vsl_udsty(handle,0xA5A5); /* define line pattern 7 */
    for (c=1;c<10;c++)
        {
        vsl_color(handle,c);    /* change color */
        vsl_type(handle,c);     /* change line pattern */
  /* give last ellipse arrow heads */
        if (c>8) vsl_ends(handle,1,1);
        v_ellarc(handle,48*dx,20*dy,dx*c,dy*c*2,0,3600);
        }

}

showrbox()
{
  int c,points[4];
```

```
vsl_type(handle,3);
for (c=0;c<5;c++)
    {
    vsl_width(handle,c*2); /* set width */
    vsl_color(handle,c);    /* and color */
    points[0]=(c+1)*dx+(3*c);
    points[1]=(c+22)*dy+c;
    points[2]=work_out[0]-(c+26)*dx-(3*c);
    points[3]=work_out[1]-(c+1)*dy-c;
    v_rbox(handle,points);
    }

}

/* End of Pline1.c */
```

Assembly Language Example

Though the principles of using the VDI line-drawing functions
are the same in assembly language as in C, the assembly lan-
guage programmer does have to move the input parameters
directly into the VDI data arrays. In order to illustrate this pro-
cess, compare the assembly language version of the GDP lines
program (Program 3-7) with Program 3-6. By comparing the
two versions, assembly language programmers should get a
better idea of how to translate the other C sample programs to
assembly language.

   You may notice that in some of the assembly language
VDI calls, we didn't bother to set all of the contrl and intin
values if we knew that they were set correctly during previous
calls. Generally speaking, the VDI will not disturb your input
parameter arrays, so once you put the workstation id handle in
contrl6, for example, you can assume that it will still be there
for subsequent VDI calls. Keep in mind, though, that as we
have set things up here, the VDI and AES share the contrl ar-
ray, so if your program uses AES calls, you should be aware of
the possibility for interference from that quarter. Also, since
the process of calling the VDI uses registers d0 and d1, you
must preserve the contents of those registers if you want them
to survive between GEM calls. In general, you should assume
that the ST system will use the first couple of data and ad-
dress registers, and not place values that you wish to preserve
between system calls in those registers.

Program 3-7. gdplines.s

```
*************************************************
*                                               *
*                                               *
*       GDPLINES.S -- assembly version of       *
*       GDP line drawing demo                   *
*                                               *
*                                               *
*                                               *
*************************************************


.xdef demo
.xref vwkhnd
.xref contrl0
.xref contrl1
.xref contrl2
.xref contrl3
.xref contrl4
.xref contrl5
.xref contrl6
.xref contrl7
.xref contrl8
.xref contrl9
.xref contrl10
.xref contrl11
.xref intin
.xref intout
.xref ptsin
.xref ptsout

.text

demo:

    move    dx,d0
    move    dy,d1
    cmp     #639,intout   * if high-res or med-res
    bne     skip1
    add     d0,d0         * double dx
    move    d0,dx

skip1:
    cmp     #399,intout+2 * if high-res
    bne     skip2
    add     d1,d1
    move    d1,dy         * double dy

skip2:
    jsr     showarc       * do arc demo
    jsr     showell       * do ellipse demo
    jmp     showrbox      * do rounded rectangle demo

*** Showarc subroutine
showarc:

*** Set line width
    move    #16,contrl0   * opcode for line width
    move    #1,contrl1    * 1 point in ptsin
    move    #0,contrl3    * no integer parameters in intin

    move    dx,ptsin
    move    #0,ptsin+2    * set width to dx
    jsr     vdi

    move    #6,d4         * loop counter
```

```
arc:
*** Set line color
    move     #17,contrl0    * opcode for line color
    move     #0,contrl1     * no points in ptsin
    move     #1,contrl3     * 1 integer parameter in intin

    move     d4,intin       * line color
    jsr      vdi

*** Draw arc

    move     #11,contrl0    * opcode for GDP
    move     #4,contrl1     * 4 points in ptsin
    move     #2,contrl3     * 2 integer parameters in intin
    move     #2,contrl5     * GDP ID for arc

    move     #0,intin       * starting & ending angle
    move     #1800,intin+2

    lea      ptsin(PC),a0      * zero out ptsin(0)-(7)
    move     #7,d0
iloop:
    move     #0,(a0)+
    dbra     d0,iloop

    move     dx,d0          * get dx and save a copy
    move     d0,d1
    move     dy,d2

    mulu     #17,d0
    move     d0,ptsin       * x coord of circle center
    add      d2,d1          * d1 =dx+dy
    move     d4,d0
    addq     #2,d0          * d0 = d4+2
    mulu     d0,d1
    move     d1,ptsin+12    * radius of circle to d1

    mulu     #20,d2
    move     d2,ptsin+2     * y coord of circle center

    jsr      vdi

    dbra     d4,arc
    rts

*** Showell subroutine
showell:

*** Set line width
    move     #16,contrl0    * opcode for line width
    move     #1,contrl1     * 1 point in ptsin
    move     #0,contrl3     * no integer parameters in intin

    move     #1,ptsin
    move     #0,ptsin+2     * set width back to 1
    jsr      vdi

*** Set user-defined line pattern
    move     #113,contrl0   * opcode for line width
    move     #0,contrl1     * no point in ptsin
    move     #1,contrl3     * 1 integer parameter in intin
    move     #$A5A5,intin   * dotted line
    jsr      vdi

    move     #8,d4          * loop counter
```

```
ell:

*** Set line color
    move      #17,contrl0  * opcode for line color
*  contrl1 and contrl3 set by last vdi call

    move      d4,intin     * line color
    jsr       vdi

*** Set line type
    move      #15,contrl0  * opcode for line pattern

* contrl1, contrl3 and intin are still set correctly from last call

    jsr       vdi

    cmp       #1,d4
    bne       skip3
    move      #108,contrl0 * opcode for set end styles
    move      #0,contrl1   * no points in ptsin
    move      #2,contrl3   * 2 ints in intsin
    move      #1,intin
    move      #1,intin+2   * arrows at both ends
    jsr       vdi

skip3:
*** Draw ellarc

    move      #11,contrl0  * opcode for GDP
    move      #2,contrl1   * 2 points in ptsin
* 2 integer parameters in intin, same as last call
    move      #6,contrl15  * GDP ID for ellarc

    move      #0,intin     * starting & ending angle
    move      #3600,intin+2

    move      dx,d0        * get dx and save a copy
    move      d0,d1
    move      dy,d2        * get dy and save a copy
    move      d2,d3

    mulu      #48,d0
    move      d0,ptsin     * x coord of ellipse

    move      d4,d0
    addq      #1,d0        * d0 = d4+1
    mulu      d0,d1        * times dx
    move      d1,ptsin+4   * xradius of ellipse

    mulu      d0,d3
    add       d3,d3
    move      d3,ptsin+6   * yradius of ellipse


    mulu      #20,d2
    move      d2,ptsin+2   * y coord of ellipse center

    jsr       vdi

    dbra      d4,ell
    rts


*** Rounded rectangle demo
```

61

```
showrbox:


*** Set line type
    move       #15,contrl0   * opcode for line pattern
    move       #0,contrl1    * no points in ptsin
    move       #1,contrl3    * 1 integer parameter in intin

    move       #3,intin      * line pattern 3
    jsr        vdi

    move       #4,d4         * set counter
rbox:

*** Set line color
    move       #17,contrl0   * opcode for line color
*   move       #0,contrl1    * no points in ptsin
*   move       #1,contrl3    * 1 integer parameter in intin

    move       d4,intin      * line color
    jsr        vdi

*** Set line width
    move       #16,contrl0   * opcode for line width
    move       #1,contrl1    * 1 point in ptsin
    move       #0,contrl3    * no integer parameters in intin

    move       d4,d5
    add        d5,d5
    move       d5,ptsin
    move       #0,ptsin+2    * set width to 2 *c
    jsr        vdi

*** Draw rounded rectangle

    move       #11,contrl0   * opcode for GDP
    move       #2,contrl1    * 2 points in ptsin
*   move       #0,contrl3    * 0 integer parameters in intin
    move       #8,contrl5    * GDP ID for rounded rectangle

    move       dx,d0         * get dx and save a copy
    move       d0,d1
    move       dy,d2         * get dy and save a copy
    move       d2,d3

    move       d4,d5         * copy the counter
    addq       #1,d5         * c+1
    mulu       d5,d0         * (c+1)*dx
    subq       #1,d5
    mulu       #3,d5         * d5 = 3*c
    add        d5,d0
    move       d0,ptsin      * first x of box

    move       d4,d0
    add        #22,d0        * d0 = c + 22
    mulu       d0,d2         * times dy
    add        d4,d2
    move       d2,ptsin+2    * first y of box

    move       d1,d2         * put dx in d2
    addq       #4,d0         * d0 = c+26
    mulu       d0,d1         * times dx
    add        d1,d5         * + 3*c
    mulu       #64,d2        * screen max
    sub        d5,d2         * - offset
    move       d2,ptsin+4    * second x point
```

```
        move    d4,d0       * d0 = c
        addq    #1,d0       * +1
        mulu    d3,d0       *  times dy
        add     d4,d0       * + c
        mulu    #40,d3      * max y
        sub     d0,d3       * -offset
        move    d3,ptsin+6  * second y coord

        jsr     vdi

        dbra    d4,rbox
        rts


**** data section


        .data
        .even
dx:             .dc.w   5
dy:             .dc.w   5



        .end
```

Line-Drawing VDI Calls and BASIC

ST BASIC doesn't support VDI Marker commands, and it supports some, but not all, of the line-drawing functions. It is quite possible, however, to access the remaining functions with VDISYS(1) calls. Of the line-drawing functions that we have discussed, ST BASIC fully supports arcs and elliptical arcs. The syntax for these commands is very similar to that of the C functions:

CIRCLE x, y, radius [,start angle, end angle]
ELLIPSE x, y, xradius, yradius [,start angle, end angle]

where the input parameter values are the same used by v_arc( ) and v_ellarc( ). ST Basic does not support the rounded rectangle function with its own keyword.

Of the line-drawing settings, ST BASIC supports the set line-drawing color function:

COLOR text color, fill color, line drawing color, fill style, fill
    index

The third input parameter of this command is used to set the line-drawing color. ST BASIC does not contain commands for setting line width or end styles. While the first ST BASIC does

not allow you to set the line-drawing pattern, it appears that
the upgraded BASIC will contain a LINEPAT command:

LINEPAT pattern number, user-defined style

where *pattern number* is the pattern chosen, and *user-defined
style* is the 16-bit pattern value that is set with vsl_udsty( ).
Likewise, though the first ST BASIC only allows you to draw
one line at a time using the LINEF command, the new BASIC
will probably offer the MAT DRAW or MAT LINEF command:

MAT LINEF points, array

where *points* is the number of vertices, and *array* is the name
of a data array holding the x and y coordinates for those
vertices.
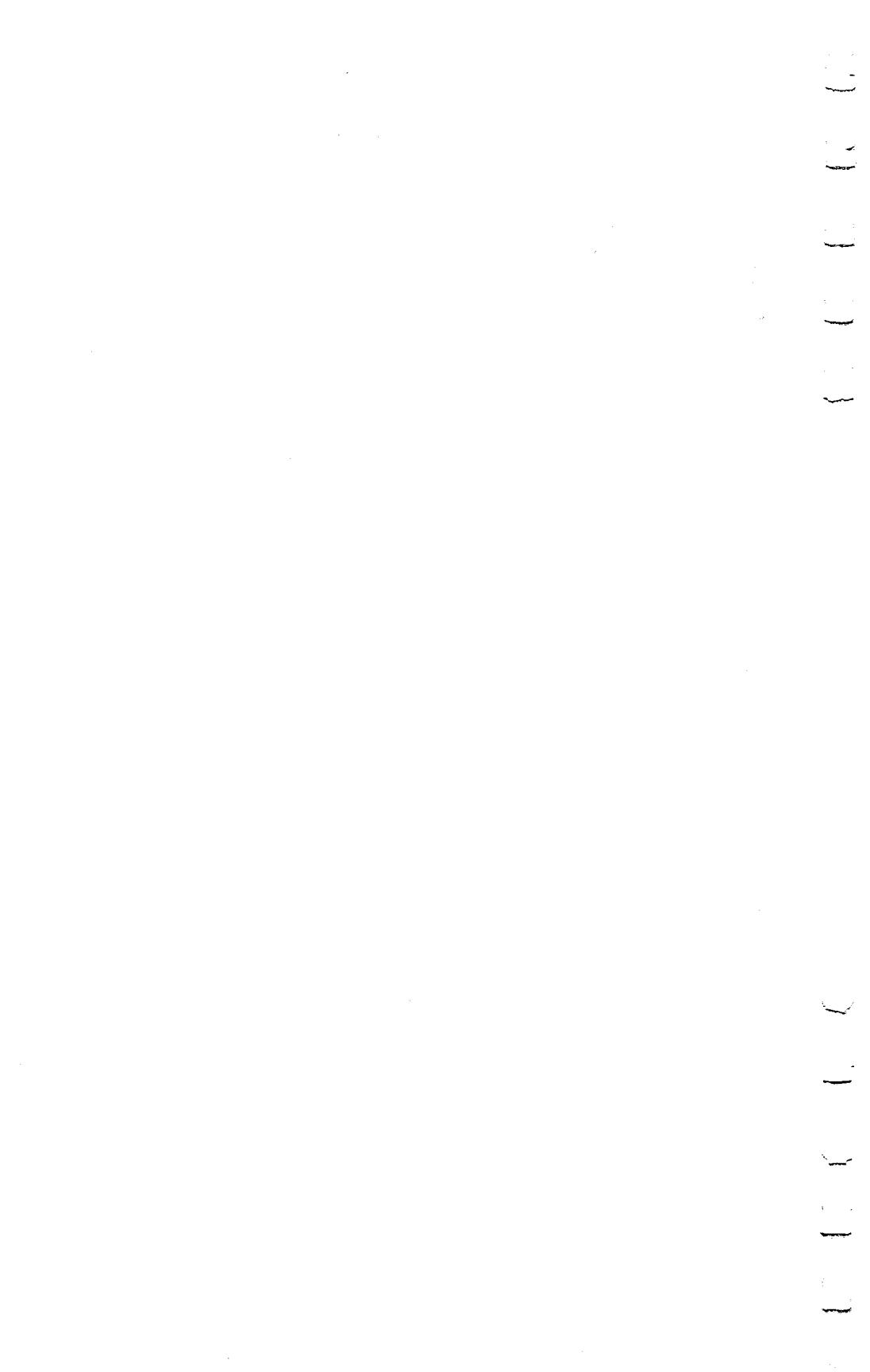
Program 3-8 shows the use of the VDISYS( ) command to
make direct calls to the VDI functions from BASIC. Note par-
ticularly that the settings you make using these functions, such
as line width and drawing pattern, do have an effect on the
output of keyword commands such as LINEF and CIRCLE.

Program 3-8. lines.bas

```
100   fullw 2: clearw 2
110   res = peek(systab)
111   if (res<4) then xmax = 639 else xmax = 319
112   if (res>1) then ymax = 199 else ymax = 399
120   REM Set polymarker Height
130   poke contrl,19 :REM opcode for set pmarker height
140   poke contrl+2, 1 :REM number of points in ptsin
150   poke contrl+6, 0 :REM nothing in intin array
160   poke ptsin,0
170   poke ptsin+2,77 :REM height of marker
180   vdisys(1)
190   REM
200   for x=0 to 4
210   REM
220   REM Set Polymarker color
230   poke contrl,20 :REM opcode
240   poke contrl+2,0
250   poke contrl+6,1
260   poke intin,x+1
270   vdisys(1)
280   REM Set Polymarker type
290   poke contrl,18
300   poke contrl+2,0
310   poke contrl+6,1
320   poke intin,2+x
330   vdisys(1)
340   REM Draw Polymarker
350   poke contrl,7
360   poke contrl+2,1
370   poke contrl+6,0
380   poke ptsin,61%x+30
390   poke ptsin+2,ymax/4
400   vdisys(1)
410   REM
```

```
420    next
430    REM
440    for x = 0 to 5
450    REM
460    REM Set line drawing pattern
470    poke contrl,15
480    poke contrl+2,0
490    poke contrl+6,1
500    poke intin,x+2
510    vdisys(1)
520    REM
530    REM Draw Rbox
540    poke contrl,11
550    poke contrl+2,2
560    poke contrl+6,0
570    poke contrl+10,8
580    poke ptsin,10+(10%x)
590    poke ptsin+2,ymax/2+(7%x)
600    poke ptsin+4,(xmax/2)-(10%x)
610    poke ptsin+6,ymax-20-(7%x)
620    vdisys(1)
630    REM
640    next x
650    REM
660    REM Set End Styles
670    poke contrl,108
680    poke contrl+2,0
690    poke contrl+6,2
700    poke intin,1
710    poke intin+2,2
720    vdisys(1)
730    REM
740    for x = 0 to 3
750    REM Set Line Width
760    poke contrl,16
770    poke contrl+2,1
780    poke contrl+6,0
790    poke ptsin,15-(x%5)
800    poke ptsin+2,0
810    vdisys(0)
820    REM
825    color 1,0,x+2
827    REM
830    linef xmax-60-(40%x),50,xmax-50-(20%x),ymax-70
840    rem
850    next x
```

Chapter 4

Color and Other Graphics Settings

Now that you've had a little experience using some of the output functions, let's take a look at some of the settings that can affect graphics output, regardless of the output function used. These include color settings, drawing modes, and clipping rectangles.

Color Settings

In the previous chapter, we discussed how to change the color of the line or marker being drawn, but we didn't explain how you could select a particular color like red or green. In order to do so, we must examine the way in which different colored dots are displayed on the ST's color screen.

On the monochrome ST, the display system is very simple. Each dot on the screen is represented by a single binary digit (bit) of memory. Screen memory is organized in such a way that the first byte represents the 8 dots in the top left corner of the screen, and each succeeding byte represents the next 8 dots to the right. Since each line contains 640 dots across, the first 80 bytes fill up the top line, and the next byte is used to represent the first 8 dots on the second line. There are 400 lines of 80 bytes each on the monochrome screen, which means that 32,000 bytes of screen memory are used to represent the 256,000 dots on screen. (See Figure 4-1.)

Figure 4-1. Monochrome Screen Memory

01110101
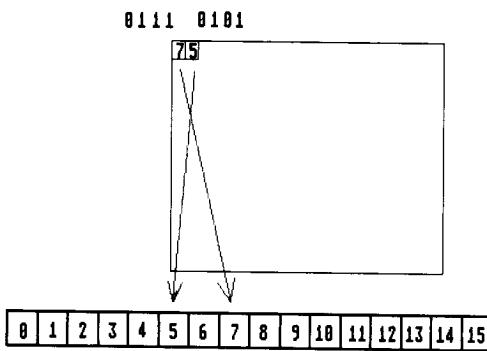
Each bit of screen memory can hold either the number 0 or 1. On a monochrome system, only one bit is needed to represent a screen dot or pixel (picture element), because each dot on the screen is either white (off) or black (on). But with a color ST system, things are somewhat different. In medium-res mode, any dot can be one of four colors. Two binary digits are used to yield four possible combinations:

00 = 0
01 = 1
10 = 2
11 = 3

In lo-res mode, any dot can be 1 of 16 colors, so four bits are required to describe a single pixel. Color screen memory is organized in much the same way as monochrome screen memory, except, instead of single bits, groups of bits are used to represent each screen dot. Thus, in medium resolution the first byte of screen memory is used to depict the four pixels at the top left corner of the screen. The two high-order bits specify the color in the first dot, the next two the color in the following dot, and so on. Since there are 640 dots per row, each row requires 160 bytes of screen memory. But since there are only 200 rows of dots, 32,000 bytes of screen memory are still sufficient to display all of the dots on the screen. In low resolution, each byte describes only two dots of color. There are only 320 dots per line in this mode, however, so 160 bytes describe all of the dots in each line in this mode also. (See Figure 4-2.)

Figure 4-2. Low-Resolution Screen Memory

0111 0101

Hardware color registers

In monochrome mode, each bit pattern can represent a specific color, because there are only a total of two colors available. But the ST is capable of displaying 512 different colors on an RGB color monitor or television set. Clearly, in color mode each set of bits cannot represent a particular color, using a code where 0 represents white, 1 stands for black, and so on. Instead, the number stored in the memory location that corresponds to a screen dot location refers to a hardware color register.

Color Registers

The color registers may be thought of as a set of 16 pens, each of which may be filled with ink that is colored in any of the 512 shades supported by the ST. Register 0 always holds the color we think of as the background color (which defaults to white on the ST). When you wish to use another color to draw a line or a point, you specify the pen (color register) that will be used to draw it. Whatever color "ink" it currently contains is the color that the pen will draw onscreen.

Unlike ink, however, the color of a dot that you have drawn onscreen can change after you have drawn it. When the display memory for a screen dot holds the number of a particular pen, that dot is displayed in whatever color is in the pen at any given moment, not in the color that was in the pen at the time the dot was drawn. This means that if you use pen 1 to draw a line, and that pen contains the default color black, the line will be black. But if you change the color in pen 1 to green after you've drawn the line, the line you drew and everything else on the screen that was drawn with pen 1 will instantly become green.

The two factors that determine which colors are assigned to the figures that you draw on the screen, therefore, are the pen you used for the drawing and the color currently contained in that pen.

As we've seen above, you can choose a different pen for drawing markers and lines by using the vsm_color and vsl_color calls. And we will see later that you may also select another pen for graphic text with the vst_color call, and one for filled shapes with vsf_color.

In BASIC, all but the marker pen are set with the same command, COLOR. If you do not specify a pen before drawing, you'll get the default drawing pen that is specified in the

work_in array at the time that the virtual screen workstation was opened (usually color 1, which has a default value of black). You should be aware that the GEM VDI drawing pens (referred to in the GEM literature as the color index) do *not* correspond numerically to the ST color registers. GEM uses a complex scheme for mapping drawing pens to hardware registers, so that drawing pen 1 corresponds to color register 15, pens 3, 4, and 5 correspond to registers 2, 4, and 6; pens 6, 7, and 8 correspond to registers 3, 5, and 7, and so forth. GEM uses a lookup table to match color index values to hardware registers. The complete correspondence is mapped out in Table 4-2.

In addition to determining which color register will be used for drawing, we must also determine the color that the register contains. Colors are chosen by mixing various levels of the colors red, green, and blue. Each color register holds one of eight color levels for each of these colors, which means that there are 512 ($8 \times 8 \times 8$) possible colors to choose from.

The VDI call to set a color register to a particular shade is Set Color Representation. The C language syntax for this call is

int handle, pen, rgb[3];
vs_color(handle, pen, rgb);

where *pen* is the number of the drawing pen whose color you wish to change, and *rgb* is a pointer to an array holding color levels for red, green, and blue. The first element of this array (rgb[0]) holds the red value; the second holds the green value; and the third holds the blue value. Since GEM is written to be non–computer-specific, these color values are expressed in tenths of a percent of color saturation, meaning the color level values range from 0 to 1000. With only eight color levels supported by the hardware, it should be obvious that many rgb values will display in the same color. Table 4-1 shows the relationships between the color value that you request (with the vs_color call), the actual value that is set, and the hardware color register level to which that value corresponds.

Table 4-1. Color Values and Register Values

Requested Value	Actual Value	Hardware Register Color Level
0–70	0	0
71–213	142	1
214–356	285	2
357–499	428	3
500–642	571	4
643–785	714	5
786–928	857	6
929 and up	1000	7

Since there are 512 possible combinations, it's nearly impossible to describe each one or to explain exactly how to find a particular shade. In general, however, the higher the color level, the brighter the color; the lower the level, the darker the color. Whether the color displayed by a register tends toward red, green, or blue depends on which value has the highest brightness level. If all three values are equal, the color will be black, white, or a shade of gray.

Mixing Colors

Thus, if rgb contains all zeros, the pen will be set to black, while a setting of straight 1000s will set it to white. You can lighten a shade by increasing the value of the two other colors in equal proportions. A setting of 1000,0,0 selects bright red as the pen color, while 1000,428,428 sets a much paler red. To darken the original red color, you can lower the red setting to 714 while keeping the other two at 0.

When you're unsure of what colors to mix, it may help to start with the nearest primary color mixture and experiment from there. These are the red, green, and blue values for these mixtures:

Color	Red	Green	Blue
Black	0	0	0
Blue	0	0	1000
Green	0	1000	0
Cyan	0	1000	1000
Red	1000	0	0
Purple	1000	0	1000
Yellow	1000	1000	0
White	1000	1000	1000

You can use the Control Panel desk accessory to get instant feedback on what color levels to use for a particular color. When you use it to mix colors using different levels of red, green, and blue, the panel displays these color levels as numbers from 0 to 7. Using Table 4-1, you can translate these hardware levels into the corresponding numbers used by the VDI.

If you do not change the colors of any of the color registers, the default VDI color palette will be used. Table 4-2 gives the default values for each of the VDI color pens. Next to each pen number in square brackets is the number of the corresponding hardware color register. In square brackets, next to the VDI red, green, and blue color values for each of the registers, are the corresponding hardware color levels. The table illustrates the 16-color low-resolution mode, but the 4-color mode is similar, with the exception that pen 1 maps to color register 3 in medium-resolution mode.

Table 4-2. Default Values of VDI Color Pens

Pen	[Reg]	Red	Green	Blue	Color
0	[0]	1000 [7]	1000 [7]	1000 [7]	White
1	[15]	0 [0]	0 [0]	0 [0]	Black
2	[1]	1000 [7]	0 [0]	0 [0]	Red
3	[2]	0 [0]	1000 [7]	0 [0]	Green
4	[4]	0 [0]	0 [0]	1000 [7]	Blue
5	[6]	0 [0]	1000 [7]	1000 [7]	Cyan
6	[3]	1000 [7]	1000 [7]	0 [0]	Yellow
7	[5]	1000 [7]	0 [0]	1000 [7]	Magenta
8	[7]	714 [5]	714 [5]	714 [5]	Low White
9	[8]	428 [3]	428 [3]	428 [3]	Gray
10	[9]	1000 [7]	428 [3]	428 [3]	Light Red
11	[10]	428 [3]	1000 [7]	428 [3]	Light Green
12	[12]	428 [3]	428 [3]	1000 [7]	Light Blue
13	[14]	428 [3]	1000 [7]	1000 [7]	Light Cyan
14	[11]	1000 [7]	1000 [7]	428 [3]	Light Yellow
15	[13]	1000 [7]	428 [3]	1000 [7]	Light Magenta

The VDI pen numbers are followed by the corresponding hardware color register numbers, in square brackets. The VDI color levels are followed by the corresponding hardware register color levels, shown in square brackets.

Locating Color Information

It is often useful to be able to find out which color is contained in a particular color register. For one thing, GEM does not reset the color palette when your application ends and the

user returns to the Desktop. So, in order to restore the user's color preferences when you end your application, you've got to have some way of knowing what those settings were when your program started. The VDI call used to learn the settings for a particular register is Inquire Color Representation, whose C language format is

int handle, pen, flag, rgb[3];
vq_color(handle, pen, flag, rgb);

where *pen* is the color register number, and *rgb* is an array where the red, green, and blue color levels will be returned (in elements 0, 1, and 2, respectively). The *flag* setting allows you to select whether you wish to learn the setting that was requested when the vs_color call was made or the actual value that was set.

As we stated above, there are 1000 different VDI settings, but only eight possible hardware settings, so a wide range of VDI settings correspond to the same hardware setting. For example, if you set all the values in rgb[0]–[3] to 650 and call vs_color, then a color level of 714 (corresponding to a hardware level of 5) will be set for each. If you request a value of 700 or 750, you'll still get a setting of 714. The flag setting determines whether the color level values that you get are the values that were requested or the values that were actually set. If you set the flag to 1 before calling vq_color, you get the actual color settings (714 in each case). But if you set the flag to 0, you get the value that was requested, not the one that was set (650, 700, or 750). You should also note that if an invalid pen number is requested, a −1 is returned in rgb[0].

Another handy bit of information to have is the pen color used to draw a given dot on the screen. The function Get Pixel returns not only that information, but the color register that corresponds to that pen setting as well. The C language version of this call is

int handle, x, y, pixel, pen;
v_get_pixel(handle, x, y, &pixel, &pen);

where *x* and *y* are the coordinates of the point, *pixel* is the variable in which the hardware color register number is returned, and *pen* is the variable in which the pen number is returned.

CHAPTER 4

Program 4-1 shows how to change the color that is contained in each of the drawing pens. This program works correctly only on a color monitor, since the principles it illustrates are not applicable to the monochrome screen.

In Program 4-1 each of the bars initially appears in the color gray; each set of three bars, though, is drawn using a different drawing pen. When we change the color of each of the pens in turn, the colored line appears to move. Note how we saved the initial color values before we changed the pen colors, and restored them after we were done. This insures that when the program ends, you will find the desktop colors just as you set them.

Program 4-1. color1.c

```
/*******************************************/
/*                                        */
/*                                        */
/*   Color1.C -- Demonstrates drawing in  */
/*   different pen colors, and changing the */
/*   colors contained in those pens       */
/*                                        */
/*                                        */
/*******************************************/

#include "shell.c"

#define REPS 3    /* number of wide lines to draw */

demo()
{
int index=0,
    b,c,
    xmax,dx,mid,
    points[4],
    rgb[3],
    newcol[3],
    pal[3][3];
long d;

    rgb[0]=rgb[1]=rgb[2]=571; /* grey values */

 /* Save original color palette */
    for (c=0;c<3;c++)vq_color(handle,c+1,1,pal+c);

    xmax = work_out[0];
    dx = xmax/20;        /* scale lines to horizontal resolution */
    mid = xmax/=2;       /* find midpoint of screen */

    points[3] = work_out[1]- 12;
    points[1] = 12;   /* set top and bottom of lines */

    vsl_width(handle, dx ); /* change width */

    for (c=0;c<REPS;c++)   /* draw horizontal lines line */
    {
    vsl_color(handle,c+1);        /* change pens */
    vs_color(handle,c+1,rgb);     /* set pen color to grey */
```

```
    points[0] = points[2] = (xmax-=(dx*2)); /* set points */
    v_pline(handle, 2, points);   /* draw it */
    points[0] = points[2] = (mid+=(dx*2)); /* set points */
    v_pline(handle, 2, points);   /* draw it */
    }

for (b=0;b<7;b++) /* for each primary color mixture... */
    {

    newcol[0] = 1000 * (b&1);
    newcol[1] =  500 * (b&2);
    newcol[2] =  250 * (b&4);

    for (c=0;c<3;c++) /* cycle color through 3 sets of bars */
        {
        if (c%3==0) vs_color(handle,3,rgb);
        else vs_color(handle,c%3,rgb); /* old bar back to grey */

        vs_color(handle,c%3+1,newcol);   /* set new bar */

        for (d=0;d<80000;d++); /* waste some time */
        }
    }
/* Restore original color pallete */
    for (c=0;c<3;c++)vs_color(handle,c+1,pal+c);

}

/* End of Color1.c */
```

Drawing Modes

We've seen that when we draw a dot of color on the screen,
what actually happens is that the screen memory representing
that dot is changed to reflect the number of the color register
that produces that color. In effect, the new drawing replaces
whatever had previously appeared in that spot on the screen.

It's also possible for drawing operations to interact with
existing screen graphics, rather than to replace them. For ex-
ample, a dotted line is partly made up of 1 bits (the graphics
object), and partly of 0 bits (the color mask). The line part is
normally drawn in whatever color is in the line drawing pen.
But how are the spaces between the lines drawn? Will they be
drawn using the background color, or will they not be drawn
at all, so that whatever display was on the screen before the
line was drawn will show through?

The VDI allows you to select from four different drawing
modes that determine how the graphics object (the 1 bits in
the pattern) and the color mask (the 0 bits in the pattern) will
affect the existing display. The drawing mode (or writing
mode, as it is sometimes called) is significant because many
GEM graphics types are made up of bit patterns containing
both 0 and 1 bits. Patterned lines, large markers, filled shapes,

and graphics text all consist of graphics patterns that are partly colored images and partly space surrounding those images. The VDI writing mode affects all of these different types of graphics renderings. Once you set a new writing mode, it stays in effect until you explicitly change it again.

Replace mode. The default writing mode is called the Replace mode. In Replace mode, the part of the image that consists of 1 bits is drawn with whatever color is in the relevant drawing pen (the line drawing pen, the marker pen, the text pen, or the fill pen). The part of the image that consists of 0 bits (the color mask) is drawn in the background color, found in pen 0. As its name suggests, Replace mode replaces whatever color was already there with the drawing color and the background color.

Transparent mode. The second mode is called Transparent mode. As with Replace mode, drawings that are made in this mode depict the graphics object (1 bits) in the color of the current drawing pen. But Transparent mode drawings leave the color mask portion (0 bits) alone, so that whatever color was there previously still shows through in the blank spaces around the image. Patterned images drawn in Transparent mode look different from those drawn in Replace mode, with the former looking as if they had been stenciled onto the existing image. Solid images look the same when drawn in either mode, however, since they are made up entirely of 1 bits.

Reverse Transparent mode. The opposite of Transparent mode is Reverse Transparent. In this mode, only the color mask portion (made up of 0 bits) is drawn, using the current drawing pen. The part of the screen which corresponds to the object portion of the image (the 1 bits) is left alone. Reverse Transparent mode can be used to draw graphics text in *inverse video*, where the space surrounding the letters, rather than the letters themselves, are rendered in the color of the current text drawing pen.

XOR mode. In the final drawing mode, neither the current foreground drawing pen nor the background pen (pen 0) is used to color the object or its color mask. This mode is known as XOR mode. It's name comes from the logical operation eXclusive OR, by which the colors on screen are complemented.

To complement the color of a pixel, you invert the bits of its color register number, changing all of the ones to zeros, and all of the zeros to ones. For example, if a dot was drawn

with the color in register 3, and you were in 16-color mode, the binary representation for that dot would be 0011. The complement of that number would be 1100, or 12 decimal. Therefore, when drawing in XOR mode, every time the object part of the image (the 1 bits) coincided with a portion on the screen that had been drawn with color register 3, that part of the display would be changed to the color in register 12.

For those of you who don't normally think in binary numbers, another way of looking at the process is to take the highest possible color register number and subtract the color register number of the existing color. What you're left with is the register number of the new color. In the above example, the highest number is 15 (since the 16-color mode counts from 0 to 15). If you subtract 3 from 15, you are left with 12. If you're using the 4-color mode, 3 would be the highest register number, so 3 minus 3 would leave you with color register 0.

Remember, the XOR mode complements the hardware color register number, *not* the VDI drawing pen number. You may use Table 4-2 to match the VDI pen numbers to their hardware register equivalents, which appear next to them in square brackets.

Like Transparent mode, XOR mode only affects the area of the screen display corresponding to the 1 bits of the object image. But XOR mode has some unique properties all its own. For example, if you use the Transparent mode to draw a green line on a portion of the screen that is already colored green, your line changes nothing, and it will not show up at all. By its very nature, however, a line drawn in XOR will always show up, since it changes whatever was on screen to another color.

Another interesting property of XOR mode is that while using it once always changes the picture, using it twice in a row restores the original colors. This makes XOR mode handy for drawing lines that will have to be erased later. It also lends itself to use in animation, where the background must be restored after the object is moved.

Set Writing Mode. The VDI function call that is used to set one of these drawing modes is called Set Writing Mode. The C format for this function is

```
int handle, mode, mode_set
mode_set = vswr_mode(handle, mode);
```

where *mode* is the drawing mode that you're requesting, and *mode_set* contains the number of the drawing mode that was actually set. The numbers assigned to the different writing modes are

Number	Mode
1	Replace
2	Transparent
3	XOR
4	Reverse Transparent

Drawmode Demonstration

Program 4-2 demonstrates the different writing modes by drawing dotted lines and text in each of the four modes.

In order to show the full effect of the different modes, half of the background screen is left as background color, and half is changed to the color in pen 3 (green). On that background, we draw black patterned lines using line pattern 5 (dashed), and graphics text. Though the text commands will not be covered until a later chapter, we included text in this example because it clearly illustrates the differences between the drawing modes.

In Replace mode, the line appears in black (the drawing pen color) and white (the background color). The Replace mode text appears as black letters on a white background.

In Transparent mode, the line appears in black, but the spaces between the lines are left alone, so they appear in white on the white background and green in that part of the screen. Only the black letters of the text are drawn.

In XOR mode, the part of the line that is drawn in black in the other two modes is drawn in the complement of whatever color it's drawn on. The complement of white is always black, but the complement of green in lo-res mode is different from what it is in medium-res mode. In medium-res, red (color register 1) is the complement of green (color register 2). But in lo-res, light cyan (color register 13) is its complement. As in transparent mode, the background is left alone in the spaces between the image patterns, or around the letters. Finally, in Reverse Transparent mode, the spaces that were ordinarily blank in the dotted line are drawn in, in black, while the line itself is left in the background color. Similarly, the

"frame" around the letters is colored in, in black, while the letter shapes themselves are filled with whatever color happened to be there already.

Program 4-2. drawmode.c

```
/*******************************************************/
/*                                                   */
/*                                                   */
/*  Drawmode.C -- Demonstrates drawing               */
/*    mode for patterned lines and graphics          */
/*    text.                                          */
/*                                                   */
/*                                                   */
/*******************************************************/

#include "shell.c"

demo()
{
int c,
    xmax,ymax,dx,dy,
    points[4];


    xmax = work_out[0];
    ymax = work_out[1];
    dx = xmax/20; /* set x offset as fraction of screen width */
    dy = ymax/20; /* set y offset as fraction of screen height */
    points[3] = (ymax -= 12);
    points[1] = 12;  /* y values of lines */

    vsl_width(handle, dx );    /* use wide lines */
    vsl_color(handle,3);       /* lines are green */

    for (c=0;c<12;c++)   /* draw a block of green lines */
    {
    points[0] = points[2] = (xmax-=dx); /* set points */
    v_pline(handle, 2, points);   /* draw it */
    }

    vsl_type(handle,5);  /* dotted line */
    vsl_width(handle,1); /* of normal width */
    vsl_color(handle,2); /* color is red */
    points[0] = 16;      /* x for line left */
    points[2] = work_out[0]-16; /* for line right */

    for (c=1;c<5;c++)   /* draw lines and text in each mode */
    {
     points[1] = points[3] = (ymax -= dy); /* set points */
     vswr_mode(handle,c);
     v_pline(handle, 2, points);    /* draw it */

    switch(c) /* pick appropriate text, and print it */
       {
       case 1:
         v_gtext(handle, work_out[0]/7, (ymax-=dy),
         "This is the Replace drawing mode");
         break;
       case 2:
         v_gtext(handle, work_out[0]/7, (ymax-=dy),
         "This is the Transparent mode");
         break;
       case 3:
         v_gtext(handle, work_out[0]/7, (ymax-=dy),
```

```
        "This is the XOR drawing mode");
    break;
    case 4:
      v_gtext(handle, work_out[0]/7, (ymax-=dy),
      "This is Reverse Transparent mode");
    }
    ymax-=(2*dy); /* space between lines */
    }
}

/* End of Drawmode.c */
```

Program 4-3 is the same Drawmode program written in assembly language.

Program 4-3. drawmode.s

```
***********************************************
*                                             *
*                                             *
*      DRAWMODE.S -- assembly version of      *
*      drawing mode program                   *
*                                             *
*                                             *
*                                             *
***********************************************


.xdef demo
.xref vwkhnd
.xref contrl0
.xref contrl1
.xref contrl2
.xref contrl3
.xref contrl4
.xref contrl5
.xref contrl6
.xref contrl7
.xref contrl8
.xref contrl9
.xref contrl10
.xref contrl11
.xref intin
.xref intout
.xref ptsin
.xref ptsout

.text

demo:

    move      intout,xmax
    move      intout+2,ymax
    move      dx,d0
    move      dy,d1
    cmp       #639,xmax       * if high-res or med-res
    bne       skip1
    add       d0,d0           * double dx
    move      d0,dx

skip1:
    cmp       #399,ymax       * if high-res
    bne       skip2
    add       d1,d1
    move      d1,dy           * double dy
```

```
skip2:

*** Set line width
    move      #16,contrl0    * opcode for line width
    move      #1,contrl1     * 1 point in ptsin
    move      #0,contrl3     * no integer parameters in intin

    move      dx,ptsin
    move      #0,ptsin+2     * set width to dx
    jsr       vdi


*** Set line color
    move      #17,contrl0    * opcode for line color
    move      #0,contrl1     * no points in ptsin
    move      #1,contrl3     * 1 integer parameter in intin

    move      #3,intin       * line color green
    jsr       vdi


*** set points ***
    move      #12,ptsin+2    * set top and bottom of block
    move      ymax,d0
    sub       #12,d0
    move      d0,ptsin+6
    move      d0,ymax
    move      xmax,d5

    move      #11,d4         * loop counter
block:
    sub       dx,d5          * decrement xposition of line
    move      d5,ptsin
    move      d5,ptsin+4

*** Draw the lines
    move      #6,contrl0     *opcode for polyline
    move      #2,contrl1     *number of points in ptsin
    move      #0,contrl3     * no integer parameters in intin

    jsr       vdi
    dbra      d4,block


*** Set line type
    move      #15,contrl0    * opcode for line pattern
    move      #0,contrl1     * no points in ptsin
    move      #1,contrl3     * 1 integer parameter in intin

    move      #5,intin       * line pattern 5 -- dotted line
    jsr       vdi


*** Set line color
    move      #17,contrl0    * opcode for line color

    move      #2,intin       * line color is red
    jsr       vdi

*** Set line width
    move      #16,contrl0    * opcode for line width
    move      #1,contrl1     * 1 point in ptsin
    move      #0,contrl3     * no integer parameters in intin

    move      #1,ptsin
    move      #0,ptsin+2     * set width to 1
    jsr       vdi
```

```
*** set points ***

    move     xmax,d0      * set left and right for dotted line
    move     d0,d6
    divu     #7,d6
    sub      #16,d0
    move     d0,ptsin+4
    move     ymax,d5


    move     #3,d4        * loop counter
modes:
*** Change drawing modes ***
    move     #32,contrl0  * Opcode for set writing mode
    move     #0,contrl1
    move     #1,contrl3
    move     d4,d0
    addq     #1,d0
    move     d0,intin
    jsr      vdi

    move     #16,ptsin
    sub      dy,d5        * decrement y of dotted line
    sub      dy,d5
    move     d5,ptsin+2
    move     d5,ptsin+6

*** Draw dotted lines
    move     #6,contrl0   *opcode for polyline
    move     #2,contrl1   *number of points in ptsin
    move     #0,contrl3   * no integer parameters in intin
    jsr      vdi

*** Print graphics text
    move     #8,contrl0   * opcode for gtext
    move     #1,contrl1   * 1 point in ptsin
    move     #33,contrl3  * 33 characters in string, including null
    sub      dy,d5
    move     d5,ptsin+2   * Set position for text
    move     d6,ptsin

    move     #32,d0       * 33 characters
    movea.l  #intin,a1    * address of destination
    movea.l  #t1,a0       * calc address of source string
    move     d4,d1
    mulu     #33,d1
    add      d1,a0
text:
    clr.w    d1
    move.b   (a0)+,d1     * move a letter from source...
    move.w   d1,(a1)+     * to word-aligned destination...
    dbra     d0,text      * until all done.

    jsr      vdi          * print text

    dbra     d4,modes     * next drawing mode
    rts


**** data section

    .data
    .even
```

```
dx:          .dc.w    16
dy:          .dc.w    10

*32 characters
t1:          .dc.b    'This is the Replace Drawing Mode',0
t2:          .dc.b    'This is the Transparent Mode    ',0
t3:          .dc.b    'This is the XOR Drawing Mode     ',0
t4           .dc.b    'This is Reverse Transparent      ',0

     .bss
xmax:        .ds.w    1
ymax         .ds.w    1

.end
```

Clipping

Another function that affects all types of graphics output is known as *clipping*. Clipping is used to confine graphics output to a designated rectangular portion of the screen. If part of the graphics output you're trying to draw lies inside of the clipping rectangle and part lies outside the rectangle, the part that is inside will be drawn, while the part that is outside won't.

By setting a clipping rectangle that is as large as the entire display, you can insure that no part of your graphics output will be "drawn" on memory that does not belong to the screen display. This can prevent nasty system crashes, since when drawing operations affect program memory, unpredictable (and usually unpleasant) results occur. Clipping is also extremely helpful for updating GEM windows. Not only can it insure that you confine your drawing to the interior of the window, it can also enable you to redraw only the portion of the window that has been uncovered after having been covered by another window.

Like all good things, clipping has its price. When clipping is on, the VDI must examine every point before it's drawn, in order to make sure that it lies within the rectangle. This extra burden can slow down graphics output. For this reason, when a workstation is first opened, clipping is turned off. To turn it on, you must use the Set Clipping Rectangle function, the C version of which looks like this:

int handle, flag, points[4];
vs_clip(handle, flag, points);

where *flag* is used to indicate whether you want to turn clipping on (flag = 1 or greater) or to turn clipping off (flag = 0). *Points* is a pointer to an array that holds the *x* and *y* coordinates of the four sides of the clipping rectangle. Points[0]

holds the coordinate of the left side, points[1] the top, points[2] the right side, and points[3] the bottom.

The sample program (Program 4-4) demonstrates the use of clipping. It draws a series of concentric circles, the first of which lies within the clipping rectangle. The remaining circles are clipped at the top and bottom.

Program 4-4. clip.c

```
/******************************************************/
/*                                                  */
/*                                                  */
/*    CLIP.C -- Demonstrates use of the             */
/*    clipping rectangle.                           */
/*                                                  */
/*                                                  */
/******************************************************/

#include "shell.c"

#define STEP 10

demo()
{
int c,
    dx,dy,
    cy,r,
    points[4];

    if (work_out[0] == 639)dx=STEP; /* set full horiz step */
        else dx = STEP/2;           /* except for lo-res */
    if (work_out[1] == 399)dy=STEP; /* set full vert step */
        else dy =  STEP/2;          /* except for color */

      vsl_width(handle,dx); /* set wide line */

      cy = 20*dy;             /* set a clipping rectangle */
      r = 2 *(dx+dy);         /* that's as wide as the screen */
      points[0]=0;            /* but not very tall */
      points[2]=work_out[0];
      points[1]=cy-r;
      points[3]=cy+r;

      vs_clip(handle,1,points); /* draw concentric circles */

      for (c=1;c<14;c++)
          {
          vsl_color(handle,c); /* change drawing pen */
          v_arc(handle,32*dx,20*dy,(c+1)*(dx+dy),0,3600); /* draw 'em */
          }

}

/* End of Clip.c */
```

NDC Example

So far, all of our sample programs have used the ST's own Raster Coordinate system. But as we saw in Chapter 2, the GEM VDI also supports a non–device-specific format called Normalized Device Coordinates. The type of coordinate system you choose, Raster or Normalized, is another setting decision that affects all subsequent drawing functions. Sample Program 4-5 opens two workstations, one using Raster Coordinates and the other using Normalized Device Coordinates. It then draws a box containing a circle, using each type of coordinate system. Note that several workstations can be open at once, each with its own set of graphics settings.

By looking at the functions showrc and shownorm, we can examine some of the differences between the two coordinate systems. In the raster coordinate system, we have to scale the horizontal and vertical coordinates for the center point of the circle, and the radius value, according to the maximum screen resolution.

In NDC mode, we use the same fixed values regardless of the screen resolution, and the VDI does the scaling for us. Notice how we must scale the vertical height of the box according to the aspect ratio of each pixel. That's because the VDI scales the circle in order to make it appear round. We use the values in work_out[3]–[4] to find the aspect ratio. Interestingly enough, we not only have to scale the vertical dimension of the NDC box according to the aspect ratio of each pixel, but we also have to scale it according to the aspect ratio of the screen. That's because in NDC mode, the VDI not only compensates for the fact that each pixel may not be as wide as it is tall, but also for the fact that there may not be an even number of rows and columns. Another point worth mentioning is that we did not have to change the line drawing pen to get the second figure to appear in black, since the NDC workstation uses its own line drawing pen which is separate from the one used by the RC workstation. This program also demonstrates the rounding error that can occur when we use Normalized coordinates. In the color modes, the circle on the right extends one dot past the border of the box.

As discussed earlier Program 4-5 requires that GDOS be in the AUTO folder of the disk used when starting your system.

CHAPTER 4

Program 4-5. ndc.c

```
/*********************************************/
/*                                         */
/*                                         */
/*    NDC.C -- Demonstrates use of the     */
/*    normalized coordinate system         */
/*    along with raster coordinates        */
/*                                         */
/*                                         */
/*********************************************/

#include "shell.c"

#define STEP 10
int dx, dy,handle1,points[14];

demo()
{

    if (work_out[0] == 639)dx=STEP; /* set full horiz step */
        else dx = STEP/2;           /* except for lo-res */
    if (work_out[1] == 399)dy=STEP; /* set full vert step */
        else dy =  STEP/2;          /* except for color */

  showrc();    /* do rc circle demo */
  showndc();   /* do ndc demo */
    }

showrc()
{
  int cx,cy,r;
  long r1;
    cx = 16*dx; /* horiz coordinate of circle center */
    cy = 20*dy; /* vertical coordinate of center */
    r = r1 = 12 * dx; /* circle radius */
    vsl_width(handle,dx); /* set wide line */
    vsl_color(handle,2); /* change color */
    v_arc(handle,cx,cy,r,0,3600); /* draw circle */

    points[0]=points[6]=points[8]=cx-r; /* x for box left */
    points[2]=points[4]=points[10]=cx+r; /* x for box right */
    r1 = (r1*work_out[3]/work_out[4]); /* scale box height */
    points[5]=points[7]=points[11]=cy+r1; /* y for box bottom */
    points[1]=points[3]=points[9]=cy-r1;  /* y for box top */
    v_pline(handle,6,points); /* draw box */
}


showndc()
{
int nul,x;

/* Initialize input array, get the physical workstation handle,
   and open the Virtual Screen Workstation with normalized coords*/

    for (x= work_in[10]=0; x<10; work_in[x++]=1);
    handle1 = graf_handle(&nul, &nul, &nul, &nul);
    v_opnvwk (work_in, &handle1, work_out);

/* perform the graphics demos */

    shownorm();
```

```
/* close the NDC virtual screen workstation */

v_clsvwk(handle1);
}

shownorm()
{
  int cx,cy,r;
  long r1;
    cx = 24575; /* x for center */
    cy = 16383; /* y for center */
    r = r1 = 6144; /* radius */

    vsl_width(handle1,512); /* set wide line */
    v_arc(handle1,cx,cy,r,0,3600); /* draw circle */

    points[0]=points[6]=points[8]=cx-r; /* x for box left */
    points[2]=points[4]=points[10]=cx+r;/* x for box right */
    r1 = (r1*dx*64)/(dy*40);            /* scale for width/height */
    r1 = (r1*work_out[3]/work_out[4]);  /* scale for aspect ratio */
    points[5]=points[7]=points[11]=cy-r1; /* y for box bottom */
    points[1]=points[3]=points[9]=cy+r1;  /* y for box top */
    v_pline(handle1,6,points);
}


/* End of NDC.c */
```

BASIC Graphics Settings

None of the generalized graphics settings that we have been talking about in this chapter have keyword support in the first version of ST BASIC. Although the revised version has not appeared at the time of this writing, there are indications that this version will include the command DRAWMODE to set the drawing mode. The form for this command is

DRAWMODE mode

where *mode* is the mode number from 1 to 4 (these correspond to the mode numbers used by vswr_mode). It may also include the commands RGB to set the color registers, and ASK RGB to read them. The syntax for these are

RGB register, red, green, blue
ASK RGB register, red, green, blue

where *register* is the color register number, and *red, green*, and *blue* are either the values for the new settings (RGB) or variables to hold the existing settings (ASK RGB). Note that these settings reflect the hardware registers, not the VDI pens. This means that the register numbers will differ from the pen numbers used in the COLOR command, and that the red, green, and blue values will be in the range 0–7, not 0–1000.

Even without these new commands, however, it is still possible to use the VDI setting commands with the old POKE, VDISYS(0) method. Program 4-6 demonstrates the BASIC translation of the Drawmode program.

Program 4-6. drawmode.bas

```
10      fullw 2: clearw 2
20      res = peek(systab)
30      if (res<4) then xmax = 639 else xmax = 319
40      if (res>1) then ymax = 199 else ymax = 399
50      dx = xmax/20: dy = ymax/20
52      REM Initialize text strings
54      text$(1)="This is the Replace Drawing Mode"
55      text$(2)="This is the Transparent Mode"
56      text$(3)="This is the XOR Drawing Mode"
57      text$(4)="This is Reverse Transparent     "
60      REM Set Line Width
70      poke contrl,16
80      poke contrl+2,1
90      poke contrl+6,0
100     poke ptsin,dx: REM width = dx
110     poke ptsin+2,0
120     vdisys(0)
130     REM Set line color
140     color 1,0,3
150     REM
160     for x = 3 to 14
170     linef xmax-(x#dx),12,xmax-(x#dx),ymax-(4#dy)
180     next x
190     REM Set line drawing pattern
200     poke contrl,15
210     poke contrl+2,0
220     poke contrl+6,1
230     poke intin,5
240     vdisys(1)
250     REM Set drawing color
260     color 1,0,2
270     REM Set Line Width
280     poke contrl,16
290     poke contrl+2,1
300     poke contrl+6,0
310     poke ptsin,1: REM width = 1
320     poke ptsin+2,0
330     vdisys(0)
340     REM
350     for x=1 to 4
360     REM change drawing modes
370     poke contrl,32
380     poke contrl+2,0
390     poke contrl+6,1
400     poke intin,x
410     vdisys(0)
420     REM
430     linef 16,ymax-((x+1)#3#dy) ,xmax-(2#dx),ymax-((x+1)#3#dy)
440     gotoxy (xmax/319)#5 ,18-(x#4)
450     print text$(x)
460     next x
```

Chapter 5

Filled Shapes

In addition to line drawing routines that create

the outlines of a figure, the GEM VDI also provides a group of output routines that create shapes whose interiors are filled with a solid color or with a pattern of colors. Patterned fills provide a means of distinguishing the interior of one shape from another on monochrome systems. For example, if you're drawing a pie chart on a monchrome screen, all of the wedges will look the same if you try to fill them with different solid colors. By filling them with different crosshatch patterns you can make them visually distinct on both monochrome and color systems. Like the line drawing routines, the fill routines share a number of common graphics settings that can be used to select the color, the fill pattern, and whether or not the figure is outlined in a solid color.

Filled Rectangle
The simplest of the filled figures is the rectangle. This shape is created with the VDI command Fill Rectangle, whose function is to quickly fill a rectangular area on the screen. (This command may not work with other output devices.) The C language version of this function is

```
int handle, sides[4];
vr_recfl(handle, sides);
```

where *sides* is an array that contains the coordinates for each side of the box. The values for the left and right sides are held in sides[0] and sides[2], and sides[1] and sides[3] contain the location of the top and bottom.

Pattern Fills
Although drawing a filled box may seem a very straightforward operation, the VDI provides a number of fill settings that allow you to vary the results significantly. The first of these are the fill pattern settings. GEM provides the ST screen display device with five different general types of fill patterns, which are referred to in the GEM literature as fill interior styles.

The Hollow fill pattern fills the interior of the figure with the current background color. The Solid style fills the shape with the currently selected fill color. The Pattern style superimposes one of a number of different drawing patterns on the fill area. These include dot patterns of varying density, horizontal and diagonal checkerboards, and herringbone patterns. The Hatch pattern fills the area with one of a number of different crosshatch patterns. These are made up of horizontal, vertical, or diagonal lines, either alone or in combination. Finally, the user-defined style allows you to display a pattern that you create yourself, using an array of 16 words to represent a 16 X 16 pattern of dots.

When you open your virtual workstation, you specify a default pattern type in the variable work_in[7]. (We've been setting it to 1, Solid.) To change the pattern type from this default, you must choose another with the command Set Fill Interior Style. The syntax for this call is

int handle, pattern_type
type_set = vsf_interior(handle, pattern_type);

where *pattern_type* is a number that corresponds to one of the five fill types:

0	Hollow
1	Solid
2	Pattern
3	Hatch
4	User-Defined

The number of the pattern type that the VDI actually sets is returned in the variable type_set. If you choose a pattern_type that isn't available, the type will be set to 0, Hollow.

As we mentioned above, two of these pattern types contain a number of different patterns with similar characterstics. The Pattern style is made up of 24 different dot patterns, and the Hatch type has 12 different type of crosshatch line patterns. Whenever you choose the Pattern or Hatch styles, the actual fill pattern that is used will be determined by the setting of the Fill Style Index, that selects one of these subpatterns. When you open the virtual screen workstation, you designate the default value for this index in the variable work_in[8] (provided that the GDOS extension is loaded). Thereafter, you can select a new subpattern with the function Set Fill Style Index, which, in C, looks like this:

```
int handle, pattern_index;
index_set = vsf_style(handle, pattern_index);
```

where *pattern_index* is the index number for the subpattern.
For Hatch patterns on the ST screen, index numbers from 1 to
12 produce different crosshatch designs, and for Pattern type
fills, index numbers from 1 to 24 produce unique results. The
number of the index that was actually set by the VDI is re-
turned in the variable *index_set*. If the index requested is not
available, an index of 1 is set.

A few things should be noted about the fill patterns. First,
patterns always repeat at even 16-dot intervals, starting with
the top left corner of the screen. Thus, if you start a pattern fill
at column 8, the left side of your filled pattern will start with
the "middle" of the pattern, not with its leftmost side. Second,
the pattern index has no effect whatever on pattern types other
than Hatch or Pattern (that is Hollow, Solid, or User-Defined).
Finally, note that subpattern 8 of the Pattern style is a solid
color fill, just like that obtained by using the the Solid style.

Program 5-1 uses the Filled Bar function (which is very
similar to Rectangle Fill) to show each of the preset fill types,
and the subpatterns for the Hatch and Pattern types.

In the output from Program 5-1, the top row of boxes are
filled with the Hollow, Solid, and user-defined fill patterns.
(Since we have not specified our own fill pattern, the default
user pattern, the Atari logo, is the one that appears.) The next
two rows contain boxes that are filled with the 24 different
Pattern style fills. The final row of boxes displays the 12
Hatch type fill patterns. For the benefit of assembly language
programmers, Program 5-2 is a translation of fillpat.c

Program 5-1. fillpat.c

```
/*********************************************/
/*                                           */
/*                                           */
/*    FILLPAT.C -- Shows the various fill     */
/*    patterns that are available             */
/*                                           */
/*                                           */
/*********************************************/

#include "shell.c"

demo()
{

    int xstep,dx,ystep,dy,scrh,scrw,c,d;
```

```
    int points[4];

      scrw = work_out[0]; /* get screen width */
      scrh = work_out[1]; /* get screen height */
      xstep = scrw/12;    /* each block 1/12 width */
      ystep = scrh/4;     /* and 1/4 height */
      dx = xstep/5;       /* with some space in between */
      dy = ystep/5;

      points[0]= points[1]=0;
      points[2]=xstep*4-dx;
      points[3]=ystep-dy;
      vsf_interior(handle,0);
      v_bar(handle,points); /* draw Hollow block */

      points[2]=xstep*8-dx;
      points[0]=xstep*4;
      vsf_interior(handle,1);
      v_bar(handle,points); /* draw Solid block */

      points[2]=xstep*12-dx;
      points[0]=xstep*8;
      vsf_interior(handle,4);
      v_bar(handle,points); /* draw User-defined (atari) block*/

      for (d=1;d<4;d++)           /* for next three rows */
          for (c=0;c<12;c++)      /* 12 columns in each row */
              {
              points[0]=xstep*c; /* set block coordinates */
              points[2]=xstep*(c+1)-dx;
              points[1]=ystep*d;
              points[3]=ystep*(d+1)-dy;

              if (d<3) vsf_interior(handle,2); /* set Pattern style */
              else vsf_interior(handle,3);
              if (d==2) vsf_style(handle,c+13);
              else vsf_style(handle,c+1);    /* & sub-pattern */

              v_bar(handle,points); /* draw block */
              }
      }
/* End of Fillpat.c */
```

Program 5-2. fillpat.s

```
*************************************************
*                                               *
*                                               *
*      FILLPAT.S -- assembly version of         *
*      the fill-pattern demo                    *
*                                               *
*                                               *
*                                               *
*************************************************


.xdef demo
.xref vwkhnd
.xref contrl0
.xref contrl1
.xref contrl2
.xref contrl3
.xref contrl4
.xref contrl5
```

```
.xref contrl6
.xref contrl7
.xref contrl8
.xref contrl9
.xref contrl10
.xref contrl11
.xref intin
.xref intout
.xref ptsin
.xref ptsout

.text

demo:

    cmp      #639,intout   * if high-res or med-res
    bne      lowx
    asl      dx            * double dx and xstep
    asl      xstep
lowx:
    cmp      #399,intout+2 * if high-res
    bne      lowy
    asl      dy            * double dx and ystep
    asl      ystep

lowy:
*** Set Interior Fill Style to Hollow
    move     #23,contrl0   * opcode for interior style
    move     #0,contrl1    * no points in ptsin
    move     #1,contrl3    * fill style only in intin

    move     #0,intin      * Hollow fill style
    jsr      vdi

*** Draw filled box
    sub      d0,d0
    move     d0,ptsin
    move     d0,ptsin+2    * x1, y1 = 0
    move     xstep,d0
    move     dx,d1
    asl      #2,d0
    sub      d1,d0
    move     d0,ptsin+4    * x2 = xstep*4-dx
    move     ystep,d0
    move     dy,d1
    sub      d1,d0
    move     d0,ptsin+6    * y2 = ystep-dy

    move     #11,contrl0   * opcode for GDP
    move     #1,contrl5    * subcode for Bar
    move     #2,contrl1    * 2 corners of box in ptsin
    move     #0,contrl3    * no integer parameters in intin

    jsr      vdi           * draw filled box

*** Set Interior Fill Style to Solid
    move     #23,contrl0   * opcode for interior style
    move     #0,contrl1    * no points in ptsin
    move     #1,contrl3    * fill style only in intin

    move     #1,intin      * Solid fill style
    jsr      vdi

*** Draw filled box
    move     xstep,d0
    move     dx,d1
    asl      #2,d0
```

```
      move      dØ,ptsin      * set x1 = xstep*4
      asl       #1,dØ
      sub       d1,dØ
      move      dØ,ptsin+4    * set x2 = xstep*8-dx

      move      #11,contrlØ   * opcode for GDP
      move      #1,contrl5    * subcode for Bar
      move      #2,contrl1    * 2 corners of box in ptsin
      move      #Ø,contrl3    * no integer parameters in intin

      jsr       vdi           * draw filled box
*** Set Interior Fill Style to User-defined
      move      #23,contrlØ   * opcode for interior style
      move      #Ø,contrl1    * no points in ptsin
      move      #1,contrl3    * fill style only in intin

      move      #4,intin      * User-defined fill style
      jsr       vdi

*** Draw filled box
      move      xstep,dØ
      move      dx,d1
      asl       #2,dØ
      move      dØ,d2
      add       dØ,dØ
      move      dØ,ptsin      * set x1 = xstep*8
      add       d2,dØ
      sub       d1,dØ
      move      dØ,ptsin+4    * set x2 = xstep*12-dx

      move      #11,contrlØ   * opcode for GDP
      move      #1,contrl5    * subcode for Bar
      move      #2,contrl1    * 2 corners of box in ptsin
      move      #Ø,contrl3    * no integer parameters in intin

      jsr       vdi           * draw filled box

      move      #2,d5         * loop counter d
row:
      move      #11,d4        * loop counter c
      move      #4,d7
      sub       d5,d7         * d+1 is in d7
      move      ystep,dØ
      move      dy,d1
      mulu      d7,dØ
      sub       d1,dØ
      move      dØ,ptsin+6    * set y2 of box = ystep*(d+1)-dy
      subq      #1,d7
      move      ystep,dØ
      mulu      d7,dØ
      move      dØ,ptsin+2    * set y1 = ystep*d

bar:
      move      #12,d6
      sub       d4,d6         * c+1 is in d6
*** Set Interior Fill Style to Pattern or Hatch
      move      #23,contrlØ   * opcode for interior style
      move      #Ø,contrl1    * no points in ptsin
      move      #1,contrl3    * fill style only in intin

      move      #2,intin      * Pattern fill style
      cmp       #Ø,d5
      bne       pattern:
      move      #3,intin      * last row is Hatch fill style
pattern:
      jsr       vdi
```

```
*** Set Fill Style Index
    move      #24,contrl0   * opcode for set index
* contrl1 and contrl3 set correctly from previous call

    move      d6,intin      * use inner counter for index
    cmp       #1,d5         * add 12 to index for middle row
    bne       notmid
    add       #12,intin

notmid:
    jsr       vdi

*** Draw filled box
    move      xstep,d0
    mulu      d6,d0
    sub       dx,d0
    move      d0,ptsin+4    * set x2
    move      xstep,d0
    subq      #1,d6
    mulu      d6,d0
    move      d0,ptsin      * set x1

    move      #11,contrl0   * opcode for GDP
    move      #1,contrl5    * subcode for Bar
    move      #2,contrl1    * 2 corners of box in ptsin
    move      #0,contrl3    * no integer parameters in intin

    jsr       vdi           * draw filled box

    dbra      d4,bar
    dbra      d5,row
    rts


**** data section


    .data
    .even

dx:       .dc.w     5      * 1/60 screen width
dy:       .dc.w     10     * 1/20 screen height
xstep:    .dc.w     26     * 1/12 screen width
ystep:    .dc.w     49     * 1/4 screen height

.end
```

User-Defined Pattern Fill

The one pattern that we have not discussed so far is the user-defined fill pattern. When the user-defined line pattern is selected, we must also tell the VDI what the pattern looks like. This is done in much the same way as we set up the image for the user-defined line pattern. Like the line pattern, each fill pattern is 16 dots wide, which means that each line can be described with a single 16-bit number. While the line pattern is only 1 line high, however, each fill pattern is 16 lines tall.

This means that it takes the equivalent of 16 line pattern descriptions, stacked one on top of the other, to describe a fill pattern. These sixteen 16-bit descriptions are placed in an array, and the address of the array is used to specify the pattern.

In order to determine the values to be placed in this array, binary digits are used to represent each line of filled dots (ones) and unfilled dots (zeros). Just writing out the pattern as binary digits may help you visualize it. For example, let's look at a pattern that draws the letters LOVE in a block, with the first two letters on top of the last two.

```
0000000000000000 = 0 X 0000
0011000001111000 = 0 X 3078
0011000011001100 = 0 X 30CC
0011000011001100 = 0 X 30CC
0011000011001100 = 0 X 30CC
0011000011001100 = 0 X 30CC
0011111001111000 = 0 X 3E78
0000000000000000 = 0 X 0000
0110011011111100 = 0 X 66FC
0110011011000000 = 0 X 66C0
0110011011000000 = 0 X 66C0
0011001011111000 = 0 X 32F8
0001111011000000 = 0 X 1EC0
0000111011000000 = 0 X 0EC0
0000011011111100 = 0 X 06FC
0000000000000000 = 0 X 0000
```

By drawing the pattern using zeros and ones, and converting those binary numbers to hexadecimal, we get the data needed for setting up our user-defined fill pattern. Once we have this data, we can use the VDI call, Set User-Defined Fill Pattern, to establish it as the pattern to be used when we choose fill style 4. The C language format for this call is

int handle, bit_planes, pattern[16*bit_planes];
vsf_udpat(handle, pattern, bit_planes);

where *pattern* is a pointer to our data array, and *bit_planes* is a number used to indicate how may colors are in our pattern. For plain two-color patterns such as the one in our example above, there are 16 elements in the pattern array, and the bit_planes variable should contain a one. A complete example program showing the use of our LOVE fill pattern can be found in the section discussing the Filled Rounded Rectangle GDP, below.

If you forget to set a user-defined pattern before chosing interior fill style 4 with the vsf_interior call, you will get the default user-defined pattern, which on the ST just happens to be the Atari logo. An example of this pattern can be seen in the output of the fillpat.c program, above.

Multicolor Pattern Fill

It's also possible to set the user-defined fill pattern to produce a multicolored pattern. Such a fill pattern is much more complex than the standard two-color fill.

In order to understand how multicolor fill patterns work, we must first discuss the concept of color bit planes. When we talked about color formation previously, we noted that when each dot on the screen can only be displayed in one of 2 colors, you only need one binary digit (bit) to represent that dot, since a one or a zero covers the whole range of possiblities. But if you want to display that dot in any one of 4 colors, you need two bits to represent it. Each time you double the number of possible colors, you need one more bit to represent the dot. Thus, in order to get 8 colors, you need three bits per dot, and to get 16 colors you need four bits per dot.
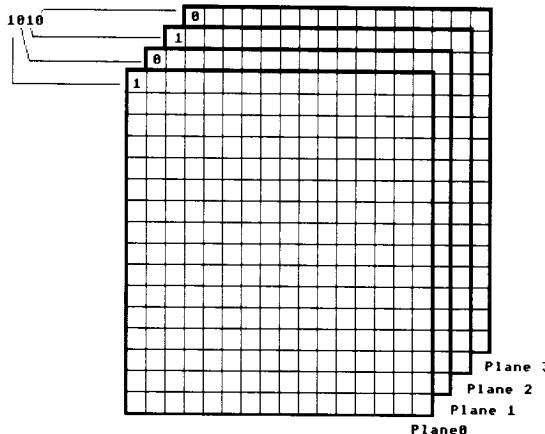
When you know that you're going to have a fixed number of colors, like the 4 or 16 colors provided on the ST, it's easy to say that each byte of display memory will be interpreted as four contiguous pairs of bits, or two 4-bit chunks.

GEM was not designed for a particular system, however, so it had to be made as flexible as possible. If, for example, if GEM was used as the operating system on a computer that displays eight colors on screen at a time, it would be very awkward to say that each byte of display memory holds the information for $2\frac{2}{3}$ dots. Therefore, for the purpose of multicolor pattern fills, color bits are grouped by what are called *bit planes*. In such a grouping, the color bits for a single dots are split up so they aren't contiguous the way they are in ST display memory. Instead, all of the least-significant bits are in one block, followed by a block of the next most-significant bits, and so on. To construct the group of bits necessary to make up the dot in the top, left corner of the picture, you must take the most-significant (leftmost) bit of the first byte of the first block, and join it with the leftmost bits of the first byte of each of the other blocks. Figure 5-1 shows how this works.

Figure 5-1. Arrangement of an Image by Bit-Planes

UPPER LEFT DOT TAKES COLOR
FROM REGISTER 6

1010

Plane 3
Plane 2
Plane 1
Plane0

The color bit plane model is used when setting up data arrays for multicolor pattern fills. Each time you wish to double the number of colors available in the fill pattern, you must add another 16-word group onto the end of the pattern array. The first 16-word group is bit plane zero, the second is bit-plane one, and so on. The total number of 16-word bit planes should be passed to the function in the variable bit_planes. Take, for example, the case of the following fill pattern array:

```
int pattern [ ]={

0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000,

0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF,
0x0000, 0x0000,
0x0000, 0x0000,
0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF,
0x0000, 0x0000,
0x0000, 0x0000,
}
```

The top four words of each bit plane are made up of all ones, so the first four lines will have a one in each bit position. This corresponds to the binary number 11, or or decimal number 3, which means that they will be drawn in with the color in color register 3. The next four words have ones in the least significant bit plane but a zero in the most significant bit, so they're drawn by color register 1. The next four words have the zero in the least significant bit and the one in the most significant bit, so they are in color register 2. And the last four words have zeros in both bit places, so they are background color.

Please note that the numbers formed by joining together the bit planes refer to color registers and not VDI pen numbers. The correspondence between the color registers and the VDI drawing pens (also known as the color index) can be found in Table 4-2. Also note that when using multicolor pattern fills, your pattern array must have the same number of bit planes as the display (for example, two for medium resolution and four for low resolution). If the bit_planes value of your vsf_udpat call does not agree with the actual number of bit planes used by the display, the call will fail and your pattern will not be installed. Finally, keep in mind that when you use the multicolor fill capability of the VDI, each bit plane is combined with the existing picture according to the writing mode, so if you use a mode other than the default Replace mode, things can get extremely complicated.

Program 5-3 shows the use of a four-color fill pattern. The program works on a monochrome monitor also, but you won't get the multicolor effect. The program displays a fill pattern that is composed of squares of color 0, 1, 2, and 3. If you run the program in low resolution, you'll notice that color 3 is shown as yellow, whereas it is black in medium resolution. That's because these colors refer to the hardware registers, not to the VDI pen colors. Even though VDI pen 1 defaults to black in both modes, lo-res uses color register 15 for black, while medium-res uses color register 3. We used the vq_extnd call to find out how may bit-planes our display uses. This value is returned in work_out[4].

Program 5-3. colorpat.c

```
/**********************************************/
/*                                            */
/*                                            */
/*    COLORPAT.C -- Shows the use of the      */
/*    multicolor pattern fill                 */
/*                                            */
/*                                            */
/**********************************************/

#include "shell.c"

int colpat[4][16] = {
   0x00FF, 0x00FF, 0x00FF, 0x00FF, /* four-color fill pattern */
   0x00FF, 0x00FF, 0x00FF, 0x00FF,
   0x00FF, 0x00FF, 0x00FF, 0x00FF,
   0x00FF, 0x00FF, 0x00FF, 0x00FF,

   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,
   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,

   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,

   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,
   };


int sides [4] = { 0,0,0,0,};


demo()
{

  int xstep,ystep,scrh,scrw,c;

    scrw = work_out[0]; /* get screen width */
    scrh = work_out[1]; /* get screen height */
    xstep = scrw/3;     /* each block 1/3 width */
    ystep = scrh/3;     /* and 1/3 height */

    vq_extnd(handle, 1, work_out);  /* find number of bit planes */
    vsf_udpat(handle,colpat, work_out[4] ); /* set our fill pattern */
    vsf_interior(handle,4);        /* and use it */

    for (c=0;c<3;c++)
       {
       sides[0]=sides[2];
       sides[1] = sides[3];
       sides[2]+=xstep;
       sides[3]+=ystep;
       vr_recfl(handle,sides); /* draw filled boxes */
       }

   }


/* End of Colorpat.c */
```

Fill Color and Outlining

There are two more settings that affect fill operations. The first changes the pen number of the foreground color drawn by these operations. (As you may remember, the default fill color was set to the value in work_in[9] at the time the virtual workstation was opened.) This function is called Set Fill Color Index, and its syntax should be familiar to you by now, because it's virtually identical to that of the calls used to set the marker and line colors:

```
int handle, pen;
pen_set = vsf_color(handle, pen)
```

where *pen* is the number of the drawing pen which you are requesting as the fill color, and *pen_set* is the variable in which the function returns the number of the pen that was actually set.

The final setting is used to determine whether or not the filled shapes created by the various VDI calls will be drawn with a solid outline around them. The Set Fill Perimeter Visibility call is the one used to change this setting, and it can be called like this:

```
int handle, visibility_flag;
visibility_set = vsf_perimeter(handle, visibility_flag);
```

where *visibility_flag* is a value used to indicate whether you want a visible outline around the fill area, and *visibility_set* is a variable in which the actual setting is returned. In both cases, a zero value indicates no outlining, and a value of one (or any other nonzero value) specifies that a visible outline will be drawn. It should be noted that this particular setting does not affect the Fill Rectangle (vr_recfl) function, which always draws the rectangle without an outline.

Settings Inquiry

As with the marker and line settings, the current status of the fill settings can be determined with a single VDI call. The name of this function is Inquire Current Fill Area Attributes, and it's called like this:

```
int handle, settings[4]
vqf_attributes(handle, settings);
```

where *settings* is a pointer to the array in which the function returns the information about the fill settings. The contents of the array are interpreted as follows:

Element	Setting
settings[0]	fill pattern type
settings[1]	fill color pen
settings[2]	fill pattern index
settings[3]	current draw mode

Filled Shape Generalized Drawing Primitives (GDPs)

The VDI supplies a number of GDPs which can be used to create a wide variety of filled shapes. The simplest is Bar, which is used to draw a rectangle. This may seem to be wasteful redundancy, since a wide polyline or a rectangle fill each produce a filled box, but Bar is just a little bit different. A wide polyline, for example, can be used for a solid box, but it cannot be filled with a pattern, and it uses the line settings rather than the fill settings. The rectangle fill function is designed to speedily clear a rectangular area of the screen only (not other graphics output devices), thus it doesn't use the outline setting. The bar function, however, can be used by any device, and it does support outlining. (As its name suggests, it's very handy for bar graphs.)

While we're on the subject of overlapping functions, you should note that if you set the fill pattern to Hollow, the drawing mode to Transparent, and turn on perimeter visibility, then the Bar function can be used to draw just the frame of a box, as you might do with Polyline.

The syntax for the C language version of Bar is

```
int handle, sides[4];
v_bar(handle, sides);
```

where *sides* is a pointer to an array that holds the location of each of the four sides of the rectangle. The location of the left and right sides are in side[0] and side[2], while the top and bottom are in side[1] and side[3], respectively.

The filled shape equivalent of the line drawing Rounded Rectangle function is called Filled Rounded Rectangle. As with the bar function above, all you have to do to draw a filled rounded rectangle is point to an array that contains the coordinates of the top left and bottom right corners of the rectangle:

int handle, sides[4];
v_rfbox(handle, sides);

Program 5-4 uses the Filled Rounded Rectangle function to demonstrate the user-defined fill pattern option. The rounded boxes are filled with the LOVE pattern discussed in the section on user-defined fills, above.

Notice how laying new boxes on top of the existing ones changes the color of the pattern, but not its placement. That's because the pattern is always aligned starting with the top left corner of the screen.

Program 5-4. userfill.c

```
/*******************************************/
/*                                         */
/*                                         */
/*    USERFILL.C -- Shows the use of the   */
/*    user-defined fill pattern            */
/*                                         */
/*                                         */
/*******************************************/

#include "shell.c"

int lovepat[16] = {
    0x0000, 0x3078, 0x30CC, 0X30CC, /* "love" fill pattern */
    0X30CC, 0X30CC, 0X3E78, 0X0000,
    0X66FC, 0X66C0, 0X66C0, 0X32F8,
    0X1EC0, 0X0EC0, 0X06FC, 0X0000,
    };


int points [4] [4] = {
    3,3,18,18,              /* array of corners for boxes */
    7,1,12,20,
    1,9,20,13,
    14,10,13,20,
    };

demo()
{

   int xstep,ystep,scrh,scrw,c,d;
   int sides[4];

    scrw = work_out[0]; /* get screen width */
    scrh = work_out[1]; /* get screen height */
    xstep = scrw/20;    /* each block 1/20 width */
    ystep = scrh/20;    /* and 1/20 height */

    vsf_udpat(handle,lovepat, 1); /* set out fill pattern */
    vsf_interior(handle,4);       /* and use it */

    for (d=0;d<4;d++)       /* for 4 boxes */
        {
        for (c=0;c<2;c++)    /* for 2 sets of corners */
            {
            sides[c*2]=points[d][c*2]*xstep; /* set block coordinates */
            sides[c*2+1]=points[d][c*2+1]*ystep;
            }
```

```
if (d==1) vsf_perimeter(handle,0); /* outline only 1st box */
vsf_color(handle,d+1); /* change fill color */
v_rfbox(handle,sides); /* draw rounded rectangle */
}
}
```

```
/* End of Userfill.c */
```

The final four filled shape GDPs allow you draw filled
circles, ellipses, or pie-shaped wedges of either. The filled cir-
cle and the circular pie functions correct the vertical radius for
the aspect ratio of the display screen, so the figures that they
draw appear to be round even on displays like the medium-
res color screen, which has tall, skinny pixels. The filled el-
lipse and elliptical pie functions use whatever horiztonal and
vertical radius that you specify.

The calling sequence for the Circle function is

int handle, x, y, radius;
v_circle(handle, x, y, radius);

where x and y describe the center point of the circle, and *ra-
dius* is the radius measured horizontally. (The vertical radius is
adjusted—it automatically corrected to compensate for the as-
pect ratio of the screen.)

The syntax of the C version of the Ellipse call is

int handle, x, y, xradius, yradius;
v_ellipse(handle, x, y, xradius, yradius);

where x and y specify the center point of the ellipse, and
xradius and *yradius* describe the horizontal and vertical radii.

The format for the Pie call is

int handle, x, y, radius, beginangle, endangle;
v_pieslice(handle, x, y, radius, beginangle, endangle);

where x and y are the coordinates for the midpoint of the cir-
cle, *radius* is its radius measured horizontally (the vertical ra-
dius is adjusted for the aspect ratio), and *beginangle* and
endangle mark the starting and ending points for the enclosed
arc. As with the v_arc call, these angles are measured in
1/10s of a degree, starting at the rightmost point of the circle
as zero degrees, and moving counterclockwise, so that the top-
most point is at 900, the leftmost at 1800, and so on. The
function draws the arc of the circle described by beginangle
and endangle, connects each end of the arc to the midpoint,

and fills the resulting shape according to the current fill pattern, fill color, and writing mode.

The syntax for the Elliptical Pie function is very similar:

int handle, x, y, xradius, yradius, beginangle,endangle;
v_ellpie(handle, x, y, xradius, yradius, beginangle, endgangle);

The only difference is that you must supply values for both the horizontal and vertical radii of the ellipse.

Program 5-5 uses the GDP Ellipse command. It also demonstrates a very important point to remember. Pattern fills are drawn according to the current writing mode, just like patterned lines are. As you can see from the display created by Program 5-5, the oval drawn in Replace mode (lower left) obscures its portion of the green block completely. The ellipses drawn in Transparent and Reverse Transparent modes (top left and top right) let the green block show through everywhere the fill color was not drawn. And the ellipse drawn in XOR mode is filled with a different color inside the block than it is outside the block, since it merely complements the existing colors.

Program 5-5. fillmode.c

```
/**********************************************/
/*                                            */
/*                                            */
/*    FILLMODE.C -- Demonstrates effect of    */
/*    the writing mode on filled shapes       */
/*                                            */
/*                                            */
/**********************************************/

#include "shell.c"

demo()
{

  int c,cx,cy,hr,vr,scrh,scrw;
  long r1;
  int points[4];

    scrw = work_out[0]; /* get screen width */
    scrh = work_out[1]; /* get screen height */

    vr = points[1] = scrh/4;  /* set box corners, */
    hr = points [0] = scrw/4; /* ovals midpoints */
    points[2] = scrw-hr;
    points[3] = scrh-vr;

    vsf_color(handle,3); /* green box */
    v_bar(handle,points);

    vsf_color(handle,2);    /* fill in red */
    vsf_interior(handle,2); /* Pattern style */
    vsf_style(handle,12);   /* sub-pattern 12 */
```

109

```
for (c=0;c<4;c++)
    {
    if (c<2) cx = hr; /* horiz coordinate of oval center */
    else cx = scrw-hr;

    if (c & 1) cy = vr; /* vertical coordinate of center */
    else cy = scrh-vr;

    vswr_mode(handle,c+1); /* change writing mode */
    v_ellipse(handle,cx,cy,hr,vr,0,3600); /* draw ellipse */
    }
/* End of Fillmode.c */
```

Area Fill

The next function is the filled shape analog of the Polyline function. The Filled Area call takes an arbitrary number of points that you specify, connects them, and fills the resulting figure using the current fill settings. The shape that you create may cross over itself in one or more places, like a figure 8; the function will fill some of the loops, but may leave some adjacent loops unfilled.

The sytnax for the Filled Area call is

int handle, count, points[2*COUNT];
v_fillarea(handle, count, points);

where *count* holds the number of vertices to be connected, and *points* is a pointer to an array of x and y coordinates for those points. Since each point has a horizontal and vertical component, the array contains twice as many elements as there are points. Note that in order to insure that the points describe an enclosed shape, this function connects the last point in the list to the first point. Thus, it only requires four points to describe a filled rectangle, while Polyline requires five points to draw the outline of that box. The function will not draw a figure that only has one point. If the shape has no area to fill, it is represented by a single dot if visible outlining is turned on, and is not drawn at all if outlining is turned off.

Program 5-6 shows how to create a complex filled polygon using the v_fillarea command. Note that where the shape crosses itself so that there are two or more adjacent enclosed spaces, the interior ones are left unfilled so that they appear to be "outside" the polygon.

Program 5-6. areafill.c

```
/**************************************************/
/*                                              */
/*                                              */
/*    AREAFILL.C -- Shows the use of the        */
/*    area fill command                         */
/*                                              */
/*                                              */
/**************************************************/

#include "shell.c"

int points [16] = {
    1,2,16,2,7,9,12,20,    /* vertices for polygon */
    6,0,0,10,18,14,20,16,
    };

demo()
{

   int xstep,ystep,scrh,scrw,c,d;
   int sides[16];

     scrw = work_out[0]; /* get screen width */
     scrh = work_out[1]; /* get screen height */
     xstep = scrw/20;     /* each block 1/20 width */
     ystep = scrh/20;     /* and 1/20 height */

     vsf_interior(handle,4);       /* Atari-fuji fill pattern */

        for (c=0;c<8;c++)    /* for 7 sets of corners */
           {
           sides[c*2]=points[c*2]*xstep; /* set x&y coordinates */
           sides[c*2+1]=points[c*2+1]*ystep;
           }
           vsf_color(handle,2); /* red fill color */
           v_fillarea(handle,8,sides); /* draw filled polygon */

     }


/* End of Areafill.c */
```

Flood Fill

The last of the shape filling commands is a general-purpose flood fill. Unlike the previous commands that we've discussed, a flood fill (or contour fill, as it is sometimes called) does not first draw a shape and then fill it in. Rather, it colors in an existing enclosed area. The color and pattern with which it fills the area depend on the fill color and pattern settings.

Flood filling operates in one of two modes. In outline mode, the entire area enclosed by a border of the outline pen color is filled. Filling begins at a point which you specify and moves outward in all directions. As it does so, every horizontally and vertically adjacent pixel which is not colored with the pen designated in the call as the outline color is filled according to the fill color and pattern. The fill pattern stops spreading at each point where it encounters a pixel colored by the outline or contour pen. If the area to be filled is not completely surrounded by a border of that outline color, the fill will "leak" out, and the entire display area (or clipping rectangle) will be filled.

In color mode, all adjacent pixels of the same color are filled. You designate the point at which filling begins, and whatever pen was used to color that point becomes the color which the fill routine displaces. As the fill moves outward, every horizontally and vertically adjacent pixel which is colored with the displacement pen is filled. The fill stops spreading at each point where a pixel drawn in another pen color is encountered.

The syntax for the Contour Fill call is

int handle, x, y, pen;
v_contourfill(handle, x, y, pen);

where x and y specify the point at which filling begins, and *pen* specifies the outline pen number for outline mode. If the value for pen is negative, the color replace mode is used, and the fill replaces adjacent pixels that are drawn in the same pen color as the point x,y.

Program 5-7 demonstrates both modes of contour filling. First, a frame is created out of two wide rounded boxes. Next, the enclosed spaces are filled in outline mode. Finally, the frame itself is filled using color mode.

Program 5-7. flood.c

```
/********************************************/
/*                                          */
/*                                          */
/*    FLOOD.C -- Demonstrates two different  */
/*    kinds of flood fill using the          */
/*    v_contourfill command                  */
/*                                          */
/********************************************/
#include "shell.c"
int points [2] [4] = {
    1,6,19,12,            /* array of corners for boxes */
```

```
    6,1,13,19,

    >;

demo()
{

  int xstep,ystep,scrh,scrw,c,d;
  int sides[4];

    scrw = work_out[0]; /* get screen width */
    scrh = work_out[1]; /* get screen height */
    xstep = scrw/20;    /* each step 1/20 width */
    ystep = scrh/20;    /* and 1/20 height */

    vsl_width(handle,9);     /* wide lines for boxes */
    vsf_color(handle,2);     /* fill color = red */
    vsf_interior(handle,2);  /* use patterned fill */
    vsf_style(handle,19);

    for (d=0;d<2;d++)        /* for 2 boxes */
        {
        for (c=0;c<2;c++)   /* for 2 sets of corners */
            {
            sides[c*2]=points[d][c*2]*xstep; /* set box coordinates */
            sides[c*2+1]=points[d][c*2+1]*ystep;
            }
        v_rbox(handle,sides); /* draw rounded rectangle */
        }

    for (d=0;d<2;d++)
        /* for 2 boxes, fill at opposite corners */
        {
        v_contourfill(handle,points[d][0]*xstep+10,
        points[d][1]*ystep+10,1);
        v_contourfill(handle,points[d][2]*xstep-10,
        points[d][3]*ystep-10,1);
        }
    vsf_color(handle,3);   /* change fill color to green */
    vsf_style(handle,16);
        /* change fill pattern for mono systems */
    v_contourfill(handle,xstep,7*ystep,-1);
        /* fill outline of boxes */

}

/* End of Flood.c */
```

BASIC Fill Commands

The first release of ST BASIC contains a number of keyword commands that correspond to the filled shape commands that we've covered in this chapter. The COLOR command can be used to set not only the fill color but also the fill style and index. There is also direct support for the GDPs that drew filled circles, ellipses, pie slices, and elliptical pies. The BASIC command PCIRCLE creates a filled circle or pie slice, while the command PELLIPSE outputs a filled ellipse or elliptical pie slice. Finally, the FILL command supports the contour fill function.

Although not yet released at the time of this writing, the planned revision of ST BASIC appears to offer even more support for the filled shape functions. Area filling is supported in two formats:

AREA x,y; x1,y1; x2,y2;.....xn,yn
MAT AREA count, array( )

In the first, the keyword *AREA* is followed by a minimum of 3 coordinate pairs separated by semicolons. These coordinates specify the area to be filled. In the second format, you place the coordinates in an array, and then specify the number of points to be drawn and the name of the array.

The Bar command is supported by a variation of the BOX command:

BOX FILL x1,y1;x2,y2

where the first set of coordinates specifies the upper left corner of the box, and the second specifies the lower right corner. Finally, the new ST BASIC supports the user-defined fill pattern with the command PATTERN:

PATTERN planes, array

where *planes* is the number of bit planes, and *array* is the name of the array which holds that number of 16 two-byte values.
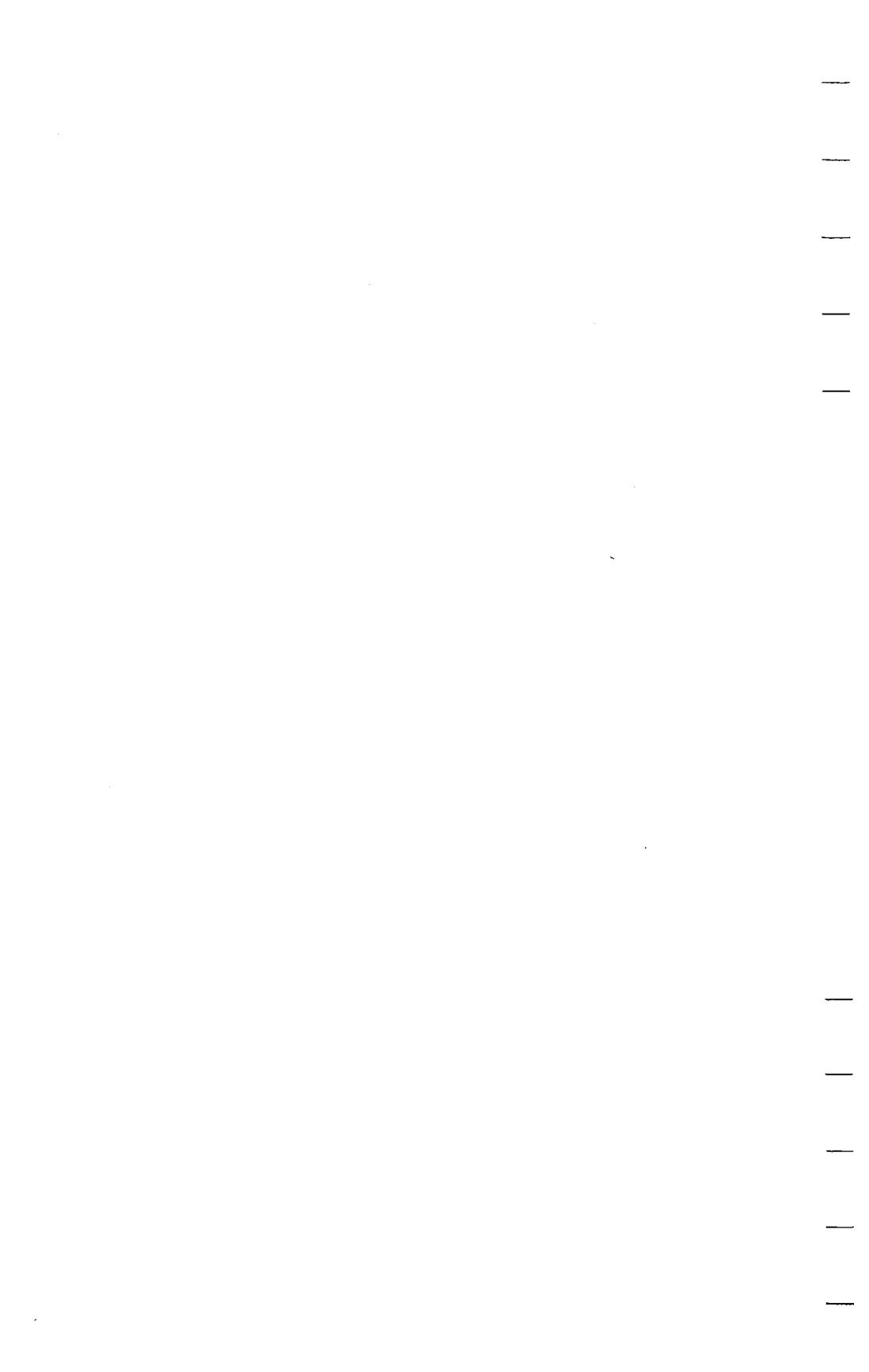
Of course, you can still access all of these functions using the POKE and VDISYS commands. Program 5-8 shows how to use some of the unsupported functions, such as v_rfbox.

Program 5-8. fill.bas

```
100    fullw 2: clearw 2
110    res = peek(systab)
120    if (res<4) then scrw = 639 else scrw = 319
130    if (res>1) then scrh = 199 else scrh = 399
140    xstep = scrw/20: ystep = scrh/20
150    REM Set User-defined Fill Pattern
160    poke contrl,112 :REM opcode for udpat
170    poke contrl+2, 0 :REM no points in ptsin
180    poke contrl+6, 16 :REM 16 words of pattern data in intin array
190    for x=0 to 15
200    read d: poke intin+(2*x),d
210    next x
220    vdisys(1)
230    REM
240    for d=0 to 3
250    if d<>1 then goto 340: REM outline only first box
260    REM
270    REM Set perimeter outline visibility
280    poke contrl,104 :REM opcode
```

114

```
290    poke contrl+2,0
300    poke contrl+6,1
310    poke intin,0
320    vdisys(1)
330    REM
340    REM Set Fill Color and interior fill style
350    COLOR 1,d+1,1,4
360    REM
370    REM Draw Filled Rounded Rectangle
380    poke contrl,11     :REM opcode for GDP
390    poke contrl+10,9   :REM sub-opcode for rfbox
400    poke contrl+2,2    :REM 2 corners in ptsin
410    poke contrl+6,0
420    for c=0 to 1
430    read x,y
440    poke ptsin+(c*4),x*xstep
450    poke ptsin+(c*4+2),y*ystep
460    next c
470    vdisys(1)
480    REM
490    next d
500    REM
510    DATA 0,12408,12492,12492,12492,12492,15992,0
520    DATA 26364,26304,26304,13048,7872,3776,1788,0
530    REM
540    DATA 4,5,16,16
550    DATA 7,3,12,18
560    DATA 1,9,18,13
570    DATA 14,10,13,18
```

Chapter 6

Drawing and Manipulating Image Blocks

So far, we've seen how the VDI provides functions to draw images point by point or with lines and geometric shapes. But perhaps the most powerful of the VDI drawing functions are the raster functions that move and manipulate an entire block of pixels at once. This type of operation, often referred to as a Bit BLiT (Bit BLock Transfer), allows you to draw and animate images on the display screen. On the original ST models, the bit manipulation is performed entirely in software. Atari has been working on hardware support, however, in the form of a blitter chip, a device that greatly speeds up such operations. By the time you read this, this hardware upgrade and new TOS ROMs that support its use may already be available.

The VDI raster operations are extremely flexible. The blocks of memory that they move may be located in the screen display area or in the program's data storage space. They can copy images that have the same number of colors as the current display mode or place two-color images into a multicolor display. The images may be reproduced exactly, or they may be combined in a number of different interesting ways with existing images. All of the image or only a selected portion of it may be moved.

The one thing that all raster operations have in common is the format used to describe the bit image. Before the VDI can perform the memory manipulation necessary to move images on the screen, it needs several key pieces of information. These include the starting memory location of the image data, the width and height of the image in pixels, the number of words of data necessary to store the image, the format of the bit image, and the number of color planes used. Since in GEM parlance a bit image is known as a raster form, the data structure in which this information is stored is called a Memory Form Definition Block (MFDB). It consists of ten, 16-bit words of information, laid out as follows:

119

Word	Contents
1	High half of the beginning address of the image data
2	Low half of the beginning address of the image data
3	Raster image width in pixels
4	Raster image height in lines
5	Raster image width in words
6	Image format flag
	0 = ST specific format
	1 = Standard GEM format
7	Number of color bit planes
8	Reserved for future use
9	Reserved for future use
10	Reserved for future use

The C language definition for this data structure is

```
typedef struct fdbstr {
    int *fd_addr; /* pointer to image data area */
    int fd_w; /* image width in pixels */
    int fd_h; /* image height in pixels */
    int fd_wdwidth; /* image width in words */
    int fd_stand; /* standard format flag */
    int fd_nplanes; /* number of color bit planes */
    int fd_r1, fd_r2, fd_r3; /* reserved for future use */
}FDB;
```

This definition may be found in some versions of the header file Obdefs.h or in another header file that comes with your C compiler. Our definition uses the variable type int to describe a 16-bit value, but for compilers that use a 32-bit-wide int, the variable type would have to be changed to short (or WORD, if that term has been defined by a portability macro).

The first member of this structure, fd_addr, is a pointer to the integer array that holds the actual shape data for the image. As an address pointer, it's a 32-bit value. Some versions of the structure definition make the first element a pointer to a char, but since the image data is always an even number of words long, it's more convenient to use a pointer to int. We'll discuss the size and format requirements of the image data block that this value points to a little bit later on. If the value stored in fd_addr is zero, rather than an actual address, it's a signal for the VDI to use screen display memory for the image block. In such a case, the VDI ignores the rest of the values in the memory form definition block. It uses the beginning address of screen memory for the first value, and the width,

height, and number of bit-planes for the current display
screen. The format flag is set to show that the display is in ST-
specific format.

The next two members, fd_w and fd_h, specify the width
and height of the image in pixels. Though the actual image
data block is made up of word-length values, and thus must
be an even-multiple-of-16 pixels wide, the image itself does
not have to occupy all of that width. For example, you can de-
scribe an image that's 26 pixels wide, even though you must
use 32 bits of data to do it. If the rightmost portion of the im-
age only uses a part of the last word on each line, like the ex-
ample above which only uses 10 bits of the last word, it's
known as a fringe. Images that are not an even-multiple-of-
16-bits wide tend to be drawn a bit more slowly than images
that are an even-number-of-words wide, since the VDI is al-
ways forced to do bit manipulation on them to mask out the
unwanted bits.

The next member of the structure, fd_wdwidth, is used to
store the number of words of image data per line. If the width
is not an even multiple of 16 pixels, you've got to round it up
to the next highest even multiple and then divide by 16 to get
this value.

Next in the structure comes fd_stand, a flag that shows
whether the image data is arranged in standard GEM format,
or the machine-specific format of the host computer's display
circuitry. A value of 1 means that it is in the standard format,
while a value of 0 means that it is arranged in the format of
the ST display memory.

The last significant item in this data structure is
fd_nplanes. This item is used to store the number of color bit
planes used by the image. As we have explained earlier, one
bit plane is needed for a monochrome (actually, 2-color) im-
age, two bit planes are needed for a 4-color image, and four
are required for a 16-color image. Since each of the ST's dis-
play modes uses a different number of bit planes, your appli-
cation should determine how many planes are in the current
screen and proceed accordingly. The vq_extnd function can be
used to determine the number of planes in the display; this
value is returned in work_out[4] when you use the call to re-
trieve the Extended Inquire information.

The most important piece of information needed to draw
bit images, though, is the actual image shape data. GEM al-

lows this image data to be stored in one of two different formats. The first, the machine-specific format, is the fastest and easiest for the VDI to use, since it conforms to the internal configuration of the ST's own display memory. The second, the GEM standard format, is offered for purposes of portability. Since the VDI offers a function for converting an image from one format to the other, you can create an image in the GEM standard format, and then convert it to the machine-specific format of the host computer, without having any idea what the display memory layout of that computer is like. If you plan to write software only for the ST, though, you'll probably have no need for the standard format.

By now, both formats for image data storage should be fairly familiar to you. We discussed the ST display memory scheme, in Chapter 4, as an interleaved bit-map. This means that color information is stored in adjacent bits in the same byte of memory. In the 4-color mode, the first byte of each line describes the four colored dots at the extreme left of that line. Each adjacent bit pair stores a number from 0 to 3, indicating the color register used to color that dot. The most significant two bits describe the leftmost dot, and each less significant bit pair describes the next dot to the right. In the 16-color mode, the first byte of each line describes first two colored dots on the line. Each nybble (four-bit group) stores a number from 0 to 15, indicating the color register used to draw the dot. The high-order nybble describes the leftmost dot, and the low order nybble holds the information for the dot to its right.

Standard GEM image format is like the format used to store multicolored fill patterns, that we described in Chapter 5. In standard format, each bit of color information is in a separate data block called a bit-plane. A 4-color image has two separate bit planes, and a 16-color image contains four bit planes. Each plane contains a different bit of color data for the same dot. For example, the most significant (leftmost) bit of the first word of each line of data in each bit plane contains information about the first dot in the top line of the picture. The bit in plane 0, the first plane, contains the least significant bit of information, and the bits in each succeeding plane contain the next most significant bit of information. Putting the

bits from the various planes together forms the number that gives the color data for that dot. (See Figure 5-1.)

Note that a two-color (monochrome) image has one bit plane in standard format, just as it does in the ST-specific format, since each dot has only one bit of associated color data. This means that for monochrome images, the standard format is exactly the same as the ST-specific format.

Copy Raster Opaque

The first of the VDI raster functions is called Copy Raster Opaque. Its name comes from the fact that this function copies the same number of bit planes from the source memory area as there is in the destination area, so that the former can be copied pixel by pixel to the latter. The source image can't be rotated or scaled in size with this function, though you can use this function to move the image data from the screen to memory, where you can manipulate it more easily. The C syntax for this call is

```
int handle, mode, points[8];
struct fdbstr *srcMFDB, *destMFDB;

vro_cpyfm(handle, mode, points, srcMFDB, destMFDB);
```

where the value *mode* indicates the writing mode used for the operation. Despite its name, this function does not necessarily perform a straight copy of the source image. Rather, it can combine an image with the existing destination image in a number of interesting ways. These writing modes are similar to the general drawing mode set by the vswr_mode( ) call, but they are set separately and are more comprehensive. They include the old standbys like Replace and XOR mode, and also add new combinations, some more useful than others. The following chart shows the 16 different combinations available with the vro_copyfm( ) call. The logic operations are described using the symbol S to refer to the source image, D to refer to the starting destination image, and $D1$ to refer to the resulting destination image. Some of the more useful modes also have a plain-language description that explains more clearly what they do.

Mode Number	Logic Operation	Description
0	D1 = 0	Clear destination block
1	D1 = S AND D	
2	D1 = S AND (NOT D)	
3	D1 = S	Replace mode
4	D1 = (NOT S) AND D	Erase mode
5	D1 = D	Destination unchanged
6	D1 = S XOR D	XOR mode
7	D1 = S OR D	Transparent mode
8	D1 = NOT (S OR D)	
9	D1 = NOT (S XOR D)	
10	D1 = NOT D	
11	D1 = S OR (NOT D)	
12	D1 = NOT S	
13	D1 = (NOT S) OR D	Reverse transparent mode
14	D1 = NOT (S AND D)	
15	D1 = 1	Fill destination block

As you can see, some of these operations are almost useless. For example, number 0 blanks the destination rectangle to the background color, while number 15 fills it with all ones, in effect changing it to pen color 1. Either of the operations could be performed more effectively with vr_recfl( ). Mode number 5 literally does nothing, leaving the destination unchanged, while mode number 10 merely reverses every bit in the destination, without regard to the source image.

The points parameter in the vro_copyfm( ) call is a pointer to an array of coordinates that describe two rectangles. Since the vro_copyfm( ) call can be used to move only a portion of the total image described in the MFDB, these rectangles describe the portion of the source MFDB from which the image is copied and the portion of the destination MFDB to which it is copied. The elements of the points array are

Element	Description
points[0]	Left edge of source rectangle
points[1]	Top edge of source rectangle
points[2]	Right edge of source rectangle
points[3]	Bottom edge of source rectangle
points[4]	Left edge of destination rectangle
points[5]	Top edge of destination rectangle
points[6]	Right edge of destination rectangle
points[7]	Bottom edge of destination rectangle

These points describe an offset from the upper left corner of the form. Though the entire source and destination forms need not be the same size, the source and destination rectangles that you describe should be. If they are not, unpredictable results may occur. (At best, the size of the source rectangle will be used.) You should also take care to make sure that the rectangles you describe do not exceed the width or height of the form as a whole.

The final two parameters are pointers to the source and destination MFDBs. Both of these MFDBs must use image data that is in ST-specific format. Standard-format MFDBs should be converted to ST-specific format prior to this call with the Transform Form (vr_trnfm) call. It is possible for the source and destination forms to be one and the same. For example, you can define a form which uses the display screen for its image data (by setting fd_addr to zero) and use this call to move an image from one part of the display to another part. It's even possible to move an image from a source rectangle that overlaps with the destination rectangle. In such a case, the VDI copies in whatever direction is necessary to preserve the source image, so that the destination doesn't corrupt the source before the copy is complete.

Program 6-1 demonstrates the use of the Copy Raster Opaque function. It first draws a happy face, using the normal VDI drawing commands. It then uses vfo_copyfm( ) to move that image from the screen to a memory form. Finally, it fills the screen with a patterned background, and then uses vro_copyfm to move the image back to the screen, using each of the 16 drawing modes.

Program 6-1 illustrates the various copy modes more clearly than any description of them. The mode numbers progress from the top left corner (0) to the bottom right corner (15). The face in the top right corner, which was copied using mode number 3, Replace, shows the image that was originally drawn on the screen, as it appeared (briefly) before we filled the screen with the pattern. If you have a color display, you will notice that the results in lo-res mode are slightly different from those in medium-res mode. If you add more colors to the picture, the results get even harder to predict. That's because the logical operations which combine the two images are performed separately on each bit plane.

Program 6-1. copymode.c

```
/**************************************************/
/*                                              */
/*                                              */
/*    COPYMODE.C -- Demonstrates copying        */
/*    modes offered by the Copy Raster          */
/*    Opaque function.                          */
/*                                              */
/*                                              */
/**************************************************/

#include "shell.c"

#define WHITE 0
#define BLACK 1
#define RED   2
#define GREEN 3


demo()
{
    struct fdbstr
    {
      int  *image;    /* memory pointer */
      int  width;     /* form width in pixels */
      int  height;    /* form height */
      int  wordw;     /* form width in words */
      int  flag;      /* form flag */
      int  planes;    /* number of color planes */
      int  r1, r2, r3;
    )srcMFDB, destMFDB;

    int imagedat[1000]; /* buffer for destMFBD image data */
    int points[8];
    int xstep,ystep,
        xres,yres,
        scrh,scrw,
        c,d,
        mode;

    scrw =  work_out[0];  /* find screen width */
    scrh =  work_out[1];  /* and height */
    xstep = scrw/4;   /* set x and y step increments */
    ystep = scrh/4;   /* to 1/4 screen width and height */

/* Draw a happy face with VDI drawing commands */
    vsf_interior(handle,1);
    v_ellipse(handle,(xstep-10)/2,(ystep-10)/2,(xstep-16)/2,
    (ystep-16)/2,0,3600);
    vsf_color(handle,GREEN);
    v_ellipse(handle,(xstep-10)/4,(ystep-10)/3,(xstep-12)/8,
    (ystep-12)/8,0,3600);
    vsf_color(handle,WHITE);
    v_ellipse(handle,(xstep-10)/4,(ystep-10)/3,(xstep-12)/16,
    (ystep-12)/16,0,3600);
    vsf_color(handle,RED);
    v_ellipse(handle,3*(xstep-10)/4,(ystep-10)/3,(xstep-12)/8,
    (ystep-12)/8,0,3600);
    vsf_color(handle,WHITE);
    v_ellipse(handle,3*(xstep-10)/4,(ystep-10)/3,
    (xstep-12)/16,(ystep-12)/16,0,3600);
    vsl_color(handle,WHITE);
    vsl_width(handle,5);
    v_ellarc(handle,(xstep-10)/2,(ystep-10)/2,
    (xstep-12)/3,(ystep-12)/3,2100,3300);
```

```
/* Use vq_extnd to find x and y resolution, # of bit planes */

    vq_extnd(handle,1,work_out);
    if ( work_out[4] == 4) xres=1; else xres=2;
    if (work_out[4]== 1) yres=2; else yres=1;

/* Set up a source form using screen data,
   and a destination form using a memory buffer */

    srcMFDB.image = 0L;             /* use screen data for srcMFDB */
    destMFDB.image = imagedat;
    destMFDB.width = 80*xres;
    destMFDB.height = 50*yres;
    destMFDB.wordw = 5*xres;
    destMFDB.flag = 0;              /* ST-specific form */
    destMFDB.planes = work_out[4];

   points[0]=points[4]=points[1]=points[5]=0;
   points[2]=points[6]=xstep-10;
   points[3]=points[7]=ystep-10;

/* Copy the happy face from the screen to the memory form */

   vro_cpyfm(handle,3,points,&srcMFDB, &destMFDB);

/* Flood the whole screen with a cross-hatch pattern */

   points[2]= scrw;
   points[3] = scrh;
   vsf_interior(handle, 3);
   vsf_style(handle,3);
   vsf_color(handle, BLACK);
   vr_recfl(handle,points);

 /* copy the form back from memory to the screen,
    using each of the 16 copy modes */

   points[2] = xstep-10;
   points[3] = ystep-10;
   points[4] = xstep*3;

    for(c=mode=0;c<4;++c)
       {
       points[5]=ystep*c+5;
       points[7]=ystep*(c+1)-5;
       for (d=0;d<4;++d)
          {
          points[4]=xstep*d+5;
          points[6]=xstep*(d+1)-5;
          vro_cpyfm(handle,mode++,points,&destMFDB, &srcMFDB);
          }
       }


   }

       /* End of Copymode.c */
```

There are many uses for the vro_copyfm( ) call. It can be used to move around large areas of the screen display, as when you scroll the contents of a window. It can be used for stamping images on the screen, or for saving predrawn images from the screen to a memory buffer, where they may be manipu-

lated and redrawn. It can even be used for animating color shapes, by drawing the shape, erasing it, and moving it.

The XOR mode is particularly useful for this type of animation. As was mentioned in the discussion of the writing modes, an XOR drawing operation is by definition reversible, since reversing the bit patterns the first time changes the color, while reversing them a second time restores them to their original pattern. Therefore, to move an image that was drawn with an XOR operation, you need only to draw it with an XOR to the same spot to erase it, and then draw it with an XOR to the new spot to make it move. While this course of action is convenient, it isn't exactly without flaw. As we have seen, the color that an XOR image takes depends on the color of the background on which it's drawn, and, if the background is multicolored, the image will be, too. If the image is moved over a very complex colored background, therefore, it will change color as it moves. And if there's more than one image moving at the same time, these images will change colors yet again when they pass over one another. Because of the complexity of the combinations of the various bit-planes in a multicolor image, these color changes can be unpredictable. Yet, despite these limitations, the XOR drawing operation can be effective for animation in many situations. An example of animation using the XOR mode can be found at the end of this chapter, in the section dealing with the vrt_copyfm( ) command.

Transform Form

The advantage of using the standard form for image data storage is that it allows you to render an image without knowing the specifics of the display memory layout used by the target computer. But since vro_cpyfm and vrt_copyfm both require that the source and destination forms be in machine-specific format, you've got to have a way of converting to that format.

Converting forms back and forth between standard and machine-specific formats is the function of the VDI routine named Transform Form. It moves the source form to the destination form, converting it to the opposite type (indicated by the fd_stand flag of the source form) along the way. The function may be called like this:

int handle;
struct fdbstr *srcMFDB, *desMFDB;

vr_trn_fm(handle, srcMFDB, desMFDB);

where *srcMFDB* is a pointer to the source MFDB, and *desMFDB* is a pointer to the destination MFDB. Note that the source and destination form definition blocks may be the same. In such a case, the form is said to be transformed in place. While this is fine for small images, the process can be very slow for larger ones on machines like the ST, where the machine-specific layout is very different from that of the standard format. So, if speed is a consideration, it would pay to set up a destination block separate from the source.

Program 6-2 uses a color image that's stored in the standard format. In fact, it's almost the same image data that was used in the Colorpat program for the four-color pattern fill. The only real difference between the data format used by color pattern fills and standard forms is that the former is only one word (16 bits) wide by 16 lines high, while size of the latter is not so restricted (our example is 32 bits wide). Since the image is not very large, we performed the transformation to the ST-specific form in place, before using the vro_copyfm( ) command to draw the image.

Notice that by providing enough data for the largest number of bit planes in use (4), we can use the same image data for all three resolution modes. The modes that need less data only use the number of planes that they need, and so have roughly the same form as the lo-res image, if less color detail. But because of the difference in the number and size of pixels in each mode, the image will not appear in the same size and aspect ratio in the various modes. Therefore, you'll probably want to supply different image data for each of the three modes, even if using the standard format. If not, your images will turn out like the icons on the Desktop—tall and skinny in medium resolution, and much larger in low resolution than in high resolution.

Program 6-2. stdform.c

```
/**********************************************/
/*                                          */
/*                                          */
/*    STDFORM.C -- Demonstrates the use     */
/*    of standard image data format, and    */
/*    the Transform Form function.          */
/*                                          */
/*                                          */
/**********************************************/

#include "shell.c"

int imagedat[] = {
    0x00FF, 0x00FF, 0x00FF, 0x00FF,0x00FF, 0x00FF, 0x00FF, 0x00FF,
    0x00FF, 0x00FF, 0x00FF, 0x00FF,0x00FF, 0x00FF, 0x00FF, 0x00FF,
    0x00FF, 0x00FF, 0x00FF, 0x00FF,0x00FF, 0x00FF, 0x00FF, 0x00FF,
    0x00FF, 0x00FF, 0x00FF, 0x00FF,0x00FF, 0x00FF, 0x00FF, 0x00FF,

    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,

    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,

    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
    };


demo()
{
    struct fdbstr
    {
      int   *image;    /* memory pointer */
      int   width;     /* form width in pixels */
      int   height;    /* form height */
      int   wordw;     /* form width in words */
      int   flag;      /* form flag */
      int   planes;    /* number of color planes */
      int   r1, r2, r3;
    }srcMFDB,screen;

    int points[8];
    int step,scrh,scrw,c;

    scrw = work_out[0]/32;  /* find screen width */
    scrh = work_out[1]/16;  /* and height */
    step = (scrw < scrh) ? scrw : scrh;
/* Use vq_extnd to find # of bit planes */

    vq_extnd(handle,1,work_out);

/* Set up a destination form using screen data,
   and a source form using a memory buffer */

    screen.image = 0L;         /* use screen data for srcMFDB */
```

130

```
    srcMFDB.image = imagedat;
    srcMFDB.width = 32;
    srcMFDB.height = 16;
    srcMFDB.wordw = 2;
    srcMFDB.flag = 1;              /* standard form */
    srcMFDB.planes = work_out[4];

 vr_trnfm(handle,&srcMFDB,&srcMFDB); /* in place transform */

 /* set up initial screen points for the image */

    points[0]=points[4]=points[1]=points[5]=0;
    points[2]=points[6]=31;
    points[3]=points[7]=15;

/* Copy the colored box from the memory form to the screen
   repeatedly in a diagonal line */

    for(c=0;c<step;c++)
        {
        vro_cpyfm(handle,3,points,&srcMFDB, &screen);
        points[5]+=16;
        points[7]+=16;
        points[4]+=32;
        points[6]+=32;
        }

 }
       /* End of stdform.c */
```

Copy Raster Transparent

The last raster function is Copy Raster Transparent. This oper-
ation copies a source form that only has one bit plane of im-
age data into a destination form that can have several color
planes. Since a single bit plane image requires the least
amount of image data, this is a very economical way of plac-
ing an image on the screen. In fact, it's the method used by
GEM for drawing icons on the screen. Since this call allows
you to specify the pen color that will be used to draw both the
foreground (one bits) and the background (zero bits), the im-
age can be drawn in any color combination that you wish. The
C language syntax for this call is

int handle, mode, points[8],pens[2];
struct fdbstr *srcMFDB, *destMFDB;

vrt_cpyfm(handle, mode, points, srcMFDB, destMFDB, pens);

The mode value used for this call is the same as that
used by vswr_mode( ), not the more complex one used by
vro_cpyfm( ). As you may recall, the four writing modes are:

Mode **Description**
  1   Replace
  2   Transparent
  3   XOR
  4   Reverse Transparent

The parameters srcMFDB and destMFDB are pointers to the source and destination forms. The source form must contain only a single bit plane, and it's irrelevant whether it's in standard or ST-specific format, since both are the same for a monochrome image. The destination format may contain 1, 2, or 4 bit-planes, and should be in ST-specific format. A screen form, the most common destination, is already in that format.

The points array is the same as that used by vro_cpyfm( ). The first four elements describe size and position of the source rectangle, and the last four describe the destination rectangle. The rectangles are offset from the top left corner of the forms, and the source and destination rectangles should be the same size.

The value that pens points to is an array which holds two pen numbers. The first, pens[0], contains the pen number of the foreground color which will be drawn wherever there is a one bit in the source image. The other, pens[1], contains the pen number of the background color which is drawn wherever there is a zero bit in the source image. Note that these are the VDI pen numbers (color index), not the actual hardware register numbers formed by the various bit combinations. These colors will be translated to the appropriate bit combinations when the single plane image is expanded to the number of planes used by the screen. The resulting expanded image is then combined with the destination image, bit plane by bit plane, according to the logic operation chosen.

Program 6-3 copies a predefined image to the screen, using the XOR mode. It demonstrates a simple form of animation.

Notice that in order to have the image appear in the same size and aspect ratio in all three resolution modes, we had to provide three different arrays of image data. This is admittedly a lot of data to type in by hand. Fortunately, some of the painting programs available for the ST, such as Neochrome and DEGAS Elite, allow you first to create an image by drawing with the mouse and then to save that image to a text file in the form of C source code. This source code file is in the

form of an initialized array that can be merged into your program file.

You may have noticed the use of the XBIOS call Vsync. This is used to combat the flickering that can occur when you move an image on screen while the display is being redrawn. The Vsync call pauses the program until the the vertical retrace interval occurs. That's when the video beam reaches the bottom of the picture and shuts off until it gets back up to the top. This allows you to change the image in display memory when the display is not being changed.

Program 6-3. copytran.c

```
/**********************************************/
/*                                          */
/*                                          */
/*    COPYTRAN.C -- Demonstrates the Copy     */
/*    Tranparent function, including some     */
/*    animation using the XOR copy mode.      */
/*                                          */
/*                                          */
/**********************************************/

#include "shell.c"
#include <osbind.h>

#define WHITE 0
#define BLACK 1
#define RED   2
#define GREEN 3

    struct fdbstr
    {
      int  *image;    /* memory pointer */
      int  width;     /* form width in pixels */
      int  height;    /* form height */
      int  wordw;     /* form width in words */
      int  flag;      /* form flag */
      int  planes;    /* number of color planes */
      int  r1, r2, r3;
    }screen,view[2]; /* forms for the screen,
                        and 2 views of the image */

int colors[2]={BLACK,WHITE}; /* XOR does use colors, but you
                                still must set up the array */

demo()
{
    int points[8];
    int scrh,scrw,
        c,d,e;

    scrw = work_out[0];  /* find screen width */
    scrh = work_out[1];  /* and height */

/* put a patterned block in mid-screen */

    points[0] = scrw*3/8;
    points[1] = scrh*5/8;
```

```
    points[2]= scrw*5/8;
    points[3] = scrh*7/8;
    vsf_interior(handle, 3);
    vsf_style(handle,3);
    vsf_color(handle, GREEN);
    v_bar(handle,points);

/* initialize bug creature forms according to resolution  */

    switch(work_out[13]) /* find out how many colors */
    {
    case 2:  inithi();   /* if 2 colors, use hi-res */
             break;

    case 4:  initmed();  /* if 4 colors, use med-res */
             break;

    case 16: initlo();   /* if 16 colors, use lo-res */
    }

/* Set up one form using screen data,
   and 2 using image data from an array */

    screen.image = 0L;          /* use screen data for this form */
    view[0].flag = view[1].flag = 0; /* views are ST-specific form */
    view[0].planes = view[1].planes = 1; /* only 1 plane */

 /* set initial locations for source and dest. rectangles */

   points[0]= points[4] = points[1] = 0;
   points[2]= points[6] = view[0].width-1;
   points[3]= view[0].height-1;
   points[5]= scrh*3/4;
   points[7]= (scrh*3/4) + view[0].height-1;

 /* draw the first bug in XOR mode */

   vrt_cpyfm(handle,3,points,&view[0], &screen,colors);


 /* copy and erase the view forms to the screen,
    alternately using XOR mode to animate them */


    for(c=0;c<scrw/7;c++) /* repeat across the screen */
       {
       for(d=0;d<2;d++)    /* alternate image each time */
          {
          for (e=0;e<10000;e++); /* add a delay so we can see */
          Vsync();                /* sync with vertical retrace
                                     to minimize flicker */
          /* erase the last one */
          vrt_cpyfm(handle,3,points,&view[d], &screen,colors);
          points[4]+=3; /* move it horizontally */
          points[6]+=3;
          /* and draw the next one */
          vrt_cpyfm(handle,3,points,&view[d^1], &screen,colors);
          } /* end of d loop */
       }    /* end of c loop */

 }          /* end of demo() */

/* Support routines to initialize form data,
   depending on the resolution mode in effect */

 initlo()
 {
```

```
static int lo_image[2][46] ={
  0x0030, 0x0C00, 0x001C, 0x3800,
  0x0006, 0x6000, 0x0006, 0x6000,
  0x001F, 0xF800, 0x003F, 0xFC00,
  0xC0FF, 0xFF03, 0xC0FF, 0xFF03,
  0xE3E3, 0xC7C7, 0x7FEB, 0xD7FE,
  0x3FE7, 0xCFFC, 0x03FF, 0xFFC0,
  0x03FF, 0xFFC0, 0x00F8, 0x1F00,
  0x00FC, 0x3F00, 0x00FF, 0xFF00,
  0x0077, 0xEE00, 0x0030, 0x0C00,
  0x0030, 0x0C00, 0x0030, 0x0C00,
  0x0060, 0x0600, 0x00C0, 0x0300,
  0x0380, 0x01C0,

  0x0000, 0x0000, 0x001C, 0x3800,
  0x0036, 0x6C00, 0x0006, 0x6000,
  0x001F, 0xF800, 0x003F, 0xFC00,
  0x00FF, 0xFF00, 0x00FF, 0xFF00,
  0x03E3, 0xC7C0, 0x3FEB, 0xD7FC,
  0x7FE7, 0xCFFE, 0x63FF, 0xFFC6,
  0xC3FF, 0xFFC3, 0xC0F8, 0x1F03,
  0x00FC, 0x3F00, 0x00FF, 0xFF00,
  0x0077, 0xEE00, 0x0030, 0x0C00,
  0x0030, 0x0C00, 0x0030, 0x0C00,
  0x0018, 0x1800, 0x000C, 0x3000,
  0x0006, 0x6000
  };

view[0].image = lo_image[0];
view[1].image = lo_image[1];
view[0].width = view[1].width = 32;
view[0].height = view[1].height = 23;
view[0].wordw = view[1].wordw = 2;
}


initmed()
{
static int med_image[2][92] = {
  0x0000, 0x0F00, 0x00F0, 0x0000,
  0x0000, 0x03F0, 0x0FC0, 0x0000,
  0x0000, 0x003C, 0x3C00, 0x0000,
  0x0000, 0x003C, 0x3C00, 0x0000,
  0x0000, 0x03FF, 0xFFC0, 0x0000,
  0x0000, 0x0FFF, 0xFFF0, 0x0000,
  0xF000, 0xFFFF, 0xFFFF, 0x000F,
  0xF000, 0xFFFF, 0xFFFF, 0x000F,
  0xFC0F, 0xFC0F, 0xF03F, 0xF03F,
  0x3FFF, 0xFCCF, 0xF33F, 0xFFFC,
  0x0FFF, 0xFC3F, 0xF0FF, 0xFFF0,
  0x000F, 0xFFFF, 0xFFFF, 0xF000,
  0x000F, 0xFFFF, 0xFFFF, 0xF000,
  0x0000, 0xFFC0, 0x03FF, 0x0000,
  0x0000, 0xFFF0, 0x0FFF, 0x0000,
  0x0000, 0xFFFF, 0xFFFF, 0x0000,
  0x0000, 0x3F3F, 0xFCFC, 0x0000,
  0x0000, 0x0F00, 0x00F0, 0x0000,
  0x0000, 0x0F00, 0x00F0, 0x0000,
  0x0000, 0x0F00, 0x00F0, 0x0000,
  0x0000, 0x3C00, 0x003C, 0x0000,
  0x0000, 0xF000, 0x000F, 0x0000,
  0x000F, 0xC000, 0x0003, 0xF000,

  0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x03F0, 0x0FC0, 0x0000,
  0x0000, 0x0F3C, 0x3CF0, 0x0000,
  0x0000, 0x003C, 0x3C00, 0x0000,
```

```
    0x0000, 0x03FF, 0xFFC0, 0x0000,
    0x0000, 0x0FFF, 0xFFF0, 0x0000,
    0x0000, 0xFFFF, 0xFFFF, 0x0000,
    0x0000, 0xFFFF, 0xFFFF, 0x0000,
    0x000F, 0xFC0F, 0xF03F, 0xF000,
    0x0FFF, 0xFCCF, 0xF33F, 0xFFF0,
    0x3FFF, 0xFC3F, 0xF0FF, 0xFFFC,
    0x3C0F, 0xFFFF, 0xFFFF, 0xF03C,
    0xF00F, 0xFFFF, 0xFFFF, 0xF00F,
    0xF000, 0xFFC0, 0x03FF, 0x000F,
    0x0000, 0xFFF0, 0x0FFF, 0x0000,
    0x0000, 0xFFFF, 0xFFFF, 0x0000,
    0x0000, 0x3F3F, 0xFCFC, 0x0000,
    0x0000, 0x0F00, 0x00F0, 0x0000,
    0x0000, 0x0F00, 0x00F0, 0x0000,
    0x0000, 0x0F00, 0x00F0, 0x0000,
    0x0000, 0x03C0, 0x03C0, 0x0000,
    0x0000, 0x00F0, 0x0F00, 0x0000,
    0x0000, 0x003C, 0x3C00, 0x0000
    };

view[0].image = med_image[0];
view[1].image = med_image[1];
view[0].width = view[1].width = 64;
view[0].height = view[1].height = 23;
view[0].wordw = view[1].wordw = 4;
}

inithi()
{
static int hi_image[2][0xB8] ={
    0x0000, 0x0F00, 0x00F0, 0x0000,
    0x0000, 0x0F00, 0x00F0, 0x0000,
    0x0000, 0x03F0, 0x0FC0, 0x0000,
    0x0000, 0x03F0, 0x0FC0, 0x0000,
    0x0000, 0x003C, 0x3C00, 0x0000,
    0x0000, 0x003C, 0x3C00, 0x0000,
    0x0000, 0x003C, 0x3C00, 0x0000,
    0x0000, 0x003C, 0x3C00, 0x0000,
    0x0000, 0x03FF, 0xFFC0, 0x0000,
    0x0000, 0x03FF, 0xFFC0, 0x0000,
    0x0000, 0x0FFF, 0xFFF0, 0x0000,
    0x0000, 0x0FFF, 0xFFF0, 0x0000,
    0xF000, 0xFFFF, 0xFFFF, 0x000F,
    0xF000, 0xFFFF, 0xFFFF, 0x000F,
    0xF000, 0xFFFF, 0xFFFF, 0x000F,
    0xFC0F, 0xFC0F, 0xF03F, 0xF03F,
    0xFC0F, 0xFC0F, 0xF03F, 0xF03F,
    0x3FFF, 0xFCCF, 0xF33F, 0xFFFC,
    0x3FFF, 0xFCCF, 0xF33F, 0xFFFC,
    0x0FFF, 0xFC3F, 0xF0FF, 0xFFF0,
    0x0FFF, 0xFC3F, 0xF0FF, 0xFFF0,
    0x000F, 0xFFFF, 0xFFFF, 0xF000,
    0x000F, 0xFFFF, 0xFFFF, 0xF000,
    0x000F, 0xFFFF, 0xFFFF, 0xF000,
    0x000F, 0xFFFF, 0xFFFF, 0xF000,
    0x0000, 0xFFC0, 0x03FF, 0x0000,
    0x0000, 0xFFC0, 0x03FF, 0x0000,
    0x0000, 0xFFF0, 0x0FFF, 0x0000,
    0x0000, 0xFFF0, 0x0FFF, 0x0000,
    0x0000, 0xFFFF, 0xFFFF, 0x0000,
    0x0000, 0xFFFF, 0xFFFF, 0x0000,
    0x0000, 0x3F3F, 0xFCFC, 0x0000,
    0x0000, 0x3F3F, 0xFCFC, 0x0000,
    0x0000, 0x0F00, 0x00F0, 0x0000,
    0x0000, 0x0F00, 0x00F0, 0x0000,
```

```
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x3C00, 0x003C, 0x0000,
0x0000, 0x3C00, 0x003C, 0x0000,
0x0000, 0xF000, 0x000F, 0x0000,
0x0000, 0xF000, 0x000F, 0x0000,
0x000F, 0xC000, 0x0003, 0xF000,
0x000F, 0xC000, 0x0003, 0xF000,

0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x03F0, 0x0FC0, 0x0000,
0x0000, 0x03F0, 0x0FC0, 0x0000,
0x0000, 0x0F3C, 0x3CF0, 0x0000,
0x0000, 0x0F3C, 0x3CF0, 0x0000,
0x0000, 0x003C, 0x3C00, 0x0000,
0x0000, 0x003C, 0x3C00, 0x0000,
0x0000, 0x03FF, 0xFFC0, 0x0000,
0x0000, 0x03FF, 0xFFC0, 0x0000,
0x0000, 0x0FFF, 0xFFF0, 0x0000,
0x0000, 0x0FFF, 0xFFF0, 0x0000,
0x0000, 0xFFFF, 0xFFFF, 0x0000,
0x0000, 0xFFFF, 0xFFFF, 0x0000,
0x0000, 0xFFFF, 0xFFFF, 0x0000,
0x0000, 0xFFFF, 0xFFFF, 0x0000,
0x000F, 0xFC0F, 0xF03F, 0xF000,
0x000F, 0xFC0F, 0xF03F, 0xF000,
0x0FFF, 0xFCCF, 0xF33F, 0xFFF0,
0x0FFF, 0xFCCF, 0xF33F, 0xFFF0,
0x3FFF, 0xFC3F, 0xF0FF, 0xFFFC,
0x3FFF, 0xFC3F, 0xF0FF, 0xFFFC,
0x3C0F, 0xFFFF, 0xFFFF, 0xF03C,
0x3C0F, 0xFFFF, 0xFFFF, 0xF03C,
0xF00F, 0xFFFF, 0xFFFF, 0xF00F,
0xF00F, 0xFFFF, 0xFFFF, 0xF00F,
0xF000, 0xFFC0, 0x03FF, 0x000F,
0xF000, 0xFFC0, 0x03FF, 0x000F,
0x0000, 0xFFF0, 0x0FFF, 0x0000,
0x0000, 0xFFF0, 0x0FFF, 0x0000,
0x0000, 0xFFFF, 0xFFFF, 0x0000,
0x0000, 0xFFFF, 0xFFFF, 0x0000,
0x0000, 0xFFFF, 0xFFFF, 0x0000,
0x0000, 0x3F3F, 0xFCFC, 0x0000,
0x0000, 0x3F3F, 0xFCFC, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x0F00, 0x00F0, 0x0000,
0x0000, 0x03C0, 0x03C0, 0x0000,
0x0000, 0x03C0, 0x03C0, 0x0000,
0x0000, 0x00F0, 0x0F00, 0x0000,
0x0000, 0x00F0, 0x0F00, 0x0000,
0x0000, 0x003C, 0x3C00, 0x0000,
0x0000, 0x003C, 0x3C00, 0x0000
};

view[0].image = hi_image[0];
view[1].image = hi_image[1];
view[0].width = view[1].width = 64;
view[0].height = view[1].height = 46;
view[0].wordw = view[1].wordw = 4;
}
        /* End of Copytran.c */
```

Raster Operations In BASIC

The original version of ST BASIC did not contain any key-word support for raster operations. The revised version is slated to include the commands SSHAPE and GSHAPE. These commands function much like the GET and PUT statements in Microsoft BASIC. To move an image from the screen to a memory array, you use the command SSHAPE, whose syntax is

SSHAPE x1,y1;x2,y2, array%()

where *x1,y1*, and *x2,y2* are the coordinates for the top left and bottom right corners of the image rectangle. The *array%( )* parameter is an integer array that has been DIMensioned so it is large enough to contain the image information. For each bit plane, you'll need one word of data for every 16 bits of width, times the number of lines of height. Let's take the example of a rectangle that goes from 32,56 to 194,98. Since this rectangle is 163 pixels wide (194−32+1), the smallest number of words that can hold a line of data is 11 (11*16 = 176). There are 43 lines of data (98−56+1), so each bit plane requires 473 words of data. There are two bit planes in medium-res and four bit planes in lo-res, so these modes require 946 or 1892 words of data, respectively. To these size requirements you must add a few words in which to store the layout of the image. (At the time of this writing the exact number is not known, but 10 words should give you a comfortable margin.) Remember to DIMension the array to the correct size before you use SSHAPE.

To copy the image that you've saved from the array to the screen, you must use the GSHAPE command. The syntax for this command is
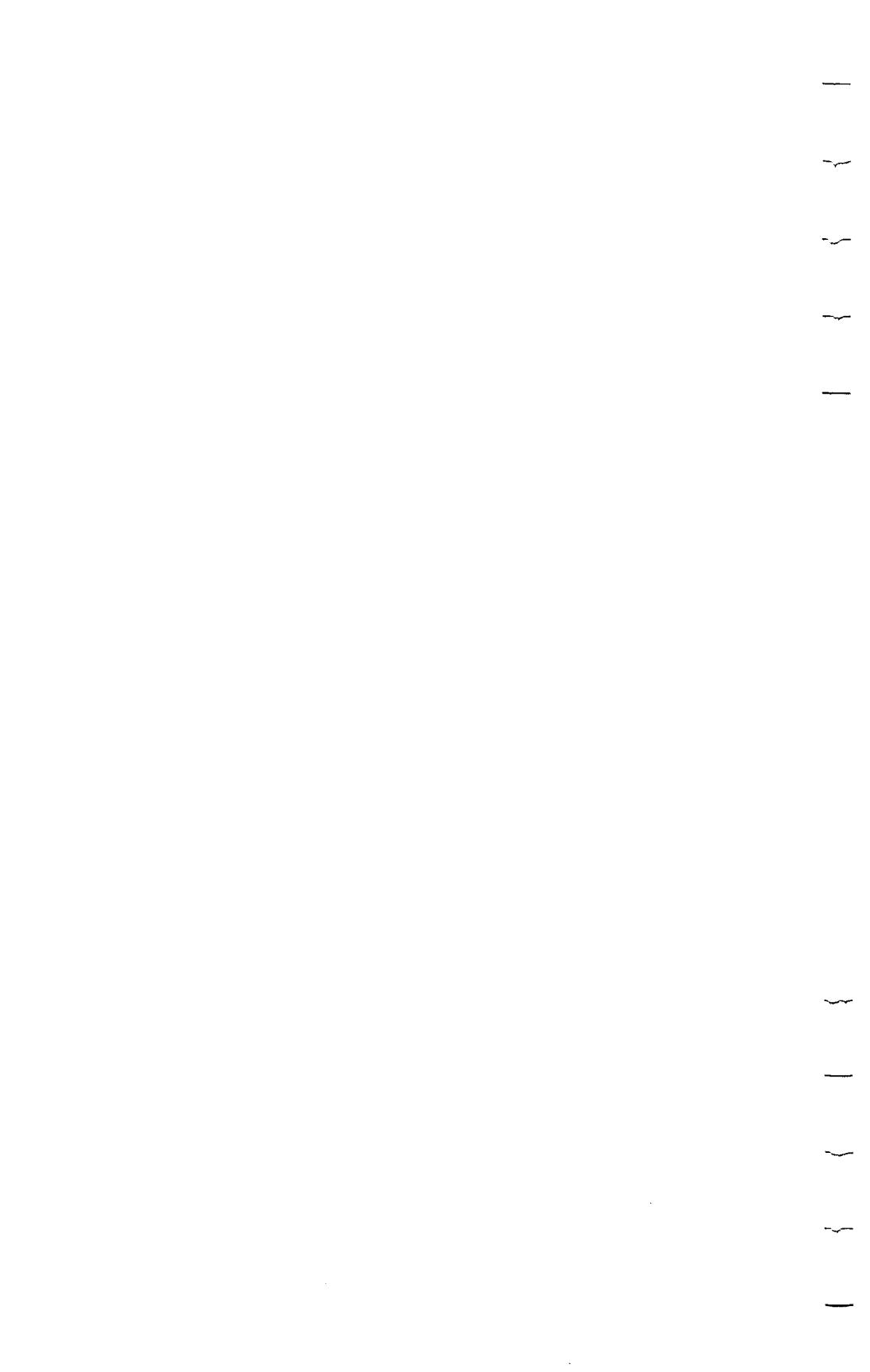
GSHAPE x1,y1,array%( )

where the point *x1,y1* describes the upper left corner of the image, and *array%( )* is the name of the array that's used to store the image. The size of the image is also taken from the array in which the data is stored.

As with the VDI raster commands, images that are copied to the screen with GSHAPE may be combined with the existing screen image in several ways. The command used to set the drawing mode also determines the copy mode for the image. This command is DRAWMODE mode, where mode is a number from 1 to 4:
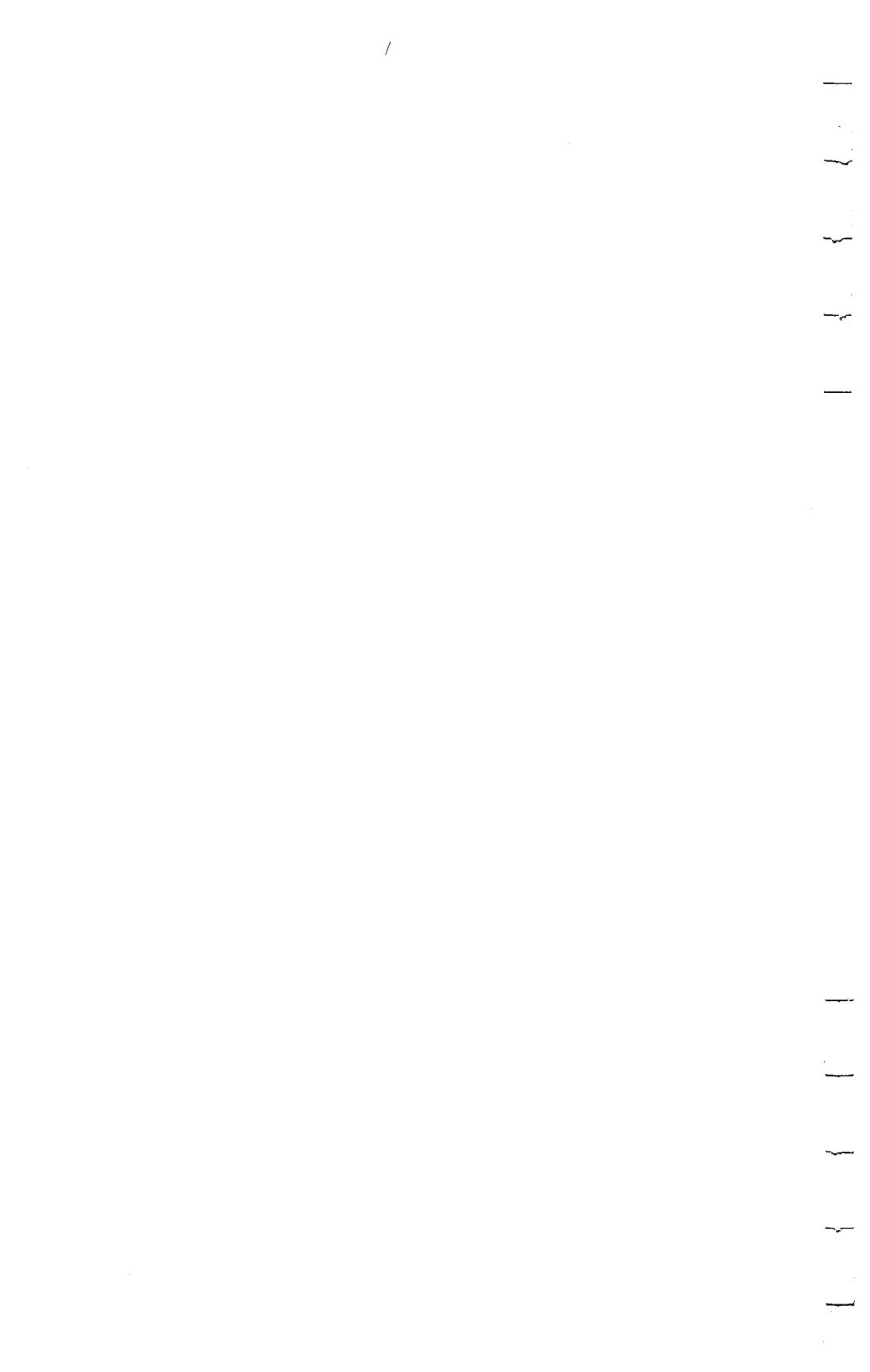
Mode Number	Description
1	Replace
2	Transparent
3	XOR
4	Reverse Transparent

Chapter 7

Text

We normally don't think of the text that appears on a computer screen as graphics, but there is actually very little difference between text and any other kind of graphics that can be displayed on the ST computers. Since the graphics display is bitmapped, text characters must be drawn on the screen dot-by-dot, just like any other kind of image. That's why the most common kind of text rendering under the VDI is called *graphics text*.

You will find that graphics settings like the clipping rectangle and the drawing mode, which apply to all graphics output, affect graphics text just as they do any other drawing operation. In addition, there is a collection of settings that apply only to graphics text. These allow you to control the color, typeface, size, rotation, and positioning of graphics text.

There are many advantages to using this system of text rendering on a computer. Characters may be placed anywhere on the screen, graphics and text may be mixed freely, and different sizes and styles of lettering may be used at the same time. It's even possible to use some of the refinements common to the printing world, like proportional fonts and microspace justification.

There are performance tradeoffs, however, with using a bitmapped screen for text, rather than a character display. Text rendering is a bit slower than with a character display, and may be noticeably slower when a whole screen full of text is being scrolled, for instance. This is particularly true when the characters are not positioned so that the image data for each falls within even byte boundaries, as is the case with a proportional font. When the left half of a character falls within one byte of screen memory, and the right half within another, the VDI must shift the image data twice before writing it to the screen.

Another drawback of the graphics text functions is that, although they give you complete control over the placement and appearance of text output, they require you to place each line of text on the screen individually. Since such complete

control is not always necessary (or even desirable, given the
performance penalty for text that does not fall within even
byte boundaries), the VDI also provides a more traditional text
mode called *alphanumeric mode.*

The alphanumeric mode behaves like a conventional text
terminal, in which characters, in a fixed type style and size,
are printed one after the other in fixed positions on the screen.
It allows you to output text conveniently, without worrying
you about all the details of its position and appearance. The
alphanumeric mode also supports many of the features of text
display terminals, such as cursor movement and absolute po-
sitioning, reverse video, and selective erasure.

While this book deals only with the VDI functions of the
GEM operating system, you should be aware that other por-
tions of the ST system software also offer text support. The
AES portion of GEM contains some functions that deal with
text manipulation and input, and GEMDOS and the ST BIOS
both contain support for a text console device. C programmers
should note that the standard C text functions such as printf( )
may also be used.

Graphics Text and Text Alignment
The basic function for the display of text is called Text, and it's
used like this:

```
int handle, x, y;
char *string;
v_gtext(handle, x, y, string);
```

where x and y are the text placement coordinates, and *string* is
a pointer to a null-terminated character string. This is an array
of ASCII character values, ending with a character whose
ASCII value is 0.

Just saying that x and y are the text placement coordi-
nates, however, doesn't really answer the question of where
the text will appear on the screen.
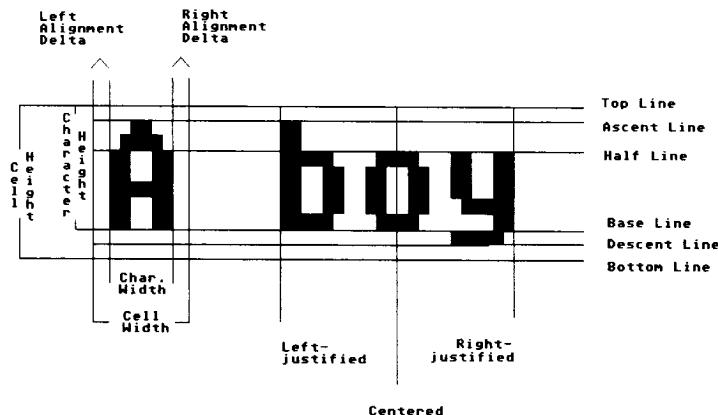
Since a line of text may cover an area containing thou-
sands of pixels, we must have some way of determining which
point the placement coordinates describe. The GEM VDI rec-
ognizes several parts of the text display as significant, and al-
lows you to align your text display with any of these points.

In order to understand how text alignment works, you
must be familiar with the parts of the character display. Each

text character occupies a space known as a *cell*. This includes not only the image data for the character, but also some blank space surrounding the character. The top and bottom of the cell are delimited by imaginary lines called the *top* and *bottom* lines. Toward the lower part of the cell, there's an imaginary line called the *baseline*. This line marks the bottom of most characters, such as the letter *o*. Some characters, like the letters *p* and *y*, extend below the baseline, down to what is called the *descent* line. The upper boundary for most of the lowercase letters is called the *half* line. The capital letters, and some lowercase letters like *b*, extend upwards past the half line, to what is known as the *ascent* line. Note that there may or may not be some blank space between the ascent line and top line, or descent line and bottom line, depending on the particular typeface. In the standard Atari system font, for example, characters extend to the top and the bottom of the cell. The various vertical alignment points are illustrated in Figure 7-1.

Figure 7-1. Vertical Alignment of Text

In addition to the vertical alignment points described above, a string of text has three significant horizontal alignment points. You may line up the string so that its left side, right side, or center is even with the text placement point. The VDI call used to select the horizontal and vertical alignment points is Set Graphics Text Alignment. The C syntax for this call is

int handle, halign, valign, hset, vset
vst_alignment(handle, halign, valign, &hset, &vset);

where *halign* and *valign* show the horizontal and vertical alignment points that you wish to set. Since every device does not support this type of alignment, the actual alignment points that were set by the call are returned in the variables *hset* and *vset*. The values in these variables are a numeric code that specifies the alignment points. For the vertical point, the possible values are:

Number	Description
0	Baseline (default)
1	Half line
2	Ascent line
3	Bottom line
4	Descent line
5	Top line

The possible horizontal alignment points are:

Number	Description
0	Left justified (default)
1	Centered
2	Right justified

By default, when you specify an *x* and *y* position for text in your v_gtext( ) call, that point defines the baseline position for the left side of the first character in your string. If the alignment is changed, however, that point can just as well specify the top right corner of the last character in the string, or the half line of the center of the string. Program 7-1 shows the effect of the alignment settings on text placement. It draws a horizontal and vertical line, and shows the possible placement of text strings whose placement coordinates are the same as those of the lines.

Program 7-1. align.c

```
/****************************************************/
/*                                              */
/*                                              */
/*    ALIGN.C -- Demonstrates alignment         */
/*    of graphics text strings.                 */
/*                                              */
/*                                              */
/*                                              */
/****************************************************/

#include "shell.c"

char *hstring[]=
{
  "Left justified",
  "Centered",
  "Right Justified"
```

```
};

char *vstring[]=
{
 "Base",
 "Half",
 "Ascent",
 "Bottom",
 "Descent",
 "Top"
};

demo()
{
int c,
    xmax,ymax,x,y,dx,dy,hset,vset,points[8];


    xmax = work_out[0];    /* screen width */
    ymax = work_out[1];    /* screen height */
    dy=(ymax+1)/25*2;      /* line height */
    y=32; x=8;             /* set default x and y */

    for (c=0;c<3;c++) /* for each horizontal position */
        {
/* Change Horiz. alignment and print text */
        vst_alignment(handle,c,0,&hset,&vset);
        v_gtext(handle,xmax/2,y+=dy,hstring[c]);
        }
    for (c=0;c<6;c++) /* for each vertical position */
        {
/* Change Vert. alignment and print text */
        vst_alignment(handle,0,c,&hset,&vset);
        v_gtext(handle,x,ymax*3/4,vstring[c]);
/* add string length to x position */
        vqt_extent(handle,vstring[c],points);
        x+=(points[2]+8);
        }

    vsl_color(handle,2);
/* Draw vertical alignment line */
    points[1]=points[3]=ymax*3/4;
    points[0]=0;
    points[2]=x+16;
    v_pline(handle,2,points);

/* Draw horizontal alignment line */
    points[0]=points[2]=xmax/2;
    points[1]=0;
    points[3]=y+16;
    v_pline(handle,2,points);
}

/* End of Align.c */
```

Microspace Justification

Some word processors allow you to stretch out a line of text to fill a certain line length on the paper by adding (or removing) minute spaces between characters or words. This feature, often referred to as microspace justification, is one of the more sophisticated text functions offered by the VDI. The name of this

function is Justified Graphics Text, and its calling sequence is as follows:

```
int x, y, length, word_space, char_space;
char *space;
v_justified(handle, x, y, string, length, word_space, char_space);
```

The string parameter points to the null-terminated character string to be printed, and the *x* and *y* parameters specify the display location, just as with the v_gtext( ) call. The *length* value specifies the size of the screen area used to display the text string (in whichever coordinate system, raster or normalized, that's currently in use). The final two parameters, *word_space* and *char_space*, are flags that tell the function how to manipulate the string to achieve the desired length. The *v_justified* function can adjust the space between characters in a word, the space between words in a line, or both. Setting either of the flags to 1 tells the function to adjust the spacing between the elements named, while setting either to 0 means that the spacing will not be affected for that element. If both of these values are set to 0, no justification is performed, and the text string is printed with the same spacing as would be used by v_gtext( ).

An example of justified text can be found in Program 7-2, rotext.c, in the section on rotated text, below.

Sizing a Text String

When printing graphics text, the programmer has total responsibility for the placement of the string. One of these duties includes making sure that the text string fits on the display screen. In order to do this, however, you must know how much size the text will occupy on the display. With monospaced fonts (like the standard ST font), each character cell is the same width, so you can multiply the width of each character by the number of characters to find out the total length of the string.

But GEM provides for proportionally spaced fonts as well. In a proportionally spaced font, a wide letter like the *w* occupies a wider cell than a narrow letter like the *l*, so there isn't a lot of extra white space on either side of the narrow letter. While this will make the text look more attractive, it makes it much harder to keep track of the overall length of the string.

The VDI does supply a function that lets you discover the the width of a particular character. It's called Inquire Character Cell Width, and its calling sequence goes like this:

```
int handle, char, cellw, ldelta, rdelta,status;
status = vqt_width(handle, char, &cellw, &ldelta, &rdelta);
```

where *char* is the ASCII value of the character in question, and *cellw* is the variable in which the width of the character cell (including spacing to the left and right of the character) is returned. The width of the blank space to the left and right of the character within the cell is stored in the variables *ldelta* and *rdelta*. Note that the width returned by this function does not account for any space added to the character because of special effects. Another function, vqt_fontinfo( ), returns information about the change in character width caused by special effects. Figure 7-1 shows the parts of the character cell.

Though it's possible to figure out the width of the line by adding the widths of the individual characters, there's an easier way to find it. The VDI provides a function which returns the size that a given text string drawn in the current font would occupy. This function is Inquire Text Extent. It's used like this:

```
int handle, points[8];
char *string;
vqt_extent(handle, string, points);
```
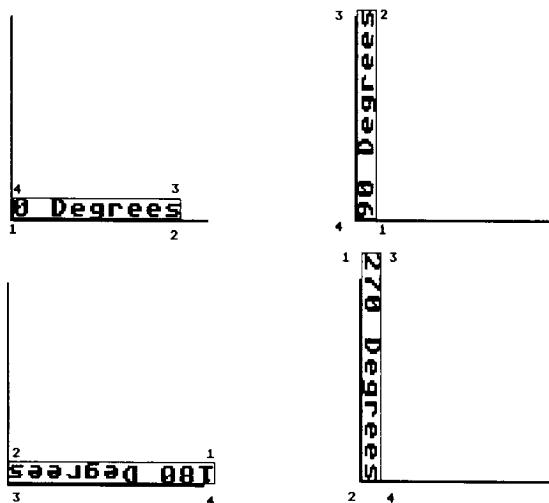
where *string* is a pointer to a null-terminated text string, and *points* is a pointer to an array which holds the coordinates for the four corner points of the smallest box that completely contains the text string. The first two elements give the x and y coordinates of the lower left corner of the text box.

The next two elements give the coordinates of the lower right corner. The third pair gives the location of the upper right corner. The last two give the x and y position of the upper left corner of the text box. When we refer to the lower left corner of the text box, we mean relative to a horizontal line of text. The reason that the VDI specifies all four corner points (instead of just two opposite corners, like most other graphics functions using rectangular areas) is that GEM provides for the display of text that has been rotated at angles other than right angles (though on the ST, only 90-degree rotation is supported).

Regardless of whether the virtual workstation uses raster or normalized coordinates, the points of the text box are given in a coordinate system which, like the normalized coordinate system, has it origin in the lower left corner. A text string that has not been rotated has its lower left corner (point 1) at the origin. One that has been rotated 90 degrees has its upper left corner (point 4) at the origin. Figure 7-2 shows the reference points described by the vqt_extent( ) function.

Figure 7-2. Rotating Text

With the first version of the TOS ROMs, when the text string is rotated 270 degrees, the coordinates for the first point (point[0] and point[1]) are not returned correctly by this function. Since the text box is always at right angles on the ST, however, you can use points 2 and 4 to describe the four sides of the text box.

Character Rotation

As we've mentioned before, one of the text settings that the VDI supports is character rotation. The full name of the VDI function is Set Character Baseline Vector. Normally the baseline for a character is horizontal, and extends from left to right. Another way of saying this is that the baseline vector is zero degrees. But GEM provides for rotation of the baseline as well.

In the case of the ST screen display, this rotation is in 90-degree increments only. This means that besides displaying the text normally, you can display text sideways or upside down and backwards as well. The syntax for the call used to make this setting is

int handle, angle, angle_set;
angle_set = vst_rotation(handle, angle);

where *angle* is the angle requested. Since not every angle is supported (on the ST screen, only even multiples of 90 degrees are available), the function returns the angle that was actually set in the variable *angle_set*. Remember, the VDI expresses angles in tenths of a degree (0–3600), starting with the rightmost point on the circle and moving counterclockwise.

Program 7-2 displays four text strings, rotated at right angles to one another. It also shows an example of justified text. Since pixels on the color screen tend to be taller than they are wide, particularly in medium-resolution mode, somewhat narrower spacing makes the text look more natural when turned sideways. The program also uses the vqt_extent function to determine the normal length of the string in order to shorten it somewhat.

Program 7-2. rotext.c

```
/***********************************************/
/*                                           */
/*                                           */
/*    ROTEXT.C -- Demonstrates rotation       */
/*    of graphics text strings.               */
/*                                           */
/*                                           */
/*                                           */
/***********************************************/

#include "shell.c"

demo()
{
    int midh,midv,len,box[8];

    midh = work_out[0]/2;
    midv = work_out[1]/2;

    v_gtext(handle,midh+8,midv,"0 Degree Rotation");

    vst_color(handle,4);
    vst_rotation(handle,1800);
    v_gtext(handle,midh-8,midv,"180 Degree Rotation");

    vst_color(handle,3);
    vst_rotation(handle,900);
```

151

```
vqt_extent(handle,"90 Degree",box);
len = box[3]-8;
v_justified(handle,midh,midv-16,"90 Degree",len,1,1);

vst_color(handle,2);
vst_rotation(handle,2700);
vqt_extent(handle,"270 Degree",box);
len = box[6]-8;
v_justified(handle,midh,midv+16,"2700 Degree",len,1,1);


}

/* End of Rotext.c */
```

Text Color

Just as with lines, markers, and filled shapes, graphics text has it own individually selectable color setting. The text color default is determined by the value placed in work_in[6] when the virtual workstation is open (which defaults to color 1, black, if the GDOS extension is not loaded). Afterwards, you may change the color index with the function Set Graphic Text Color Index, whose format is

int handle, pen, pen_set;
pen_set = vst_color(handle, pen);

where *pen* is the VDI pen color (color index) requested. Since each resolution mode has a different number of pens available, not every pen can be selected from every mode. Therefore, the function returns the number of the pen that was actually set in the variable *pen_set*.

Special Effects

Another feature that the VDI has borrowed from the word processing and printing fields is called *special effects*. These days, most word processing programs allow you to add emphasis to certain parts of your text by making characters appear in boldface, italics, or underlined type. In the same manner, the VDI can alter the image data of text characters according to a mathematical formula in order to change their appearance. The effects supported are Thickened characters (boldface), Light text (such as you see in a menu item that's been grayed out), Skewed text (italic) , Underlined text, and Outlined Characters. These effects may be used individually or combined with one another. Since the new characters are altered versions of the originals, they may not always be as legi-

ble. Also, you should keep in mind that adding effects can change the width spacing of characters. For this reason, you should try to print a whole line of text at a time if you're using special effects, so the VDI can adjust the spacing. Otherwise, you may find that the new text that you put down erases part of the existing text.

The function used to make these changes to the standard character set is called Set Graphic Text Special Effects. The C syntax for this call is

int handle, effects, effects_set;
effects_set = vst_effects(handle, effects);

where *effects* is a flag byte showing which of five different effects are set on or off. (GEM actually provides for six effects, but only five are supported on the ST display.) Because not every effect is available on every device, the function returns the settings it actually puts into effect in the variable *effects_set*.

The effects flag byte has six significant bits (five on the ST), each of which controls a different effect. For example, the first bit, Thickened, controls whether or not text will be printed in boldface. The decimal value of the first bit (bit 0) is 1, so adding a 1 to the effects flag turns on bold printing.

Since each effect has a different bit, the effects can be easily added to one another. For example, an effects value of 9 means that both underlining (8) and boldface (1) are turned on. Of course, the VDI manipulates the character image each time it adds an effect, so too many effects may detract from the appearance of your text. The effects that are controlled by the various flag bits are described in the chart below.

Bit	Value	Effect
0	1	Thickened (bold)
1	2	Light intensity (grayed or ghosted)
2	4	Skewed (italicized)
3	8	Underlined
4	16	Outlined
5	32	Shadowed (not available on ST)

As we mentioned above, adding special effects to a text font can change the width of the characters. Most of the functions that give you information about the width of the characters in the current font do not take special effects into account. To find out how the width of the current font has been altered

by special effects, use the vq_fontinfo function, which is detailed a bit later on in the chapter.

Program 7-3 demonstrates the use of special effects with graphics text. It produces two columns of text, one of which shows each effect separately, and one of which shows each new effect added to the previous ones.

Program 7-3. effects.c

```
/**********************************************/
/*                                            */
/*                                            */
/*    EFFECTS.C -- Demonstrates graphics      */
/*    text special effects.                   */
/*                                            */
/*                                            */
/*                                            */
/**********************************************/

#include "shell.c"
#define vqt_fontinfo vqt_font_info /* for Megamax only!!! */

char *string[]=
{
  "Thickened",
  "Lightened",
  "Skewed",
  "Underlined",
  "Outlined"
};

demo()
{
int c,
    maxc,minc,nul,height, y=24,effects=0,distances[5];

 /* find out default font height and double it */

    vqt_fontinfo(handle,&minc,&maxc,distances,&nul,&nul);
    height=2*distances[4]+1;
    vst_height(handle, height, &nul, &nul, &nul, &nul);

/* Print two columns of plain text */
    v_gtext(handle,16,y+=(height+=8),"Plain");
    v_gtext(handle,180,y,"Plain");

    for (c=0;c<5;c++) /* for each horizontal position */
      {
 /* show each effect separately */

        vst_effects(handle,1<<c);
        vst_color(handle,c+1);
        v_gtext(handle,16,y+=height,string[c]);

/* show each effect added to the last */

        vst_effects(handle,effects+=(1<<c));
        vst_color(handle,c+1);
        v_gtext(handle,180,y,string[c]);
      }

}
/* End of Effects.c */
```

154

Setting Character Height

One of the most significant variables that you can change through the text settings is the size of the text characters. The default character set on the ST may be printed in any one of six sizes, and additional disk-based fonts come in many sizes as well. You'll find that the VDI scales up each character set, so that it can print in both the original size and a version that's twice as large.

The VDI function that is used to change the size of the current text font measures font size in terms of the height of each character. This height is determined in one of two ways. The first is in absolute pixel height, as measured by the current raster or normalized coordinate system. The VDI function used to set character size on this basis is called Set Character Height, Absolute Mode. This function is called by a C program like this:

```
int handle, height, char_width, char_height,
    cell_width, cell_height;

vst_height(handle, height, &char_width, &char_height,
    &cell_width, &cell_height);
```

where *height* is the pixel height of the character set, as measured from the baseline to the top of the character cell.

If the character height that you request isn't available, the VDI sets the next smallest available height. The height of the actual character font that was set is returned in the variable *char_height*, and the height of the entire character cell for that font is returned in the variable *cell_height*. Likewise, the width of the font that was set is returned in *char_width*, and the width of the character cell is returned in *cell_width*. For monospaced fonts, the widths returned apply to every character in the set. For proportional fonts, the character and cell widths returned are those of the widest character in the set.

The VDI also allows you to set the character height in terms of printer points. The point is a common measurement of the height of type fonts, and is equal to 1/72 inch. When point sizes are used to set the character height, the height of the entire cell is measured, not just that of the character. The function used to set character height on the basis of point size is called Set Character Cell Height, Points Mode. Its C syntax looks like this:

```
int handle, point, char_height, char_width,
    cell_height, cell_width, point_set;

point_set = vst_point(handle, point, &char_width,
    &char_height, &cell_width, &cell_height);
```

where *point* is the point size of the character font requested.
This function, like the absolute mode function, returns the size
of the character and the character cell in pixel units on the
correct coordinate scale. These measurements are returned in
the variables *char_height, char_width, cell_height,* and
cell_width. If the function is unable to set a font of the size re-
quested, it sets the next smallest available font size. The point
size of the font that was actually set is returned in the variable
point_set.

On the ST, the default system font may be printed in the
following sizes:

Points	Char_height	Char_width	Cell_height	Cell_Width
20	27	14	32	16
18	13	14	16	16
16	9	10	12	12
10	13	7	16	8
9	6	7	8	8
8	4	5	6	6

The three larger fonts are just enlargements of the three
smaller fonts. The ten-point font (16 × 8 cell) is the default
font on the monochrome screen, while the nine-point font (8
× 8 cell) is the default for both color modes. The eight-point
font (6 × 6 cell) is used for the lettering that appears under
icons on the GEM Desktop.

Using Disk-Based Fonts

One of the most important features added by the GDOS ex-
tensions that you load by running GDOS.PRG is the ability to
use disk-based fonts in addition to the normal system font. In
order to access these fonts, several requirements must be met.
First, the GDOS extensions must be loaded. (A message like
Atari GDOS ver. 1.1 resident will appear on the screen if the
GDOS has been successfully loaded.) Next, there must be one
or more font files on the disk from which you load the GDOS
(typically the disk that you boot up with runs this program
from the AUTO folder). These font files may be included with

software that you have purchased, or you may create them yourself with a font-creation program like the one supplied with *DEGAS Elite.* Finally, your startup disk must have a file in the root directory called assign.sys. This file lists the filename of each font that's available for each screen resolution mode. Complete details of the format of this assign.sys file appear in Chapter 2.

Once these three conditions have been met, you may load the additional fonts from disk with the call Load Fonts. Keep in mind that fonts take up a certain amount of memory space, and that if you load several sizes of a given font, particularly large sizes, that font may occupy as much as 32K of memory or more. You call Load Fonts like this:

```
int handle, select, fonts_added;
fonts_added = vst_load_fonts(handle, select);
```

The *select* parameter is reserved for future use, and should be set to 0. Currently, it's not possible to select which fonts you wish to load—it's an all-or-nothing proposition. When you use the *vst_load_fonts( )* function, all of the fonts that are specified in the assign.sys file for use by the device pointed to by *handle* are loaded in at once. The number of additional fonts that have been made available to the system is returned in the variable *fonts_added.* If fonts have already been loaded, nothing will happen if you try to load them again during the execution of the same application, and a 0 will be returned in fonts_added.

Once you've loaded the disk-based fonts, you'll naturally want to know which ones are available. The VDI function Inquire Face Name and Index returns information about a font's name and ID number. The format for this function is

```
int handle, font_num, font_id;
char name[32];
font_id = vqt_name(handle, font_num, name);
```

where *font_num* is a number that's assigned to each font when it is loaded into the system. Since the system font reserves for itself font_num 1 (and font_id 1), numbering of disk-based fonts begins with 2. The variable *fonts_added* contains the total number of fonts loaded by the vst_load_fonts( ) call, so the number that you pass in font_num should always be a value between 1 and fonts_added + 1. Two items of information are returned by this function. The first is the name of

the font, which is returned in the string pointed to by name. This string contains a maximum of 32 ASCII characters, the first 16 of which contain the name of the font, and the last 16 of which contain a modifier that describes the style and thickness of the characters. The other item is the font ID number. This is an identifier which is contained in the first two bytes of the font file, and which should be unique for every font named in your assign.sys file. You will need to know this number in order to set this font as the current typeface.

Set Text Face

Once you know the name and font IDs of all of the available fonts, you may set one of them as the current text font. The VDI call used for this function is Set Text Face, and it's called like this:

int handle, font_id, font_set;
font_set = vst_font(handle, font_id);

where *font_id* is the unique number contained in the first two bytes of the font file, which identifies this font. If the system can't load the font number that you requested, you can find out about it by checking the variable *font_set*, in which the font_id number of the text font that was actually set is returned.

Unload Fonts

If you've used the vst_load_fonts function to load in a number of disk-based fonts, you should call the Unload Fonts function before your program ends, to let the VDI know that you are no longer using them. If you don't, the system may crash when the next application tries to load them. It may also be desirable at times to call this function before the program ends, since if no other process (like a desk accessory) is using the fonts, the VDI frees up the memory that the software fonts occupied. The Unload Fonts function may be called like this:

int handle, select;
vst_unload_fonts(handle, select);

where the *select* parameter should be set to 0. At some point in the future, this parameter may be used to selectively unload fonts, but, for now, all fonts are unloaded at the same time, just as they are all loaded at the same time.

Program 7-4 shows all of the text fonts available in the system, in all of the point sizes. (This program requires that gdos.prog and assign.sys be present; see above and chapter 2 for more details.)

Program 7-4. diskfont.c

```
/**********************************************/
/*                                            */
/*                                            */
/*    DISKFONT.C -- Demonstrates use of       */
/*    additional text fonts loaded from       */
/*    disk.                                    */
/*                                            */
/**********************************************/

#include <osbind.h>

/* Global variables -- For VDI bindings, etc. */

int contrl[12],
    intin[128],
    ptsin[128],
    intout[128],
    ptsout[128];

int handle;

int work_in[12],
    work_out[57];


/* Initialization starts here */

main()
{
    int x, nul, button=0,
    addfonts,c,d,id,askd,recd,y,
    charh,charw,cellh,cellw,
    points[4];
    char name[32], string[80];

/* Initialize the GEM application */

    appl_init();

/* Initialize input array, get the physical workstation handle,
   and open the Virtual Screen Workstation */

    for (x=1, work_in[10]=2; x<10; work_in[x++]=1);
    work_in[0]=Getrez()+2;
    handle = graf_handle(&nul, &nul, &nul, &nul);
    v_opnvwk (work_in, &handle, work_out);
    v_clrwk(handle);

/* Set Clipping Rectangle */

    points[0]=points[1]=0;
    points[2]=work_out[0];
    points[3]=work_out[1];
    vs_clip(handle,1,points);

/* Load fonts and display each point size */
```

```
    addfonts = vst_load_fonts(handle,0);
    for (c=1;c<addfonts+2;c++)
        {
        id = vqt_name(handle,c,name);
        printf("\033E\033b1%d system font(s), %d disk fonts\n",
        work_out[10],addfonts);
        printf("\Font %d, '%s', ID = %d\033b3\n",c,name,id);
        recd=999; askd=1000; y=24;
        vst_font(handle,id);
        while(askd>=recd)
            {
            askd=recd-1;
            recd=vst_point(handle,askd,&charw,&charh,&cellw,&cellh);
            y+=(charh+8);
            sprintf(string,"%d PTS. %dx%d CHAR  %dx%d CELL",
            recd,charw,charh,cellw,cellh);
            if (askd>=recd)v_gtext(handle,0,y,string);
            }
/* wait until user clicks the mouse button */

        while(button==0)vq_mouse(handle,&button,&nul,&nul);
        for(d=0;d<30000;d++)button=0;
        }


/* Unload fonts, close the virtual
    workstation, and exit from the application */

    vst_unload_fonts(handle,0);
    v_clsvwk(handle);
    appl_exit();
}

/* end of Diskfont.c */
```

Text Face Information and Text Setting

A couple of inquiry functions round out the set of VDI text
functions. The first is called Inquire Current Face Information,
and it supplies information about the current text font, such as
the minimum and maximum values of the ASCII characters
for which there is image data, the maximum character width,
the number of pixels added to that width by special effects,
and the spacing between the various vertical alignment points
(bottom, descent line, baseline, half line, ascent line, and top).
The format for this call is

int handle, minc, maxc, maxwidth,
    effects[3], distances[5];
vqt_fontinfo(handle, &minc, &maxc, distances,
    maxwidth, effects);

160

Note that in early versions of the bindings, this function is incorrectly referred to as vqt_font_info( ). In particular, *Megamax C* owners may discover that the linker can't find the function vqt_fontinfo( ). If this is the case, you use the statement

#define vqt_fontinfo vqt_font_info
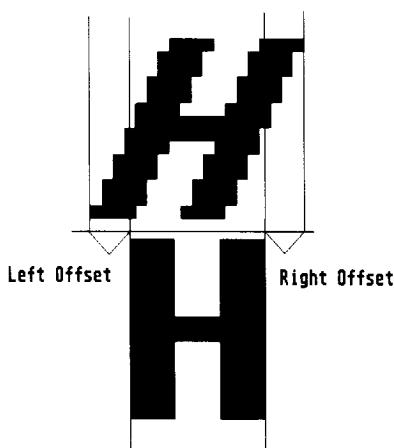
at the beginning of the program, as was done in Program 7-3, effects.c, above. Corrected versions of the bindings should be available from Megamax, as well.

This call returns the ASCII value of the first character for which there is image data in the character set in the variable *minc*, and the ASCII value of the last character in *maxc*. The variable *maxwidth* holds the maximum cell width, special effects not included. The *effects* array contains information about the adjustments that you must make to the character width to compensate for the current special effects. The contents of this array are interpreted as follows:

Element	Description
0	Total increase in character width due to effects
1	Left offset
2	Right offset

Figure 7-3. Right and Left Offset of Characters with Effects

Left Offset   Right Offset

The *distances* array contains information about the distances of the various vertical alignment points from the baseline. The organization of this array is shown below.

Element	Description
0	Bottom line to baseline
1	Descent line to baseline
2	Half line to baseline
3	Ascent line to baseline
4	Top line to baseline

The other inquiry function is called Inquire Current Graphic Text Attributes. It returns a wide variety of information about the current graphics text settings. This function is called as follows:

```
int handle, attributes[10];
vqt_attributes(handle, attributes);
```

where *attributes* is a pointer to an array of integers. The contents of that array is as follows:

Element	Description
0	Current text face
1	Text pen color
2	Angle of rotation (0–2700)
3	Horizontal alignment
4	Vertical alignment
5	Writing mode
6	Character width (in pixels)
7	Character height (in pixels)
8	Cell width (in pixels)
9	Cell height (in pixels)

Escapes and Alphanumeric Mode

The Escape function is used to access those capabilities of an output device which are peculiar to that device. In the case of the screen device, one of these capabilities is to print text in what is known as alphanumeric mode. This mode emulates the old-fashioned alphanumeric terminal display by dividing the screen into 25 imaginary rows and 80 imaginary columns (40 in lo res). These rows and columns define cells into which a text character may be placed. The alpha text functions ignore all of the current graphics text settings, and print characters only in the default size of the system text font, with no special effects, and no baseline rotation.

In alpha mode, a visible text cursor appears on the screen as a solid blinking box which marks the cell in which the next text character will be written. All of the normal rules of screen display scrolling apply. When a character is written to the last column in a row, the cursor moves down to the first column of the next row. (This wrap-around feature does not work correctly on 40-column screens.) When a character is written to the last column in the bottom row, all of the lines are scrolled up, and the cursor moves the first column of the blank line · that's inserted at the bottom of the page.

Some computers have separate graphics and character display modes. If you wish to use alphanumeric mode output with such computers, you must first switch their screen displays to character mode. The VDI function call used for this purpose is Enter Alpha Mode. The format for calling this function is

int handle;
v_enter_cur(handle);

On the ST, there is only one display "mode"—bitmapped graphics in varying resolutions. Therefore, graphics text and alphanumeric text may be mixed on the same display, and you aren't required to set alpha mode with the v_enter_cur( ) call before using the alphanumeric output function. You may wish to do so, however, because this call clears the screen and turns on the text cursor, two functions which would otherwise have to be performed separately. The VDI also provides an Exit Alpha Mode command, whose syntax is

int handle ;
v_exit_cur(handle);

On the ST, this call clears the screen and turns off the text cursor. If you don't use this command before your program exits, at least turn off the cursor, or else you'll see it flashing on the Desktop.

The sole means of writing text to the screen in alphanumeric mode is a function called Output Cursor Addressable Text. The v_curtext( ) function outputs text relative to the current cursor position, and wraps text from the end of one line to the beginning of the next. The format for this call is

int handle;
char *string;
v_curtext(handle, string);

where *string* is a pointer to a null-terminated string of text characters.

Cursor Movement Functions

Since alpha text is output relative to the current cursor position, the VDI provides a number of functions that may be used to change this position. The most powerful of these is Direct Cursor Address, which allows you to position the cursor at an absolute row and column position on the screen. The way you call this function is

```
int handle, row, column;
vs_curaddress(handle, row, column);
```

where *row* is a row number from 1 to 25, and *column* is a column number from 1 to 80 (40 for low resolution). Any number outside the range of the cursor will set the cursor at the available position closest to that number. Setting the cursor to the top left position on the screen is a special case that gets its own function. It's called Home Cursor, and looks like this:

```
int handle;
v_curhome(handle);
```

In addition to absolute cursor positioning, you may move the cursor relative to its current position in any direction with the calls Cursor Up, Cursor Down, Cursor Right, and Cursor Left. These calls look like the following:

```
int handle;
v_curup(handle);
v_curdown(handle);
v_curright(handle);
v_curleft(handle);
```

Finally, the VDI provides a call which your program can use to discover the current cursor position. It's called Inquire Current Alpha Cursor Address, and is called like this

```
int handle, row, column;
vq_curaddress(handle, &row, &column);
```

where *row* and *column* are the variables in which the cursor position is returned.

Other Alphanumeric Text Functions

In addition to cursor-positioning commands, the VDI provides a few other miscellaneous alphanumeric text commands. These include two commands to erase text from the current cursor position to the end of the line, or to the end of the screen. These calls are Erase to End of Line, and Erase to End of Screen, and their syntax is

```
int handle;
v_eeol(handle);
v_eeos(handle);
```

The VDI also provides for inverse video, in which the foreground and background colors of the text are reversed. To start printing in inverse video, the call is Reverse Video On, and to resume normal printing, the call is Reverse Video Off. The functions look like this:

```
int handle;
v_rvon(handle);
v_rvoff(handle);
```

The last of the VDI alpha text functions is called Inquire Addressable Character Cells. When GEM is used on systems where the screen format is unknown, this function allows you to find out how may rows and columns are available. Its C syntax is

```
int handle, rows, columns;
vq_chcells(handle, &rows, &columns);
```

Terminal Emulation Functions

Unlike graphics text, which will output any character for which there is image data, alphanumeric text emulates a display terminal, and treats the ASCII characters from 0 to 31 as nonprinting control characters. This means that it will interpret the ASCII character 13 as a carriage return, an instruction to move the cursor to the beginning of the line, rather than as a character that should be printed. On the ST system, the BIOS console device emulates a DEC VT-52 terminal. Since this is the device that's used for alphanumeric mode text, you'll find that the strings output by v_curtext( ) respond to VT-52 escape codes, as well as the VDI control functions. These escape

sequences are codes that begin with the ASCII character 27 (ESC), followed by one or more text characters. The VT-52 codes to which v_curtext( ) responds are

ESC A	Cursor Up
ESC B	Cursor Down
ESC C	Cursor Right
ESC D	Cursor Left
ESC E	Clear Screen and Home Cursor
ESC H	Home Cursor
ESC I	Cursor Up (scrolls screen down if at top line)
ESC J	Clear to End of Screen
ESC K	Clear to End of Line
ESC L	Insert Line
ESC M	Delete Line
ESC Y (row + 32) (column + 32)	Position Cursor at Row, Column (starts with 0)
ESC b (register)	Select Foreground (Character) Color
ESC c (register)	Select Background Color
ESC d	Clear to Beginning of Screen
ESC e	Cursor On
ESC f	Cursor Off
ESC j	Save Cursor Position
ESC k	Move Cursor to Saved Position
ESC l	Clear line
ESC o	Clear from Beginning of Line
ESC p	Reverse Video On
ESC q	Reverse Video Off
ESC v	Line Wrap On
ESC w	Line Wrap Off

In addition to the escape codes, the ST terminal emulation also responds to the following ASCII control codes:

08	Backspace
09	Tab
10–12	Linefeed
13	Carriage Return

Program 7-5 shows the various features of alphanumeric text mode.

Program 7-5. alphmode.c

```
/**************************************************/
/*                                              */
/*    ALPHMODE.C -- Demonstrates the alpha-      */
/*    numeric "Escape" text mode.               */
/*                                              */
/*                                              */
/**************************************************/

#include "shell.c"

demo()
{
    int rows, columns, nul, button=0;

    v_enter_cur(handle); /* clear screen and turn on cursor */
    v_curtext (handle, "This is a test");

 /* move cursor to 10,10, turn reverse video on, and print */
    vs_curaddress(handle,10,10);
    v_rvon(handle);
    v_curtext (handle, "\This is inverse video\r\n");
    v_rvoff(handle);

/* Show how text wraps around to the next line */
    v_curtext (handle, "This is the next line.  It is a very long line.
    Far too long to appear on one  line.  Don't you think?");

/* Mix in some graphic text to show that you can */
    vst_rotation(handle,1800);
    v_gtext(handle,500,50,"This is gtext\n\033p");

/* Show that v_gtext() doesn't move the cursor */

    v_curtext (handle, "  The cursor doesn't move.");

/* Show VT-52 commands */

    v_curtext(handle, "\033B\033BWe also obey VT-52 commands\r\n");
    v_curtext(handle, "Including\033b2foreground and
    \033c1background color\033b3\033c0");

 /* Show number of cells using C function printf() */

    vq_chcells(handle, &rows, &columns);
    printf("\r\n\nRows=%d, Columns=%d\n",rows,columns);

    while(button==0)vq_mouse(handle,&button,&nul,&nul);
    v_exit_cur(handle);

}

/* End of Alphmode.c */
```

BASIC Text Functions

ST BASIC supports the traditional BASIC text output function, PRINT. It also includes the command GOTOXY, which is used to position text in the output window. The only one of the VDI text commands that is supported directly by BASIC is vst_color( ). The text color is set by the first parameter of the COLOR command.

It is possible to control the text settings by making direct calls to the appropriate VDI functions. Since PRINT calls v_gtext( ) to do the actual printing, these settings will affect PRINTed text. You should note, however, that these settings may not work properly with the PRINT function. Take the case shown in Program 7-6.

Note that after the baseline has been rotated, the text printed with v_gtext( ) is displayed upside down with characters going from right to left. But the PRINTed text is upside down, with the characters going from left to right as usual. That's because PRINT outputs each character in the text string separately with v_gtext( ), as a one-character string, positioning each one to the right of the previous one. Another result of this can be seen in the part of the program that PRINTs a text string after the character height has been increased. The large characters are written so closely together that they partially cover each other. That's because BASIC writes out one character at a time and spaces them as it would small characters, since PRINT doesn't know about the size change. That's why you'll most likely have to call Graphics Text directly with VDISYS( ). As Program 7-6 demonstrates, you must POKE each character of the string to the intin array, which makes this method of printing much more cumbersome than PRINT. Still, in order to achieve some text effects, it may be necessary.

Please note that since PRINT uses v_gtext( ), it prints all ASCII characters from 0 to 255. This means that it does not respond to any of the terminal emulation escape codes or cursor-positioning codes, as do most BASIC PRINT statements.

Program 7-6. text.bas

```
100    fullw 2: clearw 2
110    res = peek(systab)
120    if (res<4) then xmax = 639 else xmax = 319
130    if (res>1) then ymax = 199 else ymax = 399
140    REM Set Text Baseline Rotation
150    poke contrl,13 :REM opcode for set baseline vector
```

```
160    poke contrl+2, 0 :REM no points in ptsin
170    poke contrl+6, 1 :REM angle in intin array
180    poke intin,1800
190    vdisys(1)
200    REM
210    color 2  :REM red text
220    gotoxy 16,12
230    print "This is upside down."
240    color 3  :REM green text
250    REM graphics text
260    poke contrl,8 :REM opcode for graphics text
270    poke contrl+2,1 :REM alignment point for text in ptsin
280    a$="This is also upside down."
290    poke contrl+6,len(a$) :REM length of string
300    poke ptsin, xmax*3/4
310    poke ptsin+2,ymax/2
320    for c=1 to len(a$)
330    poke intin-2+(2*c),asc(mid$(a$,c,1))
340    next c
350    vdisys(1)
360    REM Reset Text Baseline Rotation
370    poke contrl,13 :REM opcode for set baseline vector
380    poke contrl+2, 0 :REM no points in ptsin
390    poke contrl+6, 1 :REM angle in intin array
400    poke intin,0
410    vdisys(1)
420    color 1 :REM back to black
430    REM set Text Height, Points mode
440    poke contrl,107 :REM opcode for set height
450    poke contrl+2,0  :REM no ptsin
460    poke contrl+6,1  :REM height in intin
470    poke intin,20
480    vdisys(1)
490    print "This is big"
500    REM set Text Height, Points mode
510    poke contrl,107 :REM opcode for set height
520    poke contrl+2,0  :REM no ptsin
530    poke contrl+6,1  :REM height in intin
540    if (res=1) then poke intin,10 else poke intin,9
550    vdisys(1)
560    print "back to normal"
```

Using Graphic Text from Assembly Language

The conversion of the C function v_gtext( ) to assembly language is a little trickier. That's because the C language bindings take care of the drudgery of moving each character of the string into the intin array. When programming in assembly language , you must take care of this detail yourself. In addition, you must convert each 8-bit character to 16 bits, with the character information in the low-order bits. Program 7-7 demonstrates printing strings with the Graphics Text function.

Program 7-7. text.s

```
************************************************
*                                              *
*                                              *
*       TEXT.S -- assembly language            *
*       graphics text demo                     *
*                                              *
*                                              *
*                                              *
************************************************


.xdef demo
.xref vwkhnd
.xref contrl0
.xref contrl1
.xref contrl2
.xref contrl3
.xref contrl4
.xref contrl5
.xref contrl6
.xref contrl7
.xref contrl8
.xref contrl9
.xref contrl10
.xref contrl11
.xref intin
.xref intout
.xref ptsin
.xref ptsout

.text

demo:

    move     intout+2,ymax
    move     dy,d1
    cmp      #399,ymax      * if high-res
    bne      skip
    add      d1,d1
    move     d1,dy          * double dy

skip:
    move     #24,d5         * starting vertical position
    move     #24,d6         * starting horizontal position
    movea.l  #msg,a4        * save text pointer address in a4
    move     #3,d4          * loop counter
next:

*** Set text color
    move     #22,contrl0    * opcode for text color
    move     #0,contrl1     * no points in ptsin
    move     #1,contrl3     * 1 integer parameter in intin

    move     #4,d1
    sub      d4,d1
    move     d1,intin       * set color from loop counter
    jsr      vdi


*** Print graphics text
    move     #8,contrl0     * opcode for gtext
    move     #1,contrl1     * 1 point in ptsin
    add      dy,d5          * advance y
```

170

```
    move      d5,ptsin+2    * set y text position
    move      d6,ptsin      * set x text position

    move      #81,d0        * maximum no. of characters
    move      d0,d2         * save a copy
    movea.l   #intin,a1     * address of destination in a1
    movea.l   (a4)+,a0      * address of source in a0
text:
    clr.w     d1
    move.b    (a0)+,d1      * move a letter from source...
    move.w    d1,(a1)+      * to word-aligned destination...
    dbeq      d0,text       * until all done.

    sub       d0,d2         * how many characters..
    move      d2,contrl3    * # of characters in string
    jsr       vdi           * print text

    dbra      d4,next       * next text string
    rts


**** data section


    .data
    .even

dy:           .dc.w    16

* Text lines
t1:           .dc.b    'This is the First Line of Text.',0
t2:           .dc.b    'This is the Second Line of Text.',0
t3:           .dc.b    'This is the Third Line of Text.',0
t4            .dc.b    'This is the Last Line of Text.  Over and out.',0

msg:          .dc.l  t1,t2,t3,t4

    .bss
ymax          .ds.w    1

.end
```

Chapter 8

Input Functions

So far, we've concentrated on the output functions of the VDI, those related to drawing on the screen. But the VDI screen device encompasses the ST keyboard and mouse as well, just as the GEMDOS console device includes both the screen and keyboard. The VDI provides functions that directly report the status of physical devices, like the mouse pointer and mouse buttons, and some special keys on the keyboard. It also implements several logical devices that return information from the user to the program in a manner that's a little more independent of the actual hardware on which the GEM operating system is running.

The VDI input functions provide only the bare-bones type of input that you normally associate with computers that lack the elaborate user interface that the ST provides. That's because the AES portion of GEM provides much more sophisticated facilities for interacting with the user than we associate with the ST.

An important point to remember is that the VDI input functions may not be compatible with those of the AES. Most of the time, you'll find that the VDI input functions don't work properly when used in conjunction with the input functions of the AES. Therefore, if you want to use the AES input facilities, which provide all of the power of the VDI functions and much more, you'll have to abandon the VDI functions presented below. You may find, however, that (at least in earlier versions of the ST Operating System), the AES functions are somewhat slow to respond, and may not be as reliable as those of the VDI. For some demanding applications, you may find it desirable to substitute the more basic services of the VDI for the AES input functions. And, for TOS programs that don't need the windowing, icon, and menu services provided by the AES, you may find the VDI input functions adequate for your input needs, and simpler than writing a GEM type program.

So far, we've concentrated on the output functions of the VDI, those related to drawing on the screen. But the VDI screen device encompasses the ST keyboard and mouse as well, just as the GEMDOS console device includes both the screen and keyboard. The VDI provides functions that directly report the status of physical devices, like the mouse pointer and mouse buttons, and some special keys on the keyboard. It also implements several logical devices that return information from the user to the program in a manner that's a little more independent of the actual hardware on which the GEM operating system is running.

The VDI input functions provide only the bare-bones type of input that you normally associate with computers that lack the elaborate user interface that the ST provides. That's because the AES portion of GEM provides much more sophisticated facilities for interacting with the user than we associate with the ST.

An important point to remember is that the VDI input functions may not be compatible with those of the AES. Most of the time, you'll find that the VDI input functions don't work properly when used in conjunction with the input functions of the AES. Therefore, if you want to use the AES input facilities, which provide all of the power of the VDI functions and much more, you'll have to abandon the VDI functions presented below. You may find, however, that (at least in earlier versions of the ST Operating System), the AES functions are somewhat slow to respond, and may not be as reliable as those of the VDI. For some demanding applications, you may find it desirable to substitute the more basic services of the VDI for the AES input functions. And, for TOS programs that don't need the windowing, icon, and menu services provided by the AES, you may find the VDI input functions adequate for your input needs, and simpler than writing a GEM type program.

Physical Devices

The physical devices to which the VDI gives the most direct support is the mouse and its onscreen alter ego, the mouse pointer. One important mouse function that we have already encountered in the shell program is called Sample Mouse Button State. This function not only lets you know whether the left and/or right mouse button is currently being pressed, but also the exact location of the mouse pointer on screen. The C language syntax for this function is

int handle, button, x, y;
vq_mouse(handle, &button, &x, &y);

where *button* is the variable in which the function returns the current button status code, and x and y are the variables in which the function returns the onscreen coordinates of the mouse pointer (which usually looks like an arrow, or a busy bee). The button status code uses the least significant bit to record the status of the left mouse button, and next most significant bit for the right mouse button. These bits contain a 1 if the button is pressed, and a 0 if the button is up. Therefore, the possible button codes are

Code	Meaning
0	Neither button pressed
1	Left button only pressed
2	Right button only pressed
3	Both buttons pressed

If you only wait until the button code is no longer 0, you'll never see a code of 3, since the user will always push down one button a fraction of a second before the other, even if he's certain that he's pushing them both down at the exact same moment. Therefore, you'll have to test the button status several times in a row after the initial push if you want to detect the condition where both butons are pressed at once.

The other point to note about this function is that the x and y coordinates that are returned for the mouse pointer refer to the *hot spot* or *action point* of the pointer. That's the part of the pointer which is considered to be its location on the screen, even when the pointer is considerably larger than a single point. On the arrow shaped pointer, the hot spot is located at the very tip of the arrow, while the bee's hot spot is at its very center.

The Pointer

While the mouse pointer is normally shaped like an arrow (or a bee when they system is busy accessing a disk or the like), the VDI allows you to change the pointer to any 16 × 16 image. The function provided for this purpose is called Set Mouse Form, and it's called like this:

```
int handle, pointerdata[37];
vsc_form(handle, pointerdata);
```

where *pointerdata* is a pointer to an array of 37 integers that provides information about the mouse pointer. This information includes the foreground and background colors for the pointer, the coordinates of the hot spot, the shape of the mouse pointer, and a mask which allows you to specify whether the zero bits in the 16 × 16 block are *transparent* (don't replace existing background with a new color), or *opaque* (replace existing background with pointer background color). The layout of this array is

Element	Description
0	x coordinate of hot spot
1	y coordinate of hot spot
2	Reserved for future use (must be set to 1)
3	Background pen (usually 0)
4	Foreground pen (usually 1)
5–20	16 words of color mask data
21–36	16 words of image data

The x and y coordinates of the hot spot are measured from the top, left corner of the 16 × 16 pixel block. The image data block is arranged exactly the same way as the 16 × 16 pattern fill block. Each line of the image is represented by one 16-bit word, with the most significant bit of the word representing the leftmost dot, and the least significant bit, the rightmost dot. Each bit position that's filled with a 1 is colored with the foreground pen, and each bit position that holds a 0 is colored either in the background color, or whatever color is displayed by the existing background, depending on the color mask.

The color mask is used to define the shape of the pointer, without regard to color information. Those bit positions containing a 1 are considered to be "inside" the pointer, and whether or not the image data contains a 1 bit, this part of the pointer will be colored in, either by the foreground pen or the

background pen. Those bit positions containing a 0 are considered to be "outside" the pointer image, or transparent, and the corresponding image data bit positions that contain 0 will be represented on screen by whatever background data happens to be there.

Having a two-color pointer is very important since you want to make sure the pointer is always visible. Even though the normal system pointers like the arrow appear to be black only, there is actually a thin white line around the outside. This makes it possible for you to see the arrow, even when it's in front of a black background. The sample program mousebox.c (Program 8-1) creates a custom two-color pointer that shows up as red and green on a color monitor.

The vsc_form( ) function is very similar to the AES function graf_mouse( ). That function allows you to choose from several default pointer shapes, such as the arrow, the bee, the pointing hand, and open hand, the I-beam text cursor, and crosshairs.

In addition to the ability to change the appearance of the mouse pointer, the VDI provides functions that allow you to determine whether it will be visible on screen or not. The function used to turn the pointer off is called Hide Cursor, and its C language syntax is

```
int handle;
v_hide_c(handle);
```

The reverse function, used to turn the pointer back on is called Show Cursor, and it's called like this:

```
int handle, reset;
v_show_c(handle, reset);
```

In order to understand the reset flag of the Show Cursor function, you must first understand the interaction between this function and Hide Cursor. Every time you use the Hide Cursor function, the VDI makes a note of it, and hides the cursor down one level farther. So if you call Hide Cursor five times in a row, you must call Show Cursor five times before it becomes visible again—the first four times, Show Cursor just decreases the level at which the pointer is hidden. It is possible to override this system with the reset flag, however. If you call Show Cursor with the reset flag set to 0, the number of previous Hide Cursor calls is ignored, and the mouse pointer

is brought to the top, no matter how far down it was hidden. If the reset flag is set to 1, however, the function behaves normally, and depends on the number of Hide Cursor calls performed previously.

In our previous sample programs, we've seen just how important it is to turn the mouse pointer off before you do any drawing. In the shell program that forms the heart of most of our demonstration programs, we didn't turn off the mouse pointer before clearing the screen, and as a result, the old background is saved behind the pointer. That means that when you move the pointer, the old background is restored, erasing our newly cleared screen in the area of the cursor block along with anything that we drew on it as well. The solution to this problem is to hide the mouse pointer before undertaking any graphics operation, including one so simple as clearing the screen, and restoring it only when you're certain that graphics output has stopped. An example of this practice may be seen in the drawline( ) function of mousebox.c, Program 8-1.

Special Keys

The final physical device function that the VDI provides is used to check the status of some of the special keys on the ST keyboard. The Sample Keyboard State Information function returns information which lets you know whether the Control, Alt, and/or Shift keys are currently pressed. The format for this call is

int handle, key
vq_key_s(handle, &key);

where *key* is a flag indicating the status of the various keys. Bit 0 gives the status of the right Shift key; bit 1 gives that of the left Shift key; bit 2 gives that of the Control key; and bit 3 gives that of the Alt key. If the key is pressed, there will be a 1 in the corresponding bit position, if not, there will be a 0. The values of the various bit positions are as follows:

Bit	Value	Key
0	1	right Shift
1	2	left Shift
2	4	Control
3	8	Alt

Logical Devices

In addition to the physical device functions, the VDI implements some logical input devices. These logical devices provide very specialized input facilities in a device-independent manner. They are provided mostly for purposes of portability, since there are much better ways to get input in an ST-specific environment. The four logical devices are the Locator, Valuator, Choice, and String devices. The functions performed by these devices will be detailed below.

The logical devices operate in one of two modes. In Request mode, the functions do not return until a specific terminating input event occurs. In sample mode, the functions return the current status of the device as soon as they are called.

Before using any of the logical devices, you should specify whether you want it to operate in sample mode or request mode. Set Input Mode is the function used, its format looks like this:

int handle, ldevice mode;
vsin_mode(handle, ldevice, mode);

where *ldevice* is the logical device code, and *mode* is a flag showing whether that device should be set to request or sample mode. The logical devices are

Number	Device
1	Locator
2	Valuator
3	Choice
4	String

The possible modes are:

Number	Mode
1	Request
2	Sample

You may discover the current mode status of any logical device with the function Inquire Input Mode. The syntax for this function is

int handle, ldevice, mode;
vqin_mode(handle, ldevice, &mode);

where *ldevice* is the logical device number, and *mode* is the variable in which the current operating mode is returned. With the current version of the ST operating system, however, this function actually returns the mode number minus one.

Locator Device

The locator device allows the user to specify a point on the display screen. In the request mode, it turns the mouse pointer on, allows the user to move it with the mouse, or with the ALT-arrow key mouse substitutes, until a mouse button or the appropriate key is pressed. The function returns the *x* and *y* position of the mouse pointer, a code that indicates what the terminating event was, and then turns off the mouse pointer. The C syntax for the function Input Locator, Request Mode is

```
int handle, x, y, x1, y1, term;
vrq_locator(handle, x, y, &x1, &y1, &term);
```

where *x* and *y* specify the starting position for the mouse pointer, *x1* and *y1* are the variables in which its ending position is recorded, and *term* is the variable in which the terminating event is returned. The termination code is 32 for the left mouse button, 33 for the right mouse button, or the ASCII value of the key that was pressed to end the function. Note that with the current version of the operating system, this function tends to return immediately as if the left mouse button was pressed. The solution to this problem is to call it twice in a row, and ignore the results of the first call.

When the locator device is used in sample mode, the mouse pointer is not automatically turned on, so, if it isn't showing, you should turn it on with v_show_c( ). The C syntax for Input Locator, Sample Mode is

```
int handle, x, y, x1, y1, term, status;
status = vsm_locator(handle, x, y, &x1, &y1, &term);
```

The *status* variable is used to return a status code that tells whether a mouse button or key was pressed, and if the position of the mouse pointer changed. The least significant bit of this code tells whether there was a position change, and the next bit tells whether a key was pressed. The possible code values are:

Status	Description
0	No key pressed, no position change
1	No key pressed, position changed
2	Key pressed, no position change
3	Key pressed and position changed

String Device

The string device is used to input a string of text characters from the user. In the request mode, this function collects characters until the Return key is pressed, or until the maximum number of characters have filled the buffer. The syntax for Input String, Request Mode is

```
int handle, max_len, echo, xy[2];
char string[max_len];
vrq_string(handle, max_len, echo, xy, &string);
```

where *max_len* is the maximum buffer length, *echo* is a code that specifies whether or not the characters should be echoed to the screen as they are typed in (1=yes, 0=no), *xy* is an array that holds the *x* and *y* coordinates for the echoed characters, and *string* is a pointer to the string of characters that the user enters. In the current version of the ST operating system, echoing of characters (which is not a required feature of the function), isn't supported.

It's also possible to use this device in sample mode. In this mode, the function checks the keyboard once. If there are no keystrokes waiting, the call simply returns. If there are keystrokes, the function keeps collecting them until there aren't any more, the buffer is full, or a carriage return is entered. The syntax for this call is

```
int handle, max_len, echo, status, xy[2];
char string[max_len];
status = vsm_string(handle, max_len, echo, xy, &string);
```

where *status* is the length of the string gathered.

The string function returns the ASCII code for each character that's entered on the keyboard. Some key combinations, however, have no ASCII value. The function keys, the ALT key combinations, and the HELP keys are all examples of key presses without ASCII equivalents. And in some cases, two or more combinations have the same ASCII value. For example, the numbers on the keypad are not distinguishable from the

numbers on the top row of the keyboard by ASCII value
alone. When you wish to get more information about the key
presses other than the ASCII character, you may use a nega-
tive number of max_len. When you do so, the buffer size will
be set to the absolute value of the number specified, and the
values returned in the intout array will consist of two-byte
keycodes, based on the VDI standard keyboard definition. In
most cases, the first byte consists of a code that corresponds to
a particular key on the keyboard, and the second byte is the
ASCII code produced. The full set of keycodes may be found
in Appendix B. Since the C language bindings only copy the
second byte of each word to the string array, you must read
the intout[] array directly to find the full keycode value for
each character.

Program 8-1 uses the String device in sample mode to
check for the user pressing the ESC key, which ends the pro-
gram. It also illustrates many of the other functions discussed
above. It displays a custom two-color mouse pointer (red and
green on color systems), it reads the mouse with vq_mouse( ),
and it hides the mouse pointer before drawing.

Program 8-1. mousebox.c

```
/**********************************************/
/*                                          */
/*                                          */
/*    MOUSEBOX.C -- Demonstrates use of the  */
/*    input functions.                       */
/*                                          */
/*                                          */
/*                                          */
/**********************************************/

#include "shell.c"
#define XOR 3
#define REPLACE 1

/* Data for our own custom two-color pointer */

int pointer[37] =
    {
    8,8,1,    /* x and y of hot spot */
    3,2,      /* background and foreground pens */

/* 16 words of color mask data */

    0xFC7E, 0xFC7E, 0xCC66, 0xCC66, 0xCC66, 0xFC7E, 0xFC7E, 0x0000,
    0xFC7E, 0xFC7E, 0xCC66, 0xCC66, 0xCC66, 0xFC7E, 0xFC7E, 0x0000,

/* 16 words of image data */
```

```
    0xFC7E, 0xFC7E, 0xCC66, 0xCC66, 0xCC66, 0xFC7E, 0xFC7E, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,

  };

demo()
{
    int  mousex, mousey, buttons=0, notdone=1;
    int color, maxcolor, fill=1, points[10];

    maxcolor=color=work_out[13]; /* find highest available color */
/* make cursor visible in monochrome also */
    if (work_out[1]>200)pointer[3]=0;
    v_hide_c(handle);  /* hide the mouse */
    puts("\33EDrag the mouse to draw boxes.\n");
    puts("Press 'ESC' to quit.\n");
    vsc_form(handle,pointer);  /* install our new pointer */
    v_show_c(handle);  /* show the mouse again */

    vsin_mode(handle,4,2);  /* set string device to sample mode */
    vsl_type(handle,3);     /* used dotted line */
    vsf_interior(handle,2); /* Dotted patterns */

    while(notdone)
       {
       while( (buttons==0 ) && notdone) /* wait for button push */
          {
          vq_mouse(handle, &buttons, &mousex, &mousey);
          notdone=testkeys();  /* and check for ESC */
          }

       vswr_mode(handle,XOR); /* drawmode to XOR for 'rubber band' */
       points[0]=points[2]=points[4]=points[6]=points[8]=mousex;
       points[1]=points[3]=points[5]=points[7]=points[9]=mousey;
       drawline(points); /* draw initial point */
       while( (buttons!=0) && notdone) /* while button is held */
          {
          vq_mouse(handle, &buttons, &mousex, &mousey);
          notdone=testkeys(); /* is it moved, or ESC pushed? */
          if( (mousex!=points[2]) || (mousey!=points[7]) )
             {
             drawline(points); /* erase old line */
             points[2]=points[4]=mousex;
             points[5]=points[7]=mousey;
             drawline(points);  /* and draw new one */
             }/* end of if position changed*/
          } /* end of while button is pressed*/

          drawline(points); /* erase last line */

          vswr_mode(handle, REPLACE); /* set drawmode back */

          if (color==maxcolor)color=1; /* advance color */
          vsf_color(handle,color++);
          if (fill==25)fill=1;          /* and fill style */
          vsf_style(handle,fill++);

          points[2]=points[4];
          points[3]=points[5];
          v_hide_c(handle);
          v_bar(handle,points); /* draw the filled box */
          v_show_c(handle);

       } /* end of main while */

 puts("\33EThat's it--Press a mouse button to exit.\n");
```

```
} /* end of main() */

int testkeys()  /* check keyboard for ESC key */
{
   char string [2];
   int status;

   status=vsm_string(handle, -1,0,&string,&string);
   if(intout[0]==0x11B) return(0); /* code for ESC */
   else return(1);
}

drawline(points)
int *points;
{
    v_hide_c(handle);
    v_pline(handle,5,points); /* draw initial point */
    v_show_c(handle);
}

/* End of Mousebox.c */
```

Choice and Valuator Devices

The two remaining logical devices, the Choice and Valuator devices, are not required VDI functions and are not implemented for the screen device in the current version of the ST operating system. We'll describe these functions briefly, for the sake of completeness.

The choice device allows the user to choose one of several options, usually by pressing one of the function keys. In request mode, it waits for one of the keys to be pressed. If it's a function key that's pressed, its value (1–10) is returned, and if not, the default value is returned. The syntax of Input Choice, Request Mode is

int handle, default, choice;
vrq_choice (handle, default, &choice);

where *default* is an initial choice value that you supply, and *choice* is the variable in which the user's choice is returned. In sample mode, the choice and a status value are returned. The syntax for Input Choice, Sample Mode is

int handle, status, choice;
status = vsm_choice(handle, &choice);

If the user was pressing a function key during the call, *status* will contain a 1 and *choice* will indicate the function key pressed. If not, both contain a 0.

The valuator device allows the user to choose a number between 1 and 100. Typically, the user strikes the up and

down arrow keys to increase or decrease the default value. In
request mode, the function takes input until the terminating
character is struck. The syntax for this mode is

int handle, default, value, term;
vrq_valuator(handle, default, &value, &term);

where *default* is an initial value supplied by the programmer,
value is the variable in which the final value is returned, and
term is the variable in which the terminating character is
returned.

In sample mode, the function checks if the increment or
decrement conditions exist during the call, and if they are
present, it changes the value accordingly. The syntax for Input
Valuator, Sample Mode is

int handle, default, value, term, status;
vsm_valuator(handle, default, &value, &term, &status);

The *status* variable will contain a 1 if the valuator
changed during the call, and the *value* variable will contain
the new value. Status will contain a 2 if another key was
pressed, and the term variable will contain the value of this
character. If neither event occurs, status will contain a 0.

Vector Exchange Routines

The fundamental GEM input functions are performed on an
interrupt basis. This means the operating system watches for
certain input events, and calls the appropriate service routine
when such an event occurs. For example, when the mouse is
moved, it calls a routine that updates the position of the
mouse pointer on the screen.

Since an application program sometimes wishes to per-
form actions that are synchronized with one of these input
events, the VDI supplies a number of *vector exchange* routines.
These provide a means of "patching in" your own machine
language routines that will be called before the system inter-
rupt service routine when the event occurs. For example, if
you wanted your program to do something every time the
user pushed the mouse button, you could point the mouse
button interrupt vector at your routine, and have your routine
call the normal mouse button routine when it was done.

The VDI provides interrupt-vector exchange routines for
four input events. These are

Mouse movement. The mouse movement routine is called every time that the mouse moves to a new location. If the application program grabs the mouse movement vector, it gains control after the x and y address coordinates have been calculated, but before the VDI is informed for the new position, and before the mouse pointer is actually redrawn on the screen. At that point, the x coordinate of the mouse pointer is in register d0, and the y coordinate is in d1.

Mouse pointer redraw (cursor change). This routine is called each time the cursor needs to be redrawn. If this vector points to an applications service routine, the application gains control before the redraw actually occurs. This means that the application can take over the task of drawing the cursor, or just perform some action every time that a redraw is scheduled. At the point the application gains control, the x coordinate of the mouse pointer is in register d0, and the y coordinate in d1.

Mouse button press. The button press routine is called each time the state of the mouse buttons change (that is, a button is pressed or released). If this vector is diverted to the application, it receives control after the button press has been decoded, but before GEM learns of the press. At that point, register d0 contains a code that indicates the mouse button status. This code is the same as that used by the Sample Mouse Button State function:

Code	Meaning
0	Neither button pressed
1	Left button only pressed
2	Right button only pressed
3	Both buttons pressed

Timer tick. This routine is called every time the system clock advances one step (or *tick*). This allows the application to take some action on a fixed, periodic basis. The frequency of these tick events in milliseconds is returned by the vector exchange routine.

The mechanics of intercepting a vector is fairly simple. First, you must write a machine language routine that will be called every time the event happens. This routine should preserve all registers in the state in which it found them. When it's called, all interrupts are disabled, and the application code should not enable interrupts. Since the service routines are

called by GEM with a JSR instruction, your routine should end
with a RTS instruction (or JMP to the normal system routine,
which itself ends in a RTS).

Note that when your code is executed, the machine is in
supervisor mode, and its state is such that it's unlikely that
you can successfully make any OS calls from within your
code. Although the ST BIOS is supposed to be reentrant up to
three levels, you may find that this really isn't true. The up-
shot is that you're limited in the kind of tasks that you may
perform by patching into these vectors. Nonetheless, you may
find some legitimate uses for the vector exchange routines. For
example, the mouse movement routines can be used to alter
the rate of change, so as to create a slow motion mode for a
drawing program. By monitoring the mouse buttons, you can
get a good fix on button activity if the higher-level routines
get confused, as they sometimes do. And using the vector ex-
change to constantly update mouse x and y position variables
saves you from calling vq_mouse constantly from your code.

Once you've written the additional (or replacement) rou-
tine, you need to point the system to your new routine. You
do this by calling one of the Vector Exchange routines. These
routines all have pretty much the same syntax:

```
int scr_handle;
unsigned int ticklen;
long oldv, newv;

/* Exchange Button Change Vector */
vex_butv(scr_handle, &newv, &oldv);

/* Exchange Mouse Movement Vector */
vex_motv(scr_handle, &newv, &oldv);

/* Exchange Cursor Change Vector */
vex_curv(scr_handle, &newv, &oldv);

vex_timv(scr_handle, &newv, &oldv, &ticklen);
/* Exchange Timer Interrupt Vector */
```

where *&newv* is a pointer to your new machine language rou-
tine, and *oldv* is a variable in which the address of the old sys-
tem event routine is stored. You should save this address,
since in most cases your machine language routine will want
to call the old routine to perform the normal system function,
either with a JSR from your routine, or by ending your routine

with a JMP to the old one. You'll also need this address to re-store the vector when you are through with it. This can be done with a call of the form:

```
int scr_handle;
long oldv, dummy;
vex_butv(scr_handle,oldv, &dummy);
```

where *oldv* is the variable in which you stored the old vector, and *dummy* is a dummy variable that's used as a place-holder.

Practical experience has shown that in all of the above routines, the handle to use the variable *scr_handle* is *not* the VDI virtual workstation device handle that we've been using all along. Instead, it's the physical screen device handle, the value returned by the AES call graf_handle( ).

BASIC Input Functions

ST BASIC supports the common BASIC input functions, like INPUT, and it also implements the INP command, which allows you input a byte stream though a TOS device. Using device 2, the console device which consists of the keyboard and the display screen, you can read individual keys, one at a time. (Device 4, labeled keyboard, is used to send codes to the intelligent keyboard controller.) This function is comparable to the BIOS routines, however, not the VDI, and so the ASCII codes for the keys are returned, not the VDI keycodes.

The first version of ST BASIC does not incorporate any of the VDI functions for reading the keyboard, mouse pointer position, or mouse buttons. The proposed revision to ST BASIC includes functions such as ASK MOUSE, which returns the position of the mouse pointer and the button status, just like vq_mouse( ). But with the original version of ST BASIC (and to the extent the new one doesn't cover the full range of VDI input commands), you will have to resort to making VDI calls with VDISYS(1) for these functions.

Program 8-2 is a BASIC translation of the mousebox.c, Program 8-1, which was discussed earlier in this chapter. It allows the user to draw filled boxes by dragging a mouse button, and checks the keyboard for the ESC key, which ends the program. You will note, however, that in this version you have to wait for the mouse to stop moving before you see the dotted outline of the box.
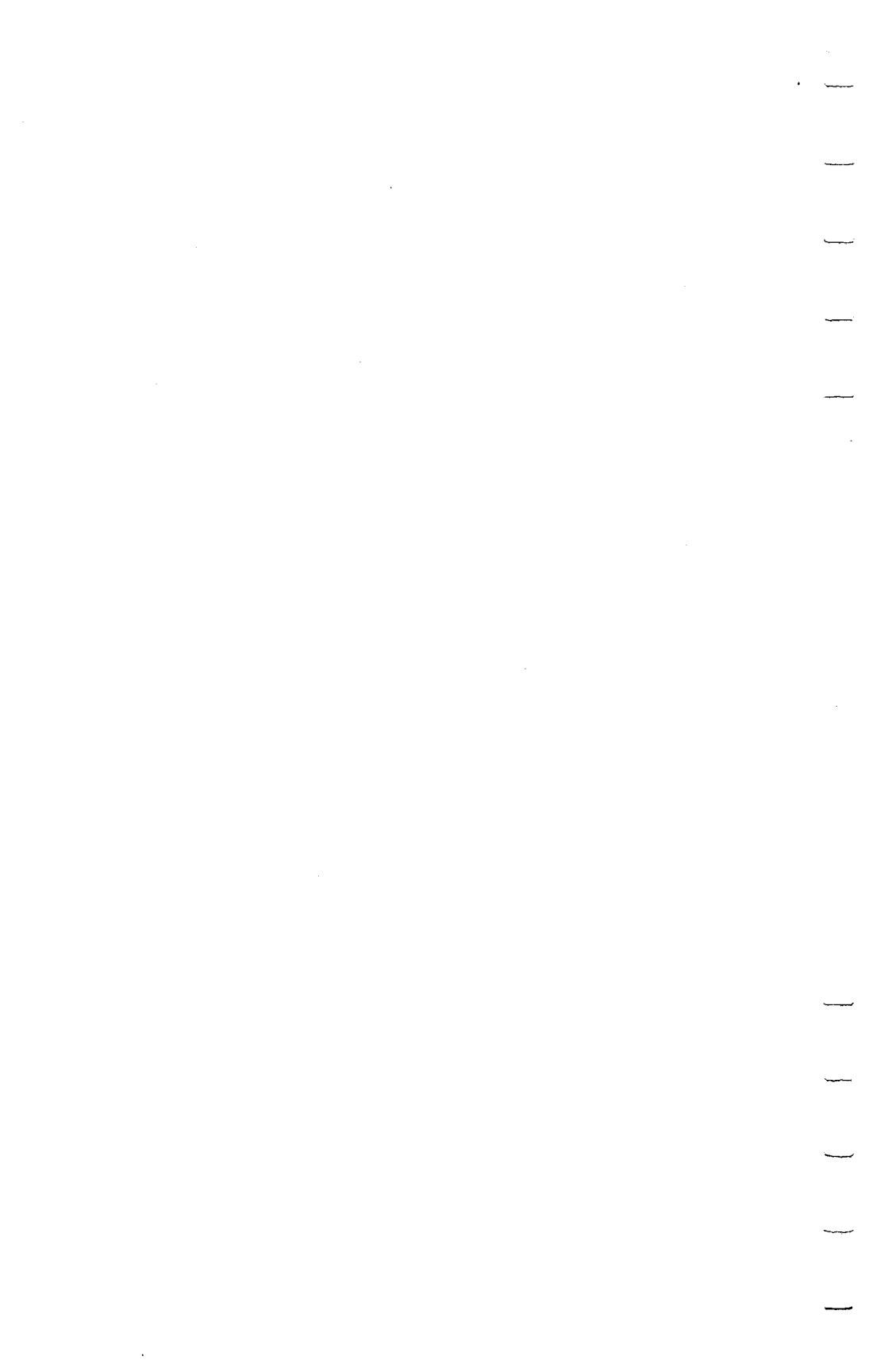
Program 8-2. mousebox.bas

```
100    fullw 2: clearw 2
110    res = peek(systab)
120    maxcolor = res#res
130    filltype=1: fillcol=1
140    REM set string device to sample mode
150    poke contrl,33: REM opcode for set input mode
160    poke contrl+2,0: poke contrl+6,2
170    poke intin,4: poke intin+2,2
180    vdisys(1)
190    REM set line type
200    poke contrl,15: REM opcode for set line type
210    poke contrl+2,0:poke contrl+6,1
220    poke intin,3
230    vdisys(1)
240    done = 0
250    gotoxy 1,1:?"Drag mouse to draw boxes"
260    print " Press ESC to quit"
270    while (done=0) :REM until escape key is pressed
280    while ( (buttons=0) AND (done=0))
290    gosub MOUSEKEYS
300    wend
310    if (done=1) then goto LOOP
320    poke intin, 3: gosub MODE
330    for x=0 to 16 step 4
340    poke ptsin+x, mousex
350    poke ptsin+x+2,mousey
360    next
370    while( (buttons>0) AND (done=0))
380    gosub MOUSEKEYS
390    if (done=1) then goto CHECK
400    gosub DRAWLINE
410    poke ptsin+4,mousex: poke ptsin+8,mousex
420    poke ptsin+10,mousey: poke ptsin+14,mousey
430    gosub DRAWLINE
440    CHECK: wend
450    if (done=1) then goto LOOP
460    gosub HIDE
470    gosub PLINE
480    poke intin,1: gosub MODE
490    if (filltype=25) then filltype=1
500    if (fillcol=maxcolor) then fillcol=1
510    color 1, fillcol, 1, filltype, 2
520    filltype=filltype+1: fillcol=fillcol+1
530    poke ptsin+4,mousex
540    poke ptsin+6,mousey
550    REM do Bar
560    poke contrl,11: poke contrl+10,1
570    poke contrl+2,2:poke contrl+6,0
580    vdisys(1)
590    gosub SHOW
600    LOOP: wend
610    END
620    REM
630    SHOW: poke contrl,122: goto CURSOR
640    HIDE: poke contrl,123
650    CURSOR: poke contrl+2,0:poke contrl+6,0
660    vdisys(1)
670    return
680    PLINE: poke contrl,6: REM pline opcode
690    poke contrl+2,5: poke contrl+6,0
700    vdisys(1)
710    return
720    REM
730    DRAWLINE: gosub HIDE: gosub PLINE: gosub SHOW: return
```

```
740   REM
750   MOUSEKEYS:   poke contrl,124: REM mouse inquiry
760   poke contrl+2,0:poke contrl+6,0
770   vdisys(1)
780   mousex=peek(ptsout): mousey=peek(ptsout+2)
790   buttons=peek(intout)
800   KEYS:   poke contrl,31
810   poke contrl+2,1:poke contrl+6,2
820   poke intin,65535:REM -1
830   vdisys(1)
840   if (peek(intout)=283) then done=1
850   return
860   REM
870   MODE: poke contrl,32
880   poke contrl+2,0
890   poke contrl+6,1
900   vdisys(1)
910   return
```

Appendix A

VDI Function Reference

Open Workstation

v—opnwk( )   **Opcode = 1**

This function opens a workstation that's used to communicate with a physical output device and to keep track of its graphics settings. The VDI communicates with the device by means of a device driver that translates the VDI commands to device-specific commands. When the workstation is opened, the device driver designated by the ASSIGN.SYS file is read in, so this driver must be present and the GDOS must be loaded in order for this command to work. The Open Workstation command allows you to make initial settings for a number of graphics output functions. The function returns the VDI device handle, along with a lot of information about the output capabilities of the device. The device is cleared upon opening.

Devices required for
All

C binding
int handle, work—in[12], work—out[57];
v—opnwk(work—in, &handle, work—out);

Inputs

                    contrl[0] = 1   Opcode
                    contrl[1] = 0   Number of points in ptsin
                    contrl[3] = 11   Number of input integers in intin

     The initial graphics output settings for the workstation are specified by the contents of the intin array (which the bindings take from work—in[ ]).

work—in[0]	intin[0]	Device ID number (from ASSIGN.SYS file)
work—in[1]	intin[1]	Line drawing pattern [vsl—type( )]
work—in[2]	intin[2]	Line pen number [vsl—color( )]
work—in[3]	intin[3]	Marker type [vsm—type( )]
work—in[4]	intin[4]	Marker pen number [vsm—color( )]
work—in[5]	intin[5]	Text font [vst—font( )]
work—in[6]	intin[6]	Text pen number [vst—color( )]
work—in[7]	intin[7]	Fill pattern type [vsf—interior( )]
work—in[8]	intin[8]	Fill pattern index [vsf—style( )]
work—in[9]	intin[9]	Fill pen number [vsf—color( )]

work—in[10]	intin[10]	NDC to RC transformation flag
		  0 = Use Normalized Device Coordinates
		  1 = Reserved for future use
		  2 = Use Raster Coordinate system

Results

	contrl[2] = 6	Number of points in ptsout
	contrl[4] = 45	Number of output integers in intout
handle	contrl[6] = n	The device handle for this device (0 if device was not opened)

The results returned in the intout and ptsout arrays given a wide range of information about the output capabilities of the device.

work—out[0]	intout[0]	Max. horizontal coordinate value (in pixels)
work—out[1]	intout[1]	Max. vertical coordinate value (in pixels)
work—out[2]	intout[2]	Device Coordinate units flag (1 = device doesn't support precise scaling)
work—out[3]	intout[3]	Width of one pixel in microns (1/1000's of a millimeter) For display screens, horizontal component of aspect ratio
work—out[4]	intout[4]	Height of one pixel in microns For display screens, vertical component of aspect ratio
work—out[5]	intout[5]	Number of text font heights (0 = continuous scaling)
work—out[6]	intout[6]	Number of line types.
work—out[7]	intout[7]	Number of line widths (0 = continuous scaling)
work—out[8]	intout[8]	Number of marker patterns
work—out[9]	intout[9]	Number of marker sizes (0 = continuous scaling)
work—out[10]	intout[10]	Number of text fonts supported by the device
work—out[11]	intout[11]	Number of pattern fill styles
work—out[12]	intout[12]	Number of crosshatch fill styles
work—out[13]	intout[13]	Number of drawing pen colors available (the number of colors that can be displayed by the device at the same time)
work—out[14]	intout[14]	Number of Generalized Drawing Primitives (GDPs)—how many of the 10 basic drawing commands are supported
work—out[15] to work—out[24]	intout[15] to intout[24]	This part of the array holds a sequential list of code numbers for the first 10 GDPs supported. Each element holds one of the following code numbers:

1 = Filled rectangle or bar (v—bar)
2 = Circle segment or arc (v—arc)
3 = Filled pie slice (v—pieslice)

	4 = Filled circle (v—circle)	
	5 = Filled ellipse (v—ellipse)	
	6 = Elliptical arc (v—ellarc)	
	7 = Filled elliptical pie slice (v—ellpie)	
	8 = Rounded rectangle (v—rbox)	
	9 = Filled rounded rectangle (v—rfbox)	
	10 = Justified graphics text (v—justified)	
	−1 = End of list	

work_out[25]   intout[25]   This part of the array holds a sequential
    to   to   list of code numbers showing what cate-
work_out[34]   intout[34]   gory of graphics operation is performed
by each of the supported GDPs. This indi-
cates what kind of graphics settings affects
each of the supported commands. Each el-
ement holds one of the following code
numbers:

     0 = Line drawing
     1 = Marker drawing
     2 = Graphics text
     3 = Filled area
     4 = No setting

work_out[35]   intout[35]   Color availability flag
     0 = Device is not capable of color output
     1 = Device is capable of color output

work_out[36]   intout[36]   Text rotation availability flag
     0 = Device is not capable of text rotation
     1 = Device is capable of text rotation

work_out[37]   intout[37]   Area fill availability flag
     0 = Device is not capable of area fill operations
     1 = Device is capable of area fill operations

work_out[38]   intout[38]   Cell array function availability flag
     0 = Device cannot perform the cell array function
     1 = Device can perform the cell array function

work_out[39]   intout[39]   Total number of color choices available in
the palette
     0 = More than 32767 colors available
     1 = Monochrome
     2−32767 = Actual number of colors available

work_out[40]   intout[40]   Input devices available for the locator
function
     1 = Keyboard only
     2 = Keyboard and other device (such as mouse)

work_out[41]   intout[41]   Input devices available for the valuator
function
     1 = Keyboard
     2 = Other device

work_out[42]   intout[42]   Input devices available for the choice
function
     1 = Function keys on keyboard
     2 = Some other key pad

work_out[43]	intout[43]	Input devices available for the string input function
		1 = Keyboard
work_out[44]	intout[44]	Workstation type
		0 = Output only
		1 = Input only
		2 = Input and output
		3 = Reserved for future use
		4 = Metafile output
work_out[45]	ptsout[0]	Minimum character width
work_out[46]	ptsout[1]	Minimum character height
work_out[47]	ptsout[2]	Maximum character width
work_out[48]	ptsout[3]	Maximum character height
work_out[49]	ptsout[4]	Minimum line width
work_out[50]	ptsout[5]	0
work_out[51]	ptsout[6]	Maximum line width
work_out[52]	ptsout[7]	0
work_out[53]	ptsout[8]	Minimum marker width
work_out[54]	ptsout[9]	Minimum marker height
work_out[55]	ptsout[11]	Maximum marker width
work_out[56]	ptsout[12]	Maximum marker height

See also

v_clswk( ), v_opnvwk( ), v_clsvwk( ), vq_extend

Close Workstation

v__clswk( )   Opcode=2

This call is used to terminate output to the graphics device and to release its environment space. If any information remains in a buffer (for a printer or metafile), it's written out at the time the device is closed.

Devices required for
All

C binding
int handle;
v__clswk(handle);

Inputs

	contrl[0] = 2	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The workstation device handle

Results
contrl[2] = 0
Number of points in ptsout
contrl[4] = 0
Number of output integers in intout

See also
v__opnwk( ), v__opnvwk( ), v__clsvwk( )

Clear Workstation

v_clrwk( )   Opcode = 3

This function performs device-specific initialization. For a screen, it clears the screen; for a printer, it erases printer buffer data and sends a form feed, and so forth. No graphics output occurs with any device. The functions provided by this call are also performed by Open Workstation.

Devices required for
All

C binding
int handle;
v_clrwk( );

Inputs

	contrl[0] = 3	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also
v_opnwk( );

Update Workstation

v—updwk( ) Opcode=4

This command is used to flush the output buffers of devices like printers, plotters and the metafile, causing the preceding graphics commands to be executed immediately. It has no effect on the screen device.

Devices required for
All

C binding
int handle;
v—updwk(handle);

Inputs

	contrl[0] = 4	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 1: Inquire Addressable Alpha Cells

vq_chcells( )   Opcode=5

                                                          Function=1

This escape provides information about the number of horizontal and vertical character cell positions at which the alphanumeric cursor may be positioned.

Devices required for
All

C binding
int handle, row, columns;
vq_chcells(handle, &row, &columns);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 1	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 2	Number of output integers in intout
	intout[0]	Number of rows on the screen (−1 means no cursor addressing)
	intout[1]	Number of columns on the screen (−1 means no cursor addressing)

ESC 2: Exit Alpha Mode

vq—exit—cur( )   Opcode=5
                                                    Function=2

This escape would cause the screen device to exit alphanumeric mode and
enter graphics mode if the two modes were separate on the ST. Since they
are not, it clears the screen and turns off the visible cursor.

Devices required for
Screen, Metafile

C binding
int handle;
vq—exit—cur(handle);

Inputs

	contrl[0] = 5		Opcode
	contrl[1] = 0		Number of points in ptsin
	contrl[3] = 0		Number of input integers in intin
	contrl[5] = 2		Function ID
handle	contrl[6] = n		The (virtual) workstation device handle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

ESC 3: Enter Alpha Mode

v_enter_cur( )   **Opcode = 5**
                                                       Function = 3

This escape would cause the screen device to exit graphics mode and enter alphanumeric mode if the two modes were separate on the ST. Since they are not, it clears the screen and turns on the visible cursor, which is positioned in the top left corner.

Devices required for
Screen, Metafile

C binding
int handle;
v_enter_cur(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 3	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

ESC 4: Alpha Cursor Up

v_curup( )   Opcode=5
                                                              Function=4

This escape moves the alpha cursor up one row, unless it's already at the
top row on the screen.

Devices required for

Screen

C binding

int handle;
v_curup(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 4	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 5: Alpha Cursor Down

v_curdown( )   **Opcode=5**

Function=5

This escape moves the alpha cursor one row down, unless it's already at the bottom row on the screen.

Devices required for

Screen

C binding

int handle;
v_curdown(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 5	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 6: Alpha Cursor Right

v—curright( )   **Opcode=5**
                                            Function=6

This escape moves the alpha cursor one column to the right, unless its already at the rightmost column on the screen.

Devices required for

Screen

C binding

int handle;
v—curright(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 6	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 7: Alpha Cursor Left

v_curleft( )   **Opcode = 5**
                                          Function = 7

This escape moves the alpha cursor one column to the left, unless it's already at the leftmost column on the screen.

Devices required for
Screen

C binding
int handle;
v_curleft(handle);

Inputs

	contrl[0]	= 5	Opcode
	contrl[1]	= 0	Number of points in ptsin
	contrl[3]	= 0	Number of input integers in intin
	contrl[5]	= 7	Function ID
handle	contrl[6]	= n	The (virtual) workstation device handle

Results

	contrl[2]	= 0	Number of points in ptsout
	contrl[4]	= 0	Number of output integers in intout

208

ESC 8: Home Alpha Cursor

v—curhome( )   Opcode=5
                                               Function=8

This escape moves the alpha cursor to the top left position on the screen.

Devices required for
Screen

C binding
int handle;
v—curhome(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 8	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 9: Erase To End of Screen

v_eeos( )   **Opcode = 5**
                                               Function = 9

This escape clears the screen from the current cursor position to the end of the screen without moving the cursor.

Devices required for

Screen

C binding

int handle;
v_eeos(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 9	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 10: Erase To End of Line

v—eeol( )   **Opcode=5**

Function=10

This escape clears the screen from the current cursor position to the end of
the line without moving the cursor.

Devices required for

Screen

C binding

int handle;
v—eeol(handle);

Inputs

	contrl[0]	= 5	Opcode
	contrl[1]	= 0	Number of points in ptsin
	contrl[3]	= 0	Number of input integers in intin
	contrl[5]	= 10	Function ID
handle	contrl[6]	= n	The (virtual) workstation device handle

Results

	contrl[2]	= 0	Number of points in ptsout
	contrl[4]	= 0	Number of output integers in intout

ESC 11: Direct Cursor Address

vs_curaddress   **Opcode = 5**
                                                    Function = 11

This escape moves the cursor directly to any position on the screen. If the specified position is outside the range of the screen, the cursor moves to the position on the screen closest to that specified.

Devices required for
Screen

C binding
int handle, row, column;
v_curaddress(handle, row, column);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
	contrl[5] = 11	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
row	intin[0]	Row number (1 to maximum of 80 or 40)
column	intin[1]	Column number (1 to 25)

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 12: Output Alpha Text

v—curtext   **Opcode=5**
Function=12

This escape displays a string of text at the current cursor position, and moves the cursor to the character following the end of the string.

Devices required for
Screen

C binding
```
int handle;
char *string;
v—curtext(handle, string);
```

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = n	Number of characters in string
	contrl[5] = 12	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
string	intin[0]-[n]	Text string, formatted as 8-bit ASCII characters, with each character set within a 16-bit word . The first byte of each word is 0, and the second contains the character code.

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

ESC 13: Reverse Video On

v—rvon( )   Opcode=5
                                            Function=13

This escape causes all subsequent text output to be displayed in reverse video.

Devices required for
Screen

C binding
int handle;
v—rvon(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 13	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 14: Reverse Video On

v—rvoff( )   **Opcode=5**
                                                        Function=14

This escape causes all subsequent text output to be displayed in normal video.

Devices required for

Screen

C binding

int handle;
v—rvof(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 14	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 15: Inquire Cursor Address

vq_curaddress   Opcode=5
                                                     Function=15

This escape returns the current row and column position of the cursor on the screen.

Devices required for
Screen

C binding
int handle, row, column;
v_curaddress(handle, &row, &column);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 15	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 2	Number of output integers in intout
row	intout[0]	Row number (1 to maximum of 80 or 40)
column	intout[1]	Column number (1 to 25)

ESC 16: Inquire Tablet Status

vq_tabstatus( )   Opcode=5
                                                   Function=16

This escape returns the availability of a graphics tablet.

Devices required for
All

C binding
int handle, status;
status = vq_tabstatus(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 16	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
status	intout[0]	Tablet status
		0 = tablet not available
		1 = tablet available

ESC 17: Hard Copy

v_hardcopy( )   **Opcode=5**
                                                     Function=17

This escape copies the physical screen to a printer or other hardcopy device.

Devices required for
All

C binding
int handle;
v_hardcopy(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 17	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

ESC 18: Place Graphic Cursor

v_dspcur( )   Opcode=5
                                                          Function=18

This escape places a graphics cursor at the position indicated.

Devices required for
Screen

C binding
int handle, x, y;
v_dspcur(handle, x, y);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 18	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	X coordinate of pixel location where cursor is to be placed
y	ptsin[1]	Y coordinate of pixel location where cursor is to be placed

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 19: Remove Graphics Cursor

v_rmcur( )   **Opcode=5**

Function=19

This escape removes the graphics cursor placed on the screen by the last
Place Graphics Cursor call.

Devices required for

Screen

C binding

int handle;
v_rmcur(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 19	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 20: Form Advance

v_form_adv( )   Opcode=5
                                                 Function=20

Like the Clear Workstation command, this escape sends a form feed com-
mand to the printer, but unlike that command, it does not discard the infor-
mation stored in the print buffer.

Devices required for
Printer

C binding
int handle;
v_form_adv(handle);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 20	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

221

ESC 21: Output Window

v_output_window( ) **Opcode=5**
Function=21

This escape enables the application to send any designated rectangular screen area to the printer.

Devices required for

Printer

C binding

int handle, points[4];
v_output_window(handle, points);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 21	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	X coordinate of left edge of rectangle
points[1]	ptsin[1]	Y coordinate of top edge of rectangle
points[2]	ptsin[2]	X coordinate of right edge of rectangle
points[3]	ptsin[3]	Y coordinate of bottom edge of rectangle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

ESC 22: Clear Display List

v—clear—disp—list   **Opcode=5**
                                                        Function=22

This escape permits the application to request that the printer display list be cleared.

Devices required for

Printer

C binding

int handle;
v—clear—disp—list(handle)

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	Number of points in PTSIN
	contrl[3] = 0	Length of INTIN array
	contrl[5] = 22	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

ESC 23: Output Bit Image File

v_bit_image( )   Opcode=5
                                                  Function=23

This escape allows the application to print out a bit image file that is stored
in the special VDI screen file format. It provides several page placement and
image scaling options.

Devices required for
Printer

C binding
int handle, aspect, scaling, num_pts, points[ ];
char *filename;
v_bit_image(handle, filename, aspect, scaling, num_pts, points);

Inputs

	contrl[0] = 5	Opcode
	contrl[1] = 0	No points in ptsin means take rectangle information from the bit image file.
	1	One point in ptsin means use this point as the upper left corner, and calculate the lower left from info in the file.
	2	Use these points to define the rectangle.
	contrl[3] = n	Length of file name + 2
	contrl[5] = 23	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
aspect	intin[0]	Aspect ratio flag:
		0 = Ignore aspect ratio
		1 = Preserve pixel aspect ratio
		2 = Preserve page aspect ratio
scaling	intin[1]	Scaling flag:
		0 = Uniform scaling
		1 = Separate scaling
filename	intin[2]−[n]	Filename character string, with one character per 16-bit word.
points[0]	ptsin[0]	X position of left edge (optional)
points[1]	ptsin[1]	Y position of top edge (optional)
points[2]	ptsin[2]	X position of right edge (optional)
points[3]	ptsin[3]	Y position of bottom edge (optional)

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

Polyline

v_pline( ) Opcode=6

This function is used to draw lines between two or more consecutive points. The points may be the same, in which case a single point is drawn. The last point is not automatically connected to the first, so in order to draw a box, five points are required—the last of which should be same as the first.

The output of this function is affected by the general graphics settings and the line settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Line color (vsl_color)
Line type (vsl_type)
Line width (vsl_width)
Line end style (vsl_ends)

Devices required for
All

C Binding
int handle, num_pts, points[ ];
v_pline(handle, num_pts, points);

Inputs

	contrl[0] = 6	Opcode
num_pts	contrl[1] = n	Number of pairs of x,y coordinate line points to draw
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	X coordinate of the first point
points[1]	ptsin[1]	Y coordinate of the first point
.	.	
.	.	
.	.	
points[2n − 2]	ptsin[2n − 2]	X coordinate of the last point
points[2n − 1]	ptsin[2n − 1]	Y coordinate of the last point

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also
vswr_mode( ), vs_clip( ), vsl_color( ), Line type (vsl_type)
vsl_width( ), vsl_ends( )

Polymarker

v_pmarker( )　　　　　　　　　　　　　　　　　**Opcode = 7**

This function draws marker, graphics shapes that range from a single point to box, star and cross shapes.

The output of this function is affected by the general graphics settings and the marker settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Marker color (vsm_color)
Marker type (vsm_type)
Marker height (vsm_height)

Devices required for
All

C Binding
int handle, num_pts, points[ ];
v_pmarker(handle, num_pts, points);

Inputs

	contrl[0] = 7	Opcode
num_pts	contrl[1] = n	Number of pairs of x,y coordinate points for marker points to draw
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	X coordinate of the first point
points[1]	ptsin[1]	Y coordinate of the first point
.	.	
.	.	
.	.	
points[2n − 2]	ptsin[2n − 2]	X coordinate of the last point
points[2n − 1]	ptsin[2n − 1]	Y coordinate of the last point

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also
vswr_mode( ), vs_clip( ), vsm_color( ), vsm_type( ), vsm_height( )

Text

v_gtext( )   Opcode = 8

This function outputs graphics text. No escape characters are recognized by this function, and even non-printing ASCII characters are drawn if there is image data for them in the current character set. Text rendering is affected by the general graphics settings and the text settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Text color (vst_color)
Text font (vst_font)
Text size (vst_height or vst_point)
Baseline rotation (vst_rotation)
Alignment (vst_alignment)
Special effects (vst_effects)

Devices required for
All

C Binding
int handle, x, y;
char *string
v_gtext(handle, x, y, string);

Inputs

	contrl[0] = 8	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = n	Number of characters in string
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	X coordinate of the text alignment point
y	ptsin[1]	Y coordinate of the text alignment point
string[0]	intin[0]	First character of text string. Though each character is an eight-bit value in the C format, the bindings position each of these bytes in a separate word in the intin array. Each member of intin has a high byte of 0 and a low byte that contains the ASCII character.
.	.	
.	.	
.	.	
string[n]	intin[n]	Last character of text string.
	ptsin[0]	X-coordinate (in NDC/RC units) of position where the text string is to be placed.
	ptsin[1]	Y-coordinate (in NDC/RC units) of position where the text string is to be placed.

227

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vst_color( ), vst_font( ), vst_height( ),
vst_point( ), vst_rotation( ), vst_alignment( ), vst_effects( )

Filled Area

v_fillarea( )   Opcode = 9

This function draws a filled polygon whose shape is outlined by a series of points. This polygon may be complex; its lines may cross each other, creating a number of sub-polygons, some of which are filled, others of which are not. In order to insure that the figure is enclosed, the last point is automatically connected to the first. If the points of the figure are the same, a single point is displayed if fill outlining is turned on. The rendering of the filled figure is affected by the general graphics settings and the fill settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Fill color (vsf_color)
Fill interior style (vsf_interior)
Fill style index (vsf_style)
Fill perimeter outline (vsf_perimeter)

Devices required for

All

C Binding

int handle, num_pts, points[ ];
v_fillarea (handle, num_pts, points);

Inputs

	contrl[0] = 9	Opcode
num_pts	contrl[1] = n	Number of pairs of x,y coordinate points in ptsin for the polygon
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	X coordinate of the first point
points[1]	ptsin[1]	Y coordinate of the first point
.	.	
.	.	
.	.	
points[2n−2]	ptsin[2n−2]	X coordinate of the last point
points[2n−1]	ptsin[2n−1]	Y coordinate of the last point

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vsf_color( ), vsf_interior( ),
vsf_style( ), vsf_perimeter( ), vq_extend( ) ( )

GDP 1: Bar

v_bar( )

Opcode = 11
Function = 1

This function draws a filled rectangle. The rendering of the filled figure is affected by the general graphics settings and the fill settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Fill color (vsf_color)
Fill interior style (vsf_interior)
Fill style index (vsf_style)
Fill perimeter outline (vsf_perimeter)

Devices required for
All

C Binding
int handle, points[4];
v_bar(handle, points);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 2	Number of pairs of x,y coordinate points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 1	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	X Coordinate of the left edge
points[1]	ptsin[1]	Y Coordinate of the top edge
points[2]	ptsin[2]	X Coordinate of the right edge
points[3]	ptsin[3]	Y Coordinate of the bottom edge

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also
vswr_mode( ), vs_clip( ), vsf_color( ), vsf_interior( ),
vsf_style( ), vsf_perimeter( ) ( )

230

GDP 2: Arc

v—arc( )

Opcode = 11
Function = 2

Arc draws an arc between any two points on a circle. Points on the circle are measured in tenths of a degree, starting at the rightmost point on the circle and moving counterclockwise. It's possible to draw an entire circle by specifying the same starting and ending points. This function takes the horizontal radius parameter that's passed to it and scales the vertical radius according to the aspect ratio of the monitor, so the circle looks round.

The output of this function is affected by the general graphics settings and the line settings:

Writing mode (vswr—mode)
Clipping rectangle (vs—clip)
Line color (vsl—color)
Line type (vsl—type)
Line width (vsl—width)
Line end style (vsl—ends)

Devices required for
All

C Binding
int handle, x, y, radius, beginangle, endangle;
v—arc(handle, x, y, radius, beginangle, endangle);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 4	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
	contrl[5] = 2	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
beginangle	intin[0]	Starting angle of the arc (0–3600)
endangle	intin[1]	Ending angle of the arc (0–3600)
x	ptsin[0]	X coordinate of center point of arc
y	ptsin[1]	Y coordinate of center point of arc
	ptsin[2]	0
	ptsin[3]	0
	ptsin[4]	0
	ptsin[5]	0
radius	ptsin[6]	Horizontal radius of the arc (the vertical radius is scaled)
	ptsin[7]	0

Results

contrl[2] = 0   Number of points in ptsout
contrl[4] = 0   Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vsl_color( ), Line type (vsl_type)
vsl_width( ), vsl_ends( )

GDP 3: Pie

v_pie( )   **Opcode = 11**
                                                Function = 3

Pie is the filled equivalent of Arc. It draws an arc between any two points on a circle, connects the endpoint of that arc to the center of the circle, and fills the resulting pie wedge with the current fill color and pattern. Points on the circle are measured in tenths of a degree, starting at the rightmost point on the circle and moving counterclockwise. This function takes the horizontal radius parameter that's passed to it and scales the vertical radius according to the aspect ratio of the monitor, so the circle looks round.

The way in which the filled figure is drawn depends on the general graphics settings and the fill settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Fill color (vsf_color)
Fill interior style (vsf_interior)
Fill style index (vsf_style)
Fill perimeter outline (vsf_perimeter)

Devices required for

All

C Binding

int handle, x, y, radius, beginangle, endangle;
v_pie(handle, x, y, radius, beginangle, endangle);

Inputs

	contrl[0]	= 11	Opcode
	contrl[1]	= 4	Number of points in ptsin
	contrl[3]	= 2	Number of input integers in intin
	contrl[5]	= 3	Function ID
handle	contrl[6]	= n	The (virtual) workstation device handle
beginangle	intin[0]		Starting angle of the arc (0–3600)
endangle	intin[1]		Ending angle of the arc (0–3600)
x	ptsin[0]		X coordinate of center point of arc
y	ptsin[1]		Y coordinate of center point of arc
	ptsin[2]		0
	ptsin[3]		0
	ptsin[4]		0
	ptsin[5]		0
radius	ptsin[6]		Horizontal radius of the arc
			(the vertical radius is scaled)
	ptsin[7]		0

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vsf_color( ), vsf_interior( ),
vsf_style( ), vsf_perimeter( )

GDP 4: Circle

v—circle( )   Opcode = 11
                                                     Function = 4

The Circle function draws a filled circle of a given horizontal radius and center point. This function scales the vertical radius according to the aspect ratio of the monitor, so the circle looks round.

The way in which the filled circle is drawn depends on the general graphics settings and the fill settings:

Writing mode (vswr—mode)
Clipping rectangle (vs—clip)
Fill color (vsf—color)
Fill interior style (vsf—interior)
Fill style index (vsf—style)
Fill perimeter outline (vsf—perimeter)

Devices required for
All

C Binding
int handle, x, y, radius;
v—circle(handle, x, y, radius);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 3	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 4	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	X coordinate of center point of circle
y	ptsin[1]	Y coordinate of center point of circle
	ptsin[2]	0
	ptsin[3]	0
radius	ptsin[4]	Horizontal radius of the circle , (the vertical radius is scaled)
	ptsin[5]	0

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also
vswr—mode( ), vs—clip( ), vsf—color( ), vsf—interior( ),
vsf—style( ), vsf—perimeter( ) ( )

GDP 5: Ellipse

v_ellipse( )

Opcode = 11
Function = 5

The Ellipse function draws a filled ellipse of a given horizontal and vertical radius and center point. The way in which the filled ellipse is drawn is affected by the general graphics settings and the fill settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Fill color (vsf_color)
Fill interior style (vsf_interior)
Fill style index (vsf_style)
Fill perimeter outline (vsf_perimeter)

Devices required for
All

C Binding
int handle, x, y, xradius, yradius;
v_ellipse(handle, x, y, radius, yradius);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 5	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	X coordinate of center point of ellipse
y	ptsin[1]	Y coordinate of center point of ellipse
xradius	ptsin[2]	Horizontal radius of the ellipse
yradius	ptsin[3]	Vertical radius of the ellipse

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also
vswr_mode( ), vs_clip( ), vsf_color( ), vsf_interior( ),
vsf_style( ), vsf_perimeter( )

GDP 6: Elliptical Arc

v_ellarc( )

Opcode = 11
Function = 6

Elliptical Arc draws an arc between any two points on an ellipse. Points on the ellipse are measured in tenths of a degree, starting at the rightmost point on the circle and moving counterclockwise. It's possible to draw an entire ellipse by specifying the same point for both the beginning and end. The output of this function is affected by the general graphics settings, and the line settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Line color (vsl_color)
Line type (vsl_type)
Line width (vsl_width)
Line end style (vsl_ends)

Devices required for

All

C Binding

int handle, x, y, xradius, yradius, beginangle, endangle;
v_arc(handle, x, y, xradius, yradius, beginangle, endangle);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
	contrl[5] = 6	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
beginangle	intin[0]	Starting angle of the arc (0–3600)
endangle	intin[1]	Ending angle of the arc (0–3600)
x	ptsin[0]	X coordinate of center point of arc
y	ptsin[1]	Y coordinate of center point of arc
xradius	ptsin[2]	Horizontal radius of the arc
yradius	ptsin[3]	Vertical radius of the arc

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vsl_color( ), Line type (vsl_type)
vsl_width( ), vsl_ends( )

GDP 7: Elliptical Pie

v_ellpie( )

Opcode = 11
Function = 7

Elliptical Pie is the filled equivalent of Elliptical Arc. It draws an arc between any two points on an ellipse, connects the endpoint of that arc to the center of the ellipse, and fills the resulting pie wedge with the current fill color and pattern. Points on the ellipse are measured in tenths of a degree, starting at the rightmost point on the ellipse and moving counterclockwise. The way in which the filled figure is drawn depends on the general graphics settings and the fill settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Fill color (vsf_color)
Fill interior style (vsf_interior)
Fill style index (vsf_style)
Fill perimeter outline (vsf_perimeter)

Devices required for

All

C Binding

int handle, x, y, radius, yradius, beginangle, endangle;
v_pie(handle, x, y, radius, yradius, beginangle, endangle);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
	contrl[5] = 7	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
beginangle	intin[0]	Starting angle of the arc (0–3600)
endangle	intin[1]	Ending angle of the arc (0–3600)
x	ptsin[0]	X coordinate of center point of arc
y	ptsin[1]	Y coordinate of center point of arc
xradius	ptsin[2]	Horizontal radius of the arc
yradius	ptsin[3]	Vertical radius of the arc

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vsf_color( ), vsf_interior( ),
vsf_style( ), vsf_perimeter( )

GDP 8: Rounded Rectangle

v_rbox( )   **Opcode=11**
                                                Function=8

This function draws a rectangle with rounded corners. The output of this function is affected by the general graphics settings and the line settings:

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Line color (vsl_color)
Line type (vsl_type)
Line width (vsl_width)
Line end style (vsl_ends)

Devices required for

All

C Binding

int handle, points[4];
v_rbox(handle, points);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 8	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	X Coordinate of the left edge
points[1]	ptsin[1]	Y Coordinate of the top edge
points[2]	ptsin[2]	X Coordinate of the right edge
points[3]	ptsin[3]	Y Coordinate of the bottom edge

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vsl_color( ), Line type (vsl_type)
vsl_width( ), vsl_ends( )

GDP 9: Filled Rounded Rectangle

v—rfbox( )   **Opcode=11**
                                           Function=9

This function draws a filled rectangle with rounded corners. The appearance of the rectangle is affected by the general graphics settings and the fill settings:

Writing mode (vswr—mode)
Clipping rectangle (vs—clip)
Fill color (vsf—color)
Fill interior style (vsf—interior)
Fill style index (vsf—style)
Fill perimeter outline (vsf—perimeter)

Devices required for
All

C Binding
int handle, points[4];
v—rfbox(handle, points);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
	contrl[5] = 9	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	X Coordinate of the left edge
points[1]	ptsin[1]	Y Coordinate of the top edge
points[2]	ptsin[2]	X Coordinate of the right edge
points[3]	ptsin[3]	Y Coordinate of the bottom edge

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also
vswr—mode( ), vs—clip( ), vsf—color( ), vsf—interior( ),
vsf—style( ), vsf—perimeter( )

240

GDP 10: Justified Graphics Text

v—justified( )   **Opcode = 11**

Function = 10

This function outputs a line of graphics text that is both right and left justified. Pixel spaces are added or removed between characters and/or words, causing the string to be printed in the specified width. No escape characters are recognized by this function, and even non-printing ASCII characters are drawn if there is image data for them in the current character set. Text rendering is affected by the general graphics settings and the text settings:

Writing mode (vswr—mode)
Clipping rectangle (vs—clip)
Text color (vst—color)
Text font (vst—font)
Text size (vst—height or vst—point)
Baseline rotation (vst—rotation)
Alignment (vst—alignment)
Special effects (vst—effects)

Devices required for
All

C Binding
int handle, x, y, length, word—space, char—space;
char *string
v—gtext(handle, x, y, string, length, word—space, char—space);

Inputs

	contrl[0] = 11	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = n+2	Number of characters in string plus two
	contrl[5] = 10	Function ID
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	X coordinate of the text alignment point
y	ptsin[1]	Y coordinate of the text alignment point
length	ptsin[2]	Requested length of string
	ptsin[3]	0
word—space	intin[0]	Interword spacing flag
		0 = Don't modify spacing
		1 = Modify spacing
char—space	intin[1]	Intercharacter spacing flag
		0 = Don't modify spacing
		1= Modify spacing
string[0]	intin[2]	First character of text string. Though each character is an eight-bit value in the C format, the bindings position each of these bytes in a separate word in the intin array. Each member of intin has a high byte of 0 and a low byte that contains the ASCII character.

241

.   .
.   .
.   .

string[n]   intin[n+2]   Last character of text string.

Results

      contrl[2] = 0   Number of points in ptsout
      contrl[4] = 0   Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vst_color)), vst_font( ), vst_height( ),
vst_point( ), vst_rotation( ), vst_alignment( ), vst_effects( )

Set Character Height, Absolute Mode

vst_height( )   Opcode=12

This function sets the graphics text character height, as measured from the baseline to the top of the character cell, in terms of an absolute pixel value. It returns information about the height and width of the characters and the character cell. For proportional fonts, the width returned is that of the widest character and character cell in the font. If the font height requested is not available, the VDI selects the next smallest available font size.

Devices required for

All

C Binding

int handle, height, char_width, char_height, cell_width, cell_height;
v_height (handle, height, char_width, char_height,
        cell_width, cell_height);

Inputs

	contrl[0] = 12	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
	ptsin[0]	0
height	ptsin[1]	Requested character height, in pixels

Results

	contrl[2] = 2	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout
char_width	ptsout[0]	Character width in pixels
char_height	ptsout[0]	Character height in pixels
cell_width	ptsout[0]	Cell width in pixels
cell_height	ptsout[0]	Cell height in pixels

See also

vst_points( )

Set Character Baseline Vector

vst_rotation( ) **Opcode = 13**

This function rotates the baseline of graphics text so that subsequent text characters are printed upside down or up and down rather than left to right. The Atari ST computers only support rotation of text in increments of 90-degree angles.

Devices required for
Metafiles

C Binding
int handle, angle, angle_set;
angle_set = vst_rotation(handle, angle);

Inputs

	contrl[0] = 13	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
angle	intin[0] n	The requested angle of rotation
		(0, 900, 1800 and 2700 are valid)

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
angle_set	intout[0] n	Angle of rotation for baseline selected

See also
v_gtext( ), v_justified( )

Set Color Representation

vs__color( )
<div align="right">Opcode = 14</div>

This function is used to change the color in the hardware color register that's associated with VDI drawing pens (color index). It lets you assign a Red, Blue, and Green color value to create a particular shade for any VDI drawing pen (color index). Since the hi-res monitor only supports black and white, this function does nothing when it is called on a monochrome system.

Devices required for
All

C Binding
int handle, pen, rgb[3];
vs__color(handle, pen, rgb);

Inputs

	contrl[0] = 14	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 4	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
pen	intin[0]	The VDI pen (color index) number associated with the hardware color register to be changed. Note that pen numbers do not correspond directly to color register numbers (for example, pen 7 is not the same as color register 7). The relationship between the two is shown in the chart below. For medium-resolution mode, only pens 0–3 are valid, and pen 1 corresponds to register 3, as shown in parentheses.

Pen Number	Register Number	Default Color
0	0	White
1	15 (*3M)	Black
2	1	Red
3	2	Green
4	4	Blue
5	6	Cyan
6	3	Yellow
7	5	Low White
9	8	Gray
10	9	Light Red
11	10	Light Green
12	12	Light Blue
13	14	Light Cyan
14	11	Light Yellow
15	13	Light Magenta

rgb[0]	intin[1]	The Red color-brightness value (0–1000)
rgb[1]	intin[2]	The Blue brightness value (0–1000)
rgb[2]	intin[3]	The Green brightness value (0–1000)

Although the VDI uses color levels from 0–1000, currently, the ST hardware supports eight brightness values for each color, 0–7. So the VDI maps its values to the actual hardware values as follows:

Requested Value	Actual Value Set	Hardware Register Color Level
0–70	0	0
71–213	142	1
214–356	285	2
357–499	428	3
500–642	571	4
643–785	714	5
786–928	857	6
929 and up	1000	7

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also

vsl—color( ), vsf—color( ), vsm—color( ), vst—color( )

Set Polyline Linetype

vsl_type( )   Opcode=15

This function sets a line pattern that is used by subsequent line-drawing routines. It can install one of six preset patterns or an arbitrary 16-bit pattern set by the programmer. If the pattern number requested is unavailable, pattern 1, a solid line, will be set. Note that patterned lines are unavailable when the vsl_width( ) function is used to set a line width greater than 1. These thicker lines always appear as solid.

Devices required for
All

C Binding
int handle, type, type_set;
type_set = vsl_type(handle, type);

Inputs

	contrl[0] = 15	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
type	intin[0]	Line pattern type

> 1 = Solid
> 2 = Long dash
> 3 = Dot
> 4 = Dash, dot
> 5 = Dash
> 6 = Dash, dot, dot
> 7 = User-defined

Pattern 7 installs the pattern that has been set with the Set User-Defined Line Style Pattern function (vsl_udsty). If no pattern has been set, the function uses the default style, a solid line. For an illustration of the various bit patterns, see Figure 5-2.

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
type_set	intout[0]	

See also
vsl_udsty( )

Set Polyline Line Width

vsl_width( )   Opcode = 16

This command sets the width of the lines produced by subsequent line-drawing operations. On the ST, the line width may be any odd number from 1 through 39. Requests for even numbers will cause the next-lowest odd-numbered width to be set. Note that when lines wider than one pixel are being used, the current line pattern will be ignored, and a solid line will be drawn.

Devices required for
Metafiles

C Binding
int handle, width, width_set;
width_set = vsl_width(handle, width);

Inputs

	contrl[0] = 16	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
width	ptsin[0]	Requested line width in pixels
	ptsin[1]	0

Results

	contrl[2] = 1	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout
width_set	ptsout[0]	Line width that was actually set
	ptsout[1]	0

Set Polyline Color Index

vsl_color( ) Opcode=17

This function sets the hardware color register that will be used for future line-drawing operations. Note that the color index (pen) number does not correspond exactly to the color register number. The VDI uses a lookup table to match the color index to a color register for the color screen. On the monochrome screen, color 0 is white, the background color; and 1 is black, the foreground color. If the color that was requested is not available, color index 1 (black) will be selected.

Devices required for

All

C Binding

int handle, color, color_set
color_set = vsl_color(handle, color);

Inputs

	contrl[0] = 17	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
color	intin[0]	The requested color index (pen) number

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
color_set	intout[0]	The color index that was actually set

Set Polymarker Type

vsm_type( ) Opcode=18

This function sets the marker shape that will be used by subsequent poly-marker operations. There are six preset marker shapes. The first of these, the dot, is always one pixel in size, and cannot be scaled the way the others can. If the marker that is requested is unavailable, marker number 3, the asterisk, is selected.

Devices required for
All

C Binding

int handle, shape, shape_set
shape_set = vsm_type(handle, shape);

Inputs

	contrl[0] = 18	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
shape	intin[0]	The marker shape that's requested
		1 = Dot (.)
		2 = Plus (+)
		3 = Asterisk (*)
		4 = Square ([])
		5 = Diagonal Cross (X)
		6 = Diamond (<>)

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
shape_set	intout[0]	The marker shape that was actually set

Set Polymarker Height

vsm_height( )   Opcode=19

This function sets the size of the polymarker shapes 2–6 (shape 1 is always one pixel in size). There are eight marker sizes available on the ST screen, ranging from 15 × 11 to 120 × 88. Each of the marker heights set by this function is an even multiple of 11 (11, 22, 33, and so on). If the requested height is not available, the next smallest available height is set. Though the function returns both the height and width of the marker that was actually set, the C bindings only return the height.

Devices required for
Metafiles

C Binding
int handle, height, height_set;
height_set = vsm_height(handle, height);

Inputs

	contrl[0] = 19	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
	ptsin[0]	0
height	ptsin[1]	The requested polymarker height

Results

	contrl[2] = 1	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout
	ptsout[0]	Width of the selected polymarker
height_set	ptsout[0]	Height of the selected polymarker

Set Polymarker Color Index

vsm_color( ) Opcode=20

This function sets the hardware color register that will be used for future polymarker operations. Note that the color index (pen) number does not correspond exactly to the color register number. The VDI uses a lookup table to match the color index to a color register for the color screen. On the monochrome screen, color 0 is white, the background color; and 1 is black, the foreground color. If the color that was requested is not available, color index 1 (black) will be selected.

Devices required for

Metafiles

C Binding

int handle, color, color_set
color_set = vsm_color(handle, color);

Inputs

	contrl[0] = 20	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
color	intin[0]	The requested color index (pen) number

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
color_set	intout[0]	The color index that was actually set

Set Text Face

vst_font( )   Opcode=21

This function selects the font that will be used for subsequent graphics text output. Font 1 is a built-in system font. All others are disk-based, and must be loaded using the Load Face operation. The font that you request in this function is identified by a font index number which you may determine by using the Inquire Face Name function. Note that in order to use disk-based fonts on the current version of the ST, the GDOS extension must first be loaded.

Devices required for
All

C Binding
int handle, fontID, font_set;
font_set = vst_font(handle, fontID);

Inputs
	contrl[0] = 21	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
fontID	intin[0]	Font ID number of the font requested
	(determined from call vqt_font)	

Results
	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
font_set	intout[0]	Font ID number that was actually set

See also
vqt_name( )

Set Graphics Text Color Index
vst_color( )   Opcode = 22

This function sets the hardware color register that will be used for future graphics text operations. Note that the color index (pen) number does not correspond exactly to the color register number. The VDI uses a lookup table to match the color index to a color register for the color screen. On the monochrome screen, color 0 is white, the background color; and 1 is black, the foreground color. If the color that was requested is not available, color index 1 (black) will be selected.

Devices required for
All

C Binding
int handle, color, color_set
color_set = vst_color(handle, color);

Inputs

	contrl[0] = 20	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
color	intin[0]	The requested color index (pen) number

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
color_set	intout[0]	The color index that was actually set

Set Fill Interior style

vsf_interior( ) Opcode=23

This command determines the type of pattern used for filled-shape output functions. If the pattern style requested is unavailable, fill style 0 (hollow) is set.

Devices required for
All

C Binding
int handle, pattern, pattern_set;
pattern_set = vsf_interior(handle, pattern);

Inputs

	contrl[0] = 23	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
pattern	intin[0]	The style of fill pattern requested

$\quad$ 0 = Hollow
$\qquad$ (Background color, pen 0)
$\quad$ 1 = Solid
$\qquad$ (Foreground color, fill pen)
$\quad$ 2 = Pattern (Dotted)
$\quad$ 3 = Crosshatch
$\quad$ 4 = User-defined style
User-defined fill style must first be set with vsf_udpat( ). If no user-defined pattern has been set the default Atari-logo pattern is used.

Results

	contrl[2] =	Number of points in ptsout
	contrl[4] =	Number of output integers in intout
pattern_set	intout[0]	The style of fill pattern actually set

See also
vsf_style( )

Set Fill Style Index

vsf_style( ) Opcode = 24

This function is used to choose a particular fill pattern from those available for a given fill type. This fill pattern will be used for all subsequent fill operations. Only the Pattern (Dotted) and Crosshatch fill styles offer a number of patterns to choose from, so the general pattern style must be set to either of those in order for this command to have any effect. On the ST screen, the Pattern (Dotted) fill contains 24 different subpatterns, while the Hatch fill includes 12.

Devices required for
All

C Binding
int handle, index, index_set;
index_set = vsf_style(handle, index);

Inputs

	contrl[0] = 24	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
index	intin[0]	The requested fill index
		1–24 for Pattern, 1–12 for Hatch

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
index_set	intout[0]	The fill index actually set

See also
vsf_interior, vsf_color

Set Fill Color Index

vsf_color( )   **Opcode = 25**

This function sets the hardware color register that will be used for future fill operations. Note that the color index (pen) number does not correspond exactly to the color register number. The VDI uses a lookup table to match the color index to a color register for the color screen. On the monochrome screen, color 0 is white, the background color; and 1 is black, the foreground color. If the color that was requested is not available, color index 1 (black) will be selected.

Devices required for
All

C Binding
int handle, color, color_set
color_set = vsf_color(handle, color);

Inputs

	contrl[0] = 25	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
color	intin[0]	The requested color index (pen) number

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
color_set	intout[0]	The color index that was actually set

Inquire Color Representation
vq—color( )   Opcode=26

This function returns either the actual RGB values for a color index (pen), or the value that was requested when the index was set by vs—color( ). If an invalid index is selected, a value of −1 is returned in intout(0). This value is not returned by the C binding.

Devices required for
Screen, Printer, Metafile

C Binding
int handle, color, flag, rgb[3];
vq—color(handle, color, flag, rgb);

Inputs

	contrl[0] = 26	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
index	intin[0]	Color index requested
flag	intin[1]	Inquiry mode flag
		0 = Return color value requested
		1 = Return color value actually set

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 4	Number of output integers in intout
	intout[0]	Color index used
rgb[0]	intout[1]	Red level (0–1000)
rbg[1]	intout[2]	Green level (0–1000)
rgb[2]	intout[3]	Blue level (0–1000)

See also
vs—color( )

Input locator, request mode

vrq_locator( ) Opcode = 28

This function waits for a terminating event, such as a keypress or mouse button press, and returns the position of the mouse pointer. The mouse pointer is shown at the beginning of this function; it remains visible and follows the mouse until the terminating event, at which time it is hidden. The ALT-arrow key combination may be used to move the mouse pointer. This mode of the locator function is chosen by first setting the device to request mode with the vsin_mode( ) function.

Devices required for

Screen

C Binding

int handle, x, y, x1, y1, term;
vrq_locator(handle, x, y, &x1, &y1, &term);

Inputs

	contrl[0] = 28	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	Starting x position of the mouse pointer
y	ptsin[1]	Starting y position of the mouse pointer

Results

	contrl[2] = 1	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
term	intout[0]	Terminating event. If a keypress, the ASCII value of the key pressed. If a button press, 32 for the left button, 33 for the right button.
x1	ptsout[0]	Ending x position of the mouse pointer
y1	ptsout[1]	Ending y position of the mouse pointer

See also

vsin_mode( ), vsm_locator( )

Input Locator, Sample Mode

vsm_locator( )   **Opcode = 28**

This function returns the current position of the mouse pointer. The mouse pointer is not shown by this function, so v_show_c( ) may be used to do so. This mode of the locator function is chosen by first setting the device to sample mode with the vsin_mode( ) function.

Devices required for
Screen

C Binding
int handle, x, y, x1, y1, term, status;
status = vsm_locator(handle, x, y, &x1, &y1, &term);

Inputs

	contrl[0] = 28	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	Starting x position of the mouse pointer
y	ptsin[1]	Starting y position of the mouse pointer

Results

	contrl[2] =	Number of points in ptsout
	1	if mouse pointer coordinates changed
	0	if coordinates didn't change
	contrl[4] =	Number of output integers in intout
	1	if terminating event occurred
	0	if no mouse button- or keypress occurred
status		A value that shows if terminating event occurred and/or if mouse moved. The binding makes this value equal to contrl[2] I (contrl[4] << 1), so that bit 0 is set if the mouse moved, and bit 1 is set if a terminating event occurred.
term	intout[0]	Terminating event. If a keypress, the ASCII value of the key pressed. If a button press, 32 for the left button, 33 for the right button.
x1	ptsout[0]	Ending x position of the mouse pointer
y1	ptsout[1]	Ending y position of the mouse pointer

See also
vsin_mode( ), vrq_locator( )

Input Valuator, Request Mode

vrq_valuator( )   Opcode=29

This function implements a logical device that allows the user to return a numerical value from 1 to 100. In request mode, the up- and down-arrow keys are used to increase or decrease the starting value until a terminating key is pressed. This mode of the valuator function is chosen by first setting the device to request mode with the vsin_mode( ) function. This function is not implemented in the current version of the TOS ROMs.

Devices required for

None

C Binding

int handle, begin_value, end_value, term;
vrq_valuator(handle, begin_value, &end_value, &term);

Inputs

	contrl[0] = 29	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
begin_value	intin[0]	The starting valuator value

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 2	Number of output integers in intout
end_value	intout[0]	The value set by the user at time of termination
term	intout[1]	The ASCII value of the terminating key

See also

vsin_mode( ), vsm_valuator( )

Input Valuator, Sample Mode

vsm_valuator( )   **Opcode = 29**

This function implements a logical device that allows the user to return a numerical value from 1 to 100. In sample mode, the function checks whether the up- or down-arrow keys are currently pressed, thereby increasing or decreasing the starting value. This mode of the valuator function is chosen by first setting the device to sample mode with the vsin_mode( ) function. This function is not implemented in the current version of the TOS ROMs.

Devices required for

None

C Binding

int handle, begin_value, end_value, term, status;
vsm_valuator(handle, begin_value, &end_value, &term, &status);

Inputs

	contrl[0] = 29	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
begin_value	intin[0]	The starting valuator value

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] =	Number of output integers in intout
	0	if no value change and no terminator
	1	if value changed
	2	if terminating keypress occurred
end_value	intout[0]	The value set by the user (if changed)
term	intout[1]	The ASCII value of the terminating key, if any

See also

vsin_mode( ), vrq_valuator( )

Input Choice, Request Mode
vrq_choice( )   Opcode=30

This function implements a logical device that allows the user to choose a number from 1 to 10 via the function keys. In request mode, this function waits until a key is pressed, and, if a function key is pressed, it returns the key number. If the terminating event is not a function keypress, the default choice number is returned. This mode of the choice function is chosen by first setting the device to request mode with the vsin_mode( ) function. This function is not implemented in the current version of the TOS ROMs.

Devices required for
None

C Binding
int handle, begin_choice, end_choice;
vrq_choice(handle, begin_choice, &end_choice);

Inputs

	contrl[0] = 30	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
begin_choice	intin[0]	The default choice number

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
end_value	intout[0]	The choice number

See also
vsin_mode( ), vsm_choice( )

Input Choice, Sample Mode
vsm—choice( )   Opcode=30

This function implements a logical device that allows the user to choose a number from 1 to 10 via the function keys. In sample mode, this function checks the keyboard, and returns a choice number if one of the function keys is pressed. This mode of the choice function is chosen by first setting the device to sample mode with the vsin—mode( ) function. This function is not implemented in the current version of the TOS ROMs.

Devices required for
None

C Binding
int handle, choice;
vsm—choice(handle, &choice);

Inputs

	contrl[0] = 30	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] =	Number of output integers in intout
	0	if no function key is pressed
	1	if function key is pressed
choice	intout[0]	The function key pressed (if any)

See also
vsin—mode( ), vrq—choice( )

Input String, Request Mode

vrq—string( )   **Opcode=31**

This function allows the user to enter a string of text characters. In request mode, each keypress adds an ASCII character (or a 16-bit keycode value) to the end of the string until the Return key is pressed or the maximum string length is reached. This mode of the choice function is chosen by first setting the device to request mode with the vsin—mode( ) function.

Devices required for
Screen

C Binding
int handle, max—length, echo—mode, echo—xy[2];
char string[max—length];
vrq—string(handle, max—length, echo—mode, echo—xy, &string);

Inputs

	contrl[0] = 31	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
max—length	intin[0]	The maximum string length. If this length is specified as a negative number, the absolute value of that number is used as the length, and the 16-bit keycode value for each keypress is placed in the string, instead of its 8-bit ASCII value.

echo—mode	intin[1]	Echo mode flag
		   0 = no echo
		   1 = echo characters to screen as they are entered, starting at position echo—xy (not supported on ST)

echo—xy[0]	ptsin[0]	X coordinate of start of echo text
echo—xy[1]	ptsin[1]	Y coordinate of start of echo text

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = n	Number of string characters in intout
string[0]	intout[0]	First character of text string. Each 8-bit ASCII value is positioned in the low byte of a 16-bit intout word. (The high byte is always zero.) If max—length is negative, indicating that 16-bit keycodes are to be used instead of 8-bit ASCII characters, the C bindings still only copy the low byte of each character to string. In order to read the full 16-bit value of each character, therefore, you must read each one directly from the intout array:

.   .
.   .
.   .

string[n−1]   intout[n−1]   Last character of text string

See also

vsin_mode( ), vsm_string( )

Input String, Sample Mode

vsm_string( )   Opcode = 31

This function allows the user to enter a string of text characters. In sample mode, the function tests to see if any key is pressed. If any has been, each keypress adds an ASCII character (or a 16-bit keycode value) to the end of the string, until the Return key is pressed, the maximum string length is reached, or no more keys are pressed. This mode of the choice function is chosen by first setting the device to sample mode with the vsin_mode( ) function.

Devices required for
Screen

C Binding
int handle, max_length, echo_mode, status, echo_xy[2];
char string[max_length];
status = vsm_string(handle, max_length, echo_mode, echo_xy, &string);

Inputs

	contrl[0] = 31	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
max_length	intin[0]	The maximum string length. If this length is specified as a negative number, the absolute value of that number is used as the length, and the 16-bit keycode value for each keypress is placed in the string, instead of its 8-bit ASCII vlaue.

echo_mode	intin[1]	Echo mode flag
		     0 = no echo
		     1 = echo characters to screen as they are entered, starting at position echo_xy (not supported on ST)

echo_xy[0]	ptsin[0]	X coordinate of start of echo text
echo_xy[1]	ptsin[1]	Y coordinate of start of echo text

Results

	contrl[2] = 0	Number of points in ptsout
status	contrl[4] = n	Number of string characters in intout 0 = keypress data is available >0 = number of characters gathered
string[0]	intout[0]	First character of text string (if any). Each 8-bit ASCII value is positioned in the low byte of a 16-bit intout word. (The high byte is always zero.) If max_length is negative, indicating that 16-bit keycodes are to be used instead of 8-bit ASCII characters, the C bindings still only copy the

low byte of each character to string. In order to read the full 16-bit value of each character, therefore, you must read each one directly from the intout array

.　　　　　　.
.　　　　　　.
.　　　　　　.

string[n − 1]　　intout[n − 1]　　Last character of text string

See also

vsin_mode( ), vrq_string( )

Set Writing Mode

vswr_mode( ) Opcode = 32

This function sets the writing mode, which affects how all subsequent drawing operations will be performed. Not only may a VDI drawing operation replace an existing background picture, but it may also be combined with that background in various ways. Four drawing modes are supported: Replace, Transparent, Exclusive OR (XOR), and Reverse Transparent. The writing mode affects marker, line, fill, and graphics-text operations.

Devices required for

Screen, Printer, Metafile

C Binding

int handle, mode, mode_set;
mode_set = vswr_mode(handle, mode);

Inputs

	contrl[0] = 32	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
mode	intin[0]	Writing mode requested

1 = Replace
Both the 1 and the 0 bits in the graphics object replace the background (1's with current foreground color, 0's with color 0).
2 = Transparent
Only the 1 bits in the graphics object replace the background (with current foreground color)
3 = XOR
The 1 bits in the graphics object are colored with the complement of the current background color
4 = Reverse Transparent
Only the 0 bits in the graphics object replace the background (with current foreground color)
If number requested is out of range, number 1 (Replace) is selected.

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
mode_set	intout[0]	Writing mode actually selected

Set Input Mode

vsin__mode( ) Opcode=33

This function is used to set any of the four logical input devices (string, valuator, locator, or choice) to request or sample mode. The proper mode should be set before using any of these devices. This function returns the mode that was actually set in inout[0], but the C bindings do not make this information available in a C variable.

Devices required for

Screen

C Binding

int handle, device, mode;
vsin__mode(handle, device, mode);

Inputs

	contrl[0] = 33	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
device	intin[0]	Logical input device
		1 = Locator
		2 = Valuator
		3 = Choice
		4 = String
mode	intin[1]	Input mode for that device
		1 = Request
		2 = Sample

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
	intout[0]	Input mode selected

Inquire Current Polyline Attributes
vql_attributes( )   **Opcode=35**

This function returns information about the settings that affect line-drawing operations, including line-drawing pattern, line width, color, end styles and writing mode. The C binding does not return the information about the end styles (intout[3]–inout[4]) in a C variable.

Devices required for
All

C Binding
int handle, settings[4];
vql_attributes(handle, settings);

Inputs

	contrl[0] = 35	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 1	Number of points in ptsout
	contrl[4] = 5	Number of output integers in intout
settings[0]	intout[0]	Line type setting
settings[1]	intout[1]	Line-drawing pen (color index)
settings[2]	intout[2]	Writing mode
	intout[3]	End style of first point of line
	intout[4]	End style of last point of line
settings[3]	ptsout[0]	Line width setting
	ptsout[1]	0

See also
vsl_type( ), vsl_width( ), vsl_color( ), vsl_ends( ), vswr_mode( )

Inquire Current Polymarker Attribute

vqm_attributes( ) Opcode=36

This function returns information about the settings that affect marker drawing operations, including marker type, marker height, marker width, marker color, and writing mode. The C binding does not return the information about the marker width (ptsout[0]) in a C variable.

Devices required for

All

C Binding

int handle, settings[4];
vqm_attributes(handle, settings);

Inputs

	contrl[0] = 36	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 1	Number of points in ptsout
	contrl[4] = 3	Number of output integers in intout
settings[0]	intout[0]	Marker type setting
settings[1]	intout[1]	Marker drawing pen (color index)
settings[2]	intout[2]	Writing mode
settings[3]	ptsout[1]	Marker height setting (in pixels)

See also

vsm_type( ), vsm_height( ), vsm_color( ), vswr_mode( )

Inquire Current Fill Area Attributes
vqf_attributes( )   **Opcode = 37**

This function returns information about the settings that affect area fill operations, including interior style, fill style index, fill color, perimeter outlining, and writing mode. The C binding does not return the information about perimeter outlining (intout[4]) in a C variable.

Devices required for
All

C Binding
int handle, settings[4];
vqf_attributes(handle, settings);

Inputs

	contrl[0] = 37	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 5	Number of output integers in intout
settings[0]	intout[0]	Area fill interior style setting
settings[1]	intout[1]	Area fill drawing pen (color index)
settings[2]	intout[2]	Area fill style index
settings[3]	intout[3]	Writing mode
	intout[4]	Perimeter outlining status

See also
vsf_interior( ), vsf_style( ), vsf_color( ), vsf_perimeter( ), vswr_mode( )

Inquire Current Graphics Text Attributes
vqt_attributes( )   **Opcode = 38**

This function returns information about the settings that affect graphics-text operations, including current text face, color, baseline rotation, alignment, character and cell size, and writing mode.

Devices required for
All

C Binding
int handle, settings[10];
vqt_attributes(handle, settings);

Inputs
	contrl[0] = 38	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results
	contrl[2] = 2	Number of points in ptsout
	contrl[4] = 6	Number of output integers in intout
settings[0]	intout[0]	Current graphics-text face (font id)
settings[1]	intout[1]	Graphics-text pen (color index)
settings[2]	intout[2]	Angle of rotation of text baseline (in tenths of degrees, 0–3600)
settings[3]	intout[3]	Horizontal alignment setting
settings[4]	intout[4]	Vertical alignment setting
settings[5]	intout[5]	Writing mode
settings[6]	ptsout[0]	Character width
settings[7]	ptsout[1]	Character height
settings[8]	ptsout[2]	Character cell width
settings[9]	ptsout[3]	Character cell heigth

See also
vst_font( ), vst_height( ), vst_point( ), vst_color( ), vsl_alignment, vst_rotation, vswr_mode( )

Set Graphics Text Alignment

vst_alignment   **Opcode = 39**

This function controls the horizontal and vertical alignment points for graphics text. The horizontal alignment determines whether the text string will be centered or left- or right-justified. The vertical alignment determines whether the y coordinate of the text placement point refers to the character baseline, half line, ascent line, bottom line, descent line, or top line. The default alignment makes the graphics-text position the baseline of the leftmost character in the string.

Devices required for

All

C Binding

int handle, halign, valign, hresult, vresult;
vst_alignment(handle, halign, valign, &hresult, &vresult);

Inputs

	contrl[0] = 39	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
halign	intin[0]	The horizontal alignment requested
		0 = Left-justified (default)
		1 = Centered
		2 = Right-justified
valign	intin[1]	The vertical alignment requested
		0 = Baseline (default)
		1 = Half line
		2 = Ascent line
		3 = Bottom line
		4 = Descent line
		5 = Top line

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 2	Number of output integers in intout
hresult	intout[0]	The horizontal alignment actually set
vresult	intout[1]	The vertical alignment actually set

Open Virtual Workstation

v—opnvwk( )   **Opcode = 100**

This function opens a virtual workstation that allows the application to
share the physical screen device with other tasks. Each virtual workstation
has access to the full screen, but its graphics settings are maintained sepa-
rately from all of the others. The screen device drivers are part of the TOS
ROMs, so they do not have to be loaded in from disk when a virtual work-
station is opened. But if you wish to use disk-loaded fonts with your virtual
workstation, they must be properly identified in the ASSIGN.SYS file, and
the GDOS must be loaded. Like Open Workstation, this command allows
you to make initial settings for a number of graphics output functions. But
note that, unlike Open Workstation, the handle parameter does double duty
for this call. As an input, it should be set to the value of the current screen
device handle, which can be found by using the AES call graf—handle( ).
Upon return from this call, the handle parameter contains the virtual work-
station's device handle.

Devices required for
Screen

C Binding
int handle, work—in[12], work—out[57];
v—opnwk(work—in, &handle, work—out);

Inputs

	contrl[0] = 100	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 11	Number of input integers in intin
handle	contrl[6] = n	The physical screen device handle
		(obtained from graf—handle( ) call)

        The initial graphics output settings for the virtual workstation are speci-
fied by the contents of the intin array (which the bindings take from
work—in[ ]).

work—in[0]	intin[0]	Screen Device ID (from ASSIGN.SYS file)
		01 Default screen
		02 Lo-res color
		03 Medium-res color
		04 Hi-res monochrome
		05–10 Reserved for Atari expansion
work—in[1]	intin[1]	Line drawing pattern [vsl—type( )]
work—in[2]	intin[2]	Line pen number [vsl—color( )]
work—in[3]	intin[3]	Marker type [vsm—type( )]
work—in[4]	intin[4]	Marker pen number [vsm—color( )]
work—in[5]	intin[5]	Text font [vst—font( )]
work—in[6]	intin[6]	Text pen number [vst—color( )]
work—in[7]	intin[7]	Fill pattern type [vsf—interior( )]
work—in[8]	intin[8]	Fill pattern index [vsf—style( )]

work_in[9]	intin[9]	Fill pen number [vsf_color( )]
work_in[10]	intin[10]	NDC to RC transformation flag
		0 = Use Normalized Device Coordinates
		1 = Reserved for future use
		2 = Use Raster Coordinate system

Results

	contrl[2] = 6	Number of points in ptsout
	contrl[4] = 45	Number of output integers in intout
handle	contrl[6] = n	The device handle for this device
		(0 if device was not opened)

The results returned in the intout and ptsout arrays give a wide range of information about the output capabilities of the device.

work_out[0]	intout[0]	Maximum horizontal coordinate value (in pixels)
		(medium-res & hi-res = 639, lo-res = 319)
work_out[1]	intout[1]	Maximum Max. vertical coordinate value (in pixels)
		(hi-res = 399, medium-res & lo-res = 199)
work_out[2]	intout[2]	Device Coordinate units flag
		(1 = device doesn't support precise scaling)
		All ST screens return 0, showing that they support precise scaling
work_out[3]	intout[3]	Width of one pixel in microns
		(1/1000's of a millimeter)
		For display screens, horizontal component of aspect ratio. The values returned are: hi-res = 372, medium-res = 169, lo-res = 338
work_out[4]	intout[4]	Height of one pixel in microns
		For display screens, vertical component of aspect ratio. All screens return 372.
work_out[5]	intout[5]	Number of text font heights
		(0 = continuous scaling)
		There are 3 text heights in the system font, each of which can be printed double high.
work_out[6]	intout[6]	Number of line patterns (7)
work_out[7]	intout[7]	Number of line widths
		(All screens return 0—continuous scaling)
work_out[8]	intout[8]	Number of marker patterns (6)
work_out[9]	intout[9]	Number of marker sizes (8)
		(0 = continuous scaling)
work_out[10]	intout[10]	Number of text fonts supported by the device
		Only one system font is available in ROM

work_out[11]	intout[11]	Number of pattern fill styles (24)
work_out[12]	intout[12]	Number of crosshatch fill styles (12)
work_out[13]	intout[13]	Number of drawing pen colors available (the number of colors that can be displayed by the device at the same time) hi-res = 2, medium-res = 4, lo-res = 16
work_out[14]	intout[14]	Number of Generalized Drawing Primitives (GDPs)—how many of the 10 basic drawing commands are supported (all 10 on the ST)

| work_out[15] to work_out[24] | intout[15] to intout[24] | This part of the array holds a sequential list of code numbers for the first 10 GDPs supported. Each element holds one of the following code numbers: |

1 = Filled Rectangle or Bar (v_bar)—fill
2 = Circle Segment or Arc (v_arc)—line
3 = Filled Pie Slice (v_pieslice)—fill
4 = Filled Circle (v_circle)—fill
5 = Filled Ellipse (v_ellipse)fill
6 = Elliptical Arc (v_ellarc)—line
7 = Filled Elliptical Pie (v_ellpie)—fill
8 = Rounded Rectangle (v_rbox)—line
9 = Filled Rounded Rectangle (v_rfbox)—fill
10 = Justified Text (v_justified)—text
−1 = End of list

| work_out[25] to work_out[34] | intout[25] to intout[34] | This part of the array holds a sequential list of code numbers showing what category of graphics operation is performed by of each of the supported GDPs. This indicates what kind of graphics settings affects each of the supported commands. Each element holds one of the following code numbers: |

0 = Line drawing
1 = Marker drawing
2 = Graphics text
3 = Filled area
4 = No setting

The graphics category for each function on the ST can be found in the table of functions, above, at the end of each entry.

| work_out[35] | intout[35] | Color availability flag |

0 = Device is not capable of color output
1 = Device is capable of color output
    ST shows 0 for mono screen, 1 for color

work_out[36]	intout[36]	Text rotation availability flag 0 = Device is not capable of text rotation 1 = Device is capable of text rotation ST shows 1 for all res modes
work_out[37]	intout[37]	Area fill availability flag 0 = Device isn't capable of area fill 1 = Device is capable of area fill ST shows 1 for all res modes
work_out[38]	intout[38]	Cell array function availability flag 0 = Device can't perform cell array function 1 = Device can perform cell array function ST shows 0 for all res modes
work_out[39]	intout[39]	Total number of color choices available in the palette 0 = More than 32767 colors available 1 = Monochrome 2–32767 = Actual number of colors available ST hi-res = 2, medium-res & lo-res = 512
work_out[40]	intout[40]	Input devices available for the locator function 1 = Keyboard only 2 = Keyboard and mouse ST shows 2 for all res modes
work_out[41]	intout[41]	Input devices available for the valuator function 1 = Keyboard 2 = Other device ST shows 1 for all res modes
work_out[42]	intout[42]	Input devices available for the choice function 1 = Function keys on keyboard 2 = Some other key pad ST shows 1 for all res modes
work_out[43]	intout[43]	Input devices available for the string input function 1 = keyboard ST shows 1 for all res modes
work_out[44]	intout[44]	Workstation type 0 = Output only 1 = Input only 2 = Input and output 3 = Reserved for future use 4 = Metafile output ST shows 2 for all res modes
work_out[45]	ptsout[0]	Minimum character width (5)
work_out[46]	ptsout[1]	Minimum character height (4)
work_out[47]	ptsout[2]	Maximum character width (7)

work_out[48]	ptsout[3]	Maximum character height (13)
work_out[49]	ptsout[4]	Minimum line width (1)
work_out[50]	ptsout[5]	0
work_out[51]	ptsout[6]	Maximum line width (40)
work_out[52]	ptsout[7]	0
work_out[53]	ptsout[8]	Minimum marker width (15)
work_out[54]	ptsout[9]	Minimum marker height (11)
work_out[55]	ptsout[11]	Maximum marker width (120)
work_out[56]	ptsout[12]	Maximum marker height (88)

See also

v_clswk( ), v_opnwk( ), v_clsvwk( )

Close Virtual Workstation

v—clsvwk( )   Opcode=101

This call is used to terminate output to a virtual workstation and release its environment space.

Devices required for
Screen

C Binding
int handle;
v—clsvwk(handle);

Inputs

	contrl[0]	= 101	Opcode
	contrl[1]	= 0	Number of points in ptsin
	contrl[3]	= 0	Number of input integers in intin
handle	contrl[6]	= n	The workstation device handle

Results

contrl[2]	= 0	Number of points in ptsout
contrl[4]	= 0	Number of output integers in intout

See also
v—opnvwk( ), v—opnwk( ), v—clswk( )

Extended Inquire

vq_extnd( )   Opcode = 102

This function returns either the same information about about a graphics output device as the Open Workstation and Open Virtual Workstation calls, or some additional information about the device's output capabilities.

Devices required for
All

C Binding
int handle, info_flag, work_out[57];
v_opnwk(work_in, info_flag, work_out);

Inputs

	contrl[0] = 102	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
	contrl[6] = n	The (virtual) workstation device handle
info_flag	intin[0]	Type of information flag
		0 = Open Workstation values
		1 = Extended Inquire values

Results

	contrl[2] = 6	Number of points in ptsout
	contrl[4] = 45	Number of output integers in intout

The results returned in the intout and ptsout arrays give a wide range of information about the output capabilities of the device. If info_flag was set to 0, these values are the same as those returned by the Open Workstation or Open Virtual Workstation calls. (See v_opnwk( ) and v_opnvwk( ) for more details.) If info_flag is set to 1, the following information is returned. Note that 6 points and 45 integers are always returned, even though many of these values are undefined when the information type flag is set for Extended Inquire.

work_out[0]	intout[0]	Type of screen display
		0 = This device doesn't have a screen
		1 = Separate alphanumeric and graphics controllers, and separate display screens
		2 = Separate alpha and graphics controllers, with a common display screen.
		3 = Common alpha and graphics controller, with separate display memory.
		4 = Common alpha and graphics controller, common display memory. (The ST display is this type.)
work_out[1]	intout[1]	Number of background colors available (on the ST, the same as number of colors in palette—512 in color, 1 in monochrome.)

282

work_out[2]	intout[2]	Number of text special effects supported (31 on the ST, since 5 effects can be combined in that many ways).
work_out[3]	intout[3]	Raster scaling availability flag 0 = Scaling of rasters not supported (as is the case on the ST) 1 = Raster scaling supported
work_out[4]	intout[4]	Number of color bit planes (monochrome=1, medium-res=2, lo-res=4)
work_out[5]	intout[5]	Color palette lookup table flag 0 = Lookup table supported (as on color screens, where VDI drawing pen index numbers are different from hardware color register numbers) 1 = Lookup table not supported (as on ST monchrome screen)
work_out[6]	intout[6]	Performance factor, the number of 16 × 16 raster operations that can be performed per second (1000 for machines without blitter chip)
work_out[7]	intout[7]	Contour fill capability flag 0 = Contour fill not supported (All ST screens support contour fill, and return a value of 1)
work_out[8]	intout[8]	Character baseline rotation flag 0 = text characters cannot be rotated 1 = characters can be rotated in 90—degree increments only (all ST screens) 2 = Characters can be rotated at any angle
work_out[9]	intout[9]	Number of writing modes available (4)
work_out[10]	intout[10]	Highest input mode available 0 = no input 1 = request mode 2 = sample mode (all ST screens)
work_out[11]	intout[11]	Text alignment capability flag (all ST screens = 1, text can be aligned)
work_out[12]	intout[12]	Inking capability flag (all ST screens = 0, device can't ink)
work_out[13]	intout[13]	Rubberband line capability flag 0 = no rubberband lines (all ST screens) 1 = rubberband lines only 2 = rubberband lines and rectangles
work_out[14]	intout[14]	Maximum number of points for Polyline, Polymarker, or Filled Area (128)
work_out[15]	intout[15]	Maximum number of integers in intin (all ST screens = −1, no maximum)

work_out[16]	intout[16]	Number of mouse buttons (2)
work_out[17]	intout[17]	Wide line pattern capability flag
		0 = No patterns for wide lines
		(all ST screens)
		1 = wide lines can use line pattern
work_out[18]	intout[18]	Drawing modes for wide lines (0)
work_out[19]	intout[19]	Reserved for future use
to	to	
work_out[44]	intout[44]	
work_out[45]	ptsout[0]	Reserved for future use
to	to	
work_out[56]	ptsout[11]	

See also

v_opnwk( ), v_opnvwk( )

Contour Fill

v—contourfill   Opcode = 103

This function is used to fill an enclosed polygon with the current fill pattern and color. Filling proceeds in one of two modes. In outline mode, the fill spreads from an initial point in all directions, until it comes to an outline of a given color. In color mode, the fill spreads from the initial point until it reaches a color other than that contained in the initial point. The way in which the polygon is filled is affected by the general graphics settings and the fill settings:

Writing mode (vswr—mode)
Clipping rectangle (vs—clip)
Fill color (vsf—color)
Fill interior style (vsf—interior)
Fill style index (vsf—style)

Devices required for

Metafiles

C Binding

int handle, x, y, pen;
v—contourfill(handle, x, y, pen)

Inputs

	contrl[0]	= 103	Opcode
	contrl[1]	= 1	Number of points in ptsin
	contrl[3]	= 1	Number of input integers in intin
handle	contrl[6]	= n	The (virtual) workstation device handle
pen	intin[0]		Color index for polygon outline. If this value is negative, color mode is used rather than outline mode.
x	ptsin[0]		Horizontal coordinate of initial point
y	ptsin[1]		Vertical coordinate of initial point

Results

contrl[2]	= 0	Number of points in ptsout
contrl[4]	= 0	Number of output integers in intout

See also

vswr—mode( ), vs—clip( ), vsf—color( ), vsf—interior( ), vsf—style( )

Set Perimeter Fill Visibility

vsf_perimeter( )   Opcode=104

This funtion is used to turn fill outlining on or off. When perimeter visibility is on, a filled-shape drawing operation outlines the filled shape with a solid line drawn in the current fill color.

Devices required for
All

C Binding
int handle, vis_flag;
vis_set = vsf_perimeter(handle, vis_flag);

Inputs

	contrl[0] = 104	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
vis_flag	intin[0]	Visibility flag
		0 = no outline drawn
		1 = outline is visible

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
vis_set	intout[0]	Visibility mode selected

Get Pixel

v—get—pixel( )   Opcode = 105

This function returns the VDI color index and the ST color register number for a particular point on the screen.

Devices required for
None

C Binding
int handle, x, y, register, pen;
v—get—pixel(handle, x, y, register, pen);

Inputs

	contrl[0] = 105	Opcode
	contrl[1] = 1	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
x	ptsin[0]	Horizontal coordinate of point
y	ptsin[1]	Vertical coordinate of point

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 2	Number of output integers in intout
register	intout[0]	Actual hardware color register used to color the point.
pen	intout[1]	VDI drawing pen (color index) used to color the point

See also
vs—color( )

287

Set Graphics Text Set Special Effects
vst_effects( )   Opcode = 106

This function is used to designate which special effects will be used for printing graphics text. Text can be rendered as thickened (bold), light intensity (grayed or ghosted), skewed (italics), underlined, outlined, or any combination of these effects.

Devices required for
All

C Binding
int handle, effects, effects_set;
effects_set = vst_effects(handle, effects);

Inputs

	contrl[0] = 106	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
effects	intin[0]	Special effects setting. Each effect is described in a different bit of this word. Possible effects are:

Bit	Value	Effect
0	1	Thickened (bold)
1	2	Light intensity (ghosted/grayed)
2	4	Skewed (italics)
3	8	Underlined
4	16	Outlined
5	32	Shadow (not supported on ST)

To combine effects, add the value of the desired effects together (for example, a value of 10 indicates Underline (8) and light intensity (2) will be used together).

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
effects_set	intout[0]	Effects actually selected

Set Character Cell Height, Points Mode
vst_point( )   Opcode = 107

This command is used to set the character size of graphics text using points, a printing measurement equal to 1/72 inch. The character height requested encompasses the entire character cell, which may include some blank space at the top and bottom of the character. Since all point sizes will not be available for any given font, the VDI tries to match the requested size with the next smallest available font size. The function returns information about the point size selected and the character size and cell size in pixels.

Devices required for
All

C Binding
int handle, point, charw, charh, cellw, cellh, point_set;
point_set = vst_point(handle, point, &charw, &charh, &cellw, &cellh);

Inputs

	contrl[0] = 107	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
points	intin[0]	Character cell height (in points)

Results

	contrl[2] = 2	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
point_set	intout[0]	Cell height selected (in points)
charw	ptsout[0]	Character width selected (in pixels)
charh	ptsout[1]	Character height selected (in pixels)
cellw	ptsout[2]	Cell width selected (in pixels)
cellh	ptsout[3]	Cell height selected (in pixels)

See also
vst_height( )

Set Polyline End Styles

vsl_ends( ) Opcode=108

This function is used to specify how the ends of lines produced by the line-drawing functions will appear. Either end of a line—or both ends—may be squared off (the default), rounded, or have an arrowhead attached. Note that rounding off the end of line really only affects lines that are more than a few pixels wide. If the style requested by this call is not available, the squared end style (0) is selected.

Devices required for
All

C Binding
int handle, begin, end;
vsl_ends(handle, begin, end);

Inputs

	contrl[0] = 108	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 2	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
begin	intin[0]	End style for the beginning point
end	intin[1]	End style for the endpoint of the line
		0 = Squared (default)
		1 = Arrow
		2 = Rounded

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

Copy Raster, Opaque

vro_cpyfm( )   Opcode = 109

This function is used to copy a rectangular image from one area of memory to another. The source area, the destination area, or both may in display memory. The source area may overlap the destination area; this function will copy in the correct direction for preserving the source image. The image may be copied exactly, or it may be combined in various ways with existing image data in the destination area.

The VDI uses a data structure called a Memory Form Definition Block (MFDB) to describe the source and destination memory areas. This data structure contains information about the memory location of image data, the size of the image in pixels and memory words, the number of color planes, and format of the image, either standard (each color bit plane separate), or ST-specific (color planes interleaved into one large bit plane). For the purposes of this function, the source and destination forms must both be in ST-specific format.

Devices required for

Screen

C Binding

```
int handle, mode, points[8];
struct fdbstr {
    int   *fd_addr;          /* pointer to image data area */
    int   fd_w;              /* image width in pixels */
    int   fd_h;              /* image height in pixels */
    int   fd_wdwidth;        /* image width in words */
    int   fd_stand;          /* standard format flag */
    int   fd_nplanes;        /* number of color bit planes */
    int   fd_r1, fd_r2, fd_r3;  /* reserved for future use */
}source, destination;
vro_cpyfm (handle, mode, points, &source, &destination);
```

Inputs

	contrl[0] = 109	Opcode
	contrl[1] = 4	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
&source	contrl[7-8]	Long-word address of source MFDB
&destination	contrl[9-10]	Long-word address of destination MFDB
mode	intin[0]	Logic operation used to combine the source image with the desination. The logic operations are described below, using the symbol S to refer to the source image, D to refer to the starting destination image, and D1 to refer to the resulting destination image:

Mode No.	Logic Operation	Description
0	D1 = 0	Clear destination block (all 0's)
1	D1 = S AND D	
2	D1 = S AND (NOT D)	
3	D1 = S	Replace mode
4	D1 = (NOT S) AND D	Erase mode
5	D1 = D	Destination unchanged
6	D1 = S XOR D	XOR mode
7	D1 = S OR D	Transparent mode
8	D1 = NOT (S OR D)	
9	D1 = NOT (S XOR D)	
10	D1 = NOT D	
11	D1 = S OR (NOT D)	
12	D1 = NOT S	
13	D1 = (NOT S) OR D	Reverse transparent mode
14	D1 = NOR (S AND D)	
15	D1 = 1	Fill destination block (all 1's)
points[0]	ptsin[0]	Left edge of source rectangle
points[1]	ptsin[1]	Top edge of source rectangle
points[2]	ptsin[2]	Right edge of source rectangle
points[3]	ptsin[3]	Bottom edge of source rectangle
points[4]	ptsin[4]	Left edge of destination rectangle
points[5]	ptsin[5]	Top edge of destination rectangle
points[6]	ptsin[6]	Right edge of destination rectangle
points[7]	ptsin[7]	Bottom edge of destination rectangle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

Transform Form

vr—trn—fm( )   Opcode = 110

This function is used to change a Memory Form Definition Block whose image data is in standard format (each color bit plane separate) to ST-specific format (all color bit planes interleaved), or vice versa.

The function converts the number of bit planes specified in the source form to the opposite format of that specified in the source form. It changes the format flag in the destination form, but does not change any other fields of the destination form. Note that the source and destination forms may be the same (known as an in-place transformation). Transforming a large form in place may be significantly slower than using two separate forms.

Devices required for

Screen

C Binding

```
int handle;
struct fdbstr {
    int  *fd—addr;        /* pointer to image data area */
    int  fd—w;            /* image width in pixels */
    int  fd—h;            /* image height in pixels */
    int  fd—wdwidth;      /* image width in words */
    int  fd—stand;        /* standard format flag */
    int  fd—nplanes;      /* number of color bit planes */
    int  fd—r1, fd—r2, fd—r3;  /* reserved for future use */
}source, destination;
vr—trnfm(handle, &source, &destination)
```

Inputs

	contrl[0] = 110	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
&source	contrl[7-8]	Long-word address of source MFDB
&destination	contrl[9-10]	Long-word address of destination MFDB

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

Set Mouse Form

vsc_form( )   **Opcode = 111**

This function is used to change the shape of the mouse pointer that appears on screen. It lets you to define the portion of the 16 × 16 pixel area to be drawn in the foreground color, the background color, and a transparent area through which the existing background may be seen.

You must supply two arrays of image data. The first, called the *mask*, defines the opaque area of the the pointer without regard to color information. The second is the image data itself. Bit positions within the mask that contain a 1 are considered to be *inside* the pointer. If the corresponding image data bit also contains a 1, that pixel will be colored in using the foreground color. If the corresponding image data bit contains a 0, that pixel will be color in using the background color. Mask bit positions that contain a 0 are considered to the *outside* the pointer image, or transparent, and if the corresponding image data bit contains a 0, these pixles will be represented on screen by whatever background data happens to be there.

You must also define a *hot spot* for the pointer. Although the mouse pointer may be up to 16 × 16 pixels in size, the VDI always considers it to be located at a single point on screen. The hot spot is the one pixel within the mouse pointer which is considered to be its true location. For example, the tip of the arrow-shaped mouse pointer is its hot spot, so, to activate an icon, you must position the tip of the arrow on it when you press the mouse button.

Devices required for

Screen

C Binding

int handle, pt_data[37];
vsc_form(handle, pt_data);

Inputs

	contrl[0] = 111	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 37	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
pt_data[0]	intin[0]	X coordinate of hot spot
pt_data[1]	intin[1]	Y coordinate of hot spot
pt_data[2]	intin[2]	Reserved for future use (must be 1)
pt_data[3]	intin[3]	Background pen (usually 0)
pt_data[4]	intin[4]	Foreground pen (usually 1)
pt_data[5–20]	intin[5–20]	16 words of mask data
pt_data[21–36]	intin[21–36]	16 words of image data. Each word represents a line of 16 pixels, with the first word being the top line, the second being the second, and so on. The least-significant bit in each word represents the rightmost pixel, and the most-signficant bit in each word, the leftmost.

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

Set User-Defined Fill Pattern
vsf_udpat( )   Opcode = 112

This function is used to supply the image data for the user-defined fill pattern, style 4 of vsf_interior( ). This fill pattern is a 16 × 16 pixel image, either monochrome or multicolor. For a monochrome fill pattern, 16 words of image data are used to describe the image on 16 lines. Each bit represents either a pixel of foreground color (1) or background color (0). The foreground color used is the current fill color.

To describe a multicolor fill pattern, you must use 16 words of image data for each color bit plane. Each plane contains a single bit of color information for each pixel, and in order to obtain complete color information for a single pixel, you must combine the values for each corresponding bit in all of the planes. For example, to find the color of the top, left pixel, you must combine the first bit of each bit plane. The first bit plane contains all of the least-significant bits, and each subsequent plane holds the next most significant bit.

Devices required for
Screen, Printer, Metafile

C Binding
int handle, planes, pat_dat[16*PLANES];
vsf_udpat(handle, pat_dat, planes);

Inputs

	contrl[0] = 112	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 16X	Number of input integers in intin (16 * number of planes)
handle	contrl[6] = n	The (virtual) workstation device handle
planes		The number of color bit planes (contrl[3]/16)
pat_dat[0] to pat_dat[15]	intin[0] to intin[15]	First bit plane of fill pattern
. . .	. . .	
pat_dat[n−15] to pat_dat[n]	intin[n−15] to intin[n]	Last bit plane of fill pattern

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also
vsf_interior( )

Set User-Defined Line Style

vsl_udsty( )   **Opcode = 113**

This function is used to supply the image data for the user-defined line pattern, pattern 7 of vsl_type( ). The line pattern data takes the form of a single 16-bit word, each bit of which represents a pixel drawn in either the foreground color (1) or background color (0).

Devices required for
Screen, Metafile

C Binding
int handle, pattern;
vsl_udsty(handle, pattern);

Inputs

	contrl[0] = 113	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
pattern	intin[0]	The line drawing pattern, expressed as a 16-bit word of image data

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also
vsl_type( )

Fill Rectangle

vr_recfl( ) Opcode=114

This function draws a rectangle filled with the current fill pattern and color. The rendering of the filled figure is affected by the general graphics settings and the fill settings, except for perimeter outlining (the rectangle created by vr_recfl( ) is never outlined):

Writing mode (vswr_mode)
Clipping rectangle (vs_clip)
Fill color (vsf_color)
Fill interior style (vsf_interior)
Fill style index (vsf_style)

This command is most often used to clear large rectangular areas on the screen quickly.

Devices required for

Screen, Metafile

C Binding

int handle, points[4];
vr_recfl(handle, points);

Inputs

	contrl[0] = 114	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
points[0]	ptsin[0]	Left coordinate of the rectangle
points[1]	ptsin[1]	Top coordinate of the rectangle
points[2]	ptsin[2]	Right coordinate of the rectangle
points[3]	ptsin[3]	Bottom coordinate of the rectangle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 1	Number of output integers in intout

See also

vswr_mode( ), vs_clip( ), vsf_color( ), vsf_interior( ), vsf_style( )

Inquire Input Mode

vqin_mode( )   Opcode = 115

This function is used to determine the input mode used by one of the logical input devices (locator, string, valuator, or choice).

Devices required for
Screen

C Binding
int handle, device, mode;
vqin_mode(handle, device, &mode);

Inputs

	contrl[0] = 115	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
device	intin[0]	Logical device to check
		1 = Locator
		2 = Valuator
		3 = Choice
		4 = String

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
mode	intout[0]	Input mode of device
		1 = Request
		2 = Sample

See also
vsin_mode( )

Inquire Text Extent

vqt_extent( )   **Opcode = 116**

Devices required for
Screen, Printer, Plotter

C Binding
```
int handle, points[8];
char string;
vqt_extent(handle, &string, points);
```

Inputs

	contrl[0] = 116	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = s	Number of characters in text string
handle	contrl[6] = n	The (virtual) workstation device handle
string[0]	intin[0]	First character of text string. Text is formatted with one character per memory word, with each character occupying the low byte of the word.
•	•	
•	•	
•	•	
string[s]	intin[s]	Last character of text string

Results

	contrl[2] = 4	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout
points[0]	ptsout[0]	Horizontal offset of point 1
points[1]	ptsout[1]	Vertical offset of point 1
points[2]	ptsout[2]	Horizontal offset of point 2
points[3]	ptsout[3]	Vertical offset of point 2
points[4]	ptsout[4]	Horizontal offset of point 3
points[5]	ptsout[5]	Vertical offset of point 3
points[6]	ptsout[6]	Horizontal offset of point 4
points[7]	ptsout[7]	Vertical offset of point 4

Points 1, 2, 3, and 4 refer to the bottom left, bottom right, top right, and top left corners of the text string, respectively. Point 1 is located at the origin for text strings that are rotated 0 degrees, point 2 is located at the origin for strings that are rotated 270 degrees, point 3 is located at the origin for strings that are rotated 180 degrees, and point 4 is at the origin when the string is rotated 90 degrees. (See Figure 7-2.)

Inquire Character Cell Width

vqt_width( ) Opcode=117

This function can be used to learn the character cell width of a particular character in the current text font (without making allowance for special effects or baseline rotation). The character cell may include some of the blank space surrounding the character.

Devices required for
All

C Binding
int handle, char, cellw, left_offset, right_offset, status;
status = vqt_width(handle, char, &cellw, &left_offset, &right_offset);

Inputs

	contrl[0] = 117	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
char	intin[0]	Character whose width is inquired, formatted so ASCII value is in the low byte of the word.

Results

	contrl[2] = 3	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
status	intout[0]	Character about which information is returned (-1 if character requested was not available)
cellw	ptsout[0]	Width of the character cell (in pixels)
	ptsout[1]	0
left_offset	ptsout[2]	Offset of the left side of the character from the left edge of the character cell
	ptsout[3]	0
right_offset	ptsout[4]	Offset of the right side of the character from the right edge of the character cell
	ptsout[5]	0

Exchange Timer Interrupt Vector

vex_timv( )   Opcode = 118

This function allows you to add your own machine-language program to the
ST timer interrupt handler that executes every fixed period known as a timer
tick. Your routine should preserve all registers, should not call any non-
reentrant ROM routines, and should end with an RTS instruction. The func-
tion returns the address of the normal entry point of the system timer
routine, so that your routine may call that routine when it is finished.

Devices required for

Screen

C Binding

```
int handle, tick-length;
int *new_addr, *old_addr
vex_timv(handle, new_addr, old_addr, &tick_length);
```

Inputs

	contrl[0] = 118	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] =	The physical screen device handle (ob-tained from graf_handle( ) call)
new_addr	contrl[7–8]	Long-word address of the user's timer routine

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
old_addr	contrl[9–10]	Long-word address of the normal system timer routine
tick_length	intout[0]	Length of time between timer ticks (in milliseconds)

Load Fonts

vst_load_fonts( )   Opcode=119

This function is used to load disk-based text fonts. In order to load disk-based fonts on the ST, the GDOS extension (GDOS.PRG) must be loaded, usually by placing the program in the AUTO folder of the disk with which the system is started. Furthermore, the filenames of the fonts that available for each device driver must be listed in a file called ASSIGN.SYS, located in the top directory of the boot disk. Fonts cannot be loaded selectively; all available fonts will be loaded at the same time.

Devices required for
Screen

C Binding
int handle, select, fonts_loaded;
fonts_loaded = vst_load_fonts(handle, select);

Inputs

	contrl[0] = 119	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
select	intin[0] = 0	Reserved for future use, set to 0
		(may be used to selectively load fonts in a future version of GEM)

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
fonts_loaded	intout[0]	Number of new fonts made available

See also
vst_unload_fonts( )

Unload Fonts
vst_unload_fonts( )   Opcode = 120

This function is used to terminate the availability of disk-loaded fonts to a particular device. If no other workstation is using those fonts, this function also frees up the memory taken up by those fonts. You should unload disk-based fonts whenever you are through using them.

Devices required for
Screen

C Binding
int handle, select;
vst_unload_fonts(handle, select);

Inputs

	contrl[0] = 120	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
select	intin[0] = 0	Reserved for future use, set to 0 (may be used to selectively load fonts in a future version of GEM)

Results

contrl[2] = 0	Number of points in ptsout

See also
vst_load_fonts( )

Copy Raster, Transparent

vrt_cpyfm( )   Opcode = 121

This function is used to copy a single-plane rectangular bit image to a destination memory area (usually in screen memory) that can be made up of several color bit planes. The VDI uses a data structure called a Memory Form Definition Block (MFDB) to describe the source and destination memory areas. This data structure contains information about the memory location of image data, the size of the image in pixels and memory words, the number of color planes, and format of the image, either standard (each color bit plane separate), or ST-specific (color planes interleaved into one large bit plane). For the purposes of this function, the source and destination forms must both be in ST-specific format.

The call lets you specify the pen color that will be used to draw both the foreground (one bits) and the background (zero bits), so the image can be drawn in any color combination that you wish. The image may be copied directly, or combined in various ways with the existing image data in the destination area. Note that the writing modes used to combine the image are not the same ones used by vro_cpyfm( ), but rather the more limited set offered by vswr_mode( ).

Devices required for
Screen

C Binding
```
int handle, mode, points[8], pens[2];
struct fdbstr {
    int    *fd_addr;        /* pointer to image data area */
    int    fd_w;           /* image width in pixels */
    int    fd_h;           /* image height in pixels */
    int    fd_wdwidth;     /* image width in words */
    int    fd_stand;       /* standard format flag */
    int    fd_nplanes;     /* number of color bit planes */
    int    fd_r1, fd_r2, fd_r3;  /* reserved for future use */
}source, destination;
vrt_copyfm(handle, mode, points, source, destination, pens);
```

Inputs

	contrl[0] = 121	Opcode
	contrl[1] = 4	Number of points in ptsin
	contrl[3] = 3	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
source	contrl[7-8]	Long-word address of the source Memory Form Definition Block (MFDB)
destination	contrl[9-10]	Long-word address of the destination Memory Form Definition Block (MFDB)
mode	intin[0]	The mode that determines how the source combines with the destination:

		1 = Replace
		2 = Transparent
		3 = XOR
		4 = Reverse Transparent
pen[0]	intin[1]	The VDI pen color (index) for the 1 bits in the image data (foreground).
pen[1]	intin[2]	The VDI pen color (index) for the 0 bits in the image data (background)
points[0]	ptsin[0]	Left edge of source rectangle
points[1]	ptsin[1]	Top edge of source rectangle
points[2]	ptsin[2]	Right edge of source rectangle
points[3]	ptsin[3]	Bottom edge of source rectangle
points[4]	ptsin[4]	Left edge of destination rectangle
points[5]	ptsin[5]	Top edge of destination rectangle
points[6]	ptsin[6]	Right edge of destination rectangle
points[7]	ptsin[7]	Bottom edge of destination rectangle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also

vro_cprfm( )

Show Mouse Pointer

v_show_c( )   Opcode = 122

This function is used to display the mouse pointer, which tracks the movement of the mouse on screen. Whether or not a call to this function actually displays the pointer depends on how many times Hide Mouse Pointer (v_hide_c) has been called previously. Each time v_hide_c( ) is called, pointer visibility is pushed down one level further. Therefore, if v_hide_c( ) is called twice, v_show_c( ) must also be called twice before the pointer becomes visible. This function provides a reset flag, however, which resets the counter that keeps track of how many times the pointer has been hidden. By using this flag, you may specify that the pointer become visible regardless of the level at which it was hidden.

Devices required for
Screen

C Binding
int handle, reset;
v_show_c(handle, reset);

Inputs

	contrl[0] = 122	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
reset	intin[0]	Reset flag

                    0 = Reset counter, and display
                        pointer regardless of number of
                        times hidden
                  <> 0 = Move pointer visibility up one
                        level, and display if only hidden
                        once.

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

See also
v_hide_c( )

Hide Mouse Pointer

v_hide_c( )   Opcode=123

This function removes the mouse pointer that tracks the mouse movements on the screen. Each time this function is called, a counter increments the level at which the pointer is hidden, so that an equal number of v_show_c( ) calls must be made before the pointer becomes visible again.

Devices required for

Screen

C Binding

int handle;
v_hide_c(handle);

Inputs

	contrl[0] = 123	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout

See also

v_show_c( )

Sample Mouse Button State

vq_mouse( )   Opcode=124

This function is used to discover if either or both mouse buttons are currently being pressed. It also returns the current screen position of the mouse pointer.

Devices required for
Screen, Plotter

C Binding
int handle, button, x, y;
vq_mouse(handle, &button, &x, &y);

Inputs

	contrl[0] = 124	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 1	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
button	intout[0]	Mouse button status:
		0 = No buttons pressed
		1 = Left button pressed
		2 = Right button pressed
		3 = Both buttons pressed
x	ptsout[0]	Horizontal position of the mouse pointer
y	ptsout[1]	Vertical position of the mouse pointer

Exchange Button Change Vector

vex_butv( )   Opcode = 125

This function allows you to add your own machine language program to the
ST mouse button interrupt handler that executes every time that a mouse
button is pressed. Your routine should preserve all registers, should not call
any non-reentrant ROM routines, and should end with an RTS instruction.
At the point that your program executes, the mouse button status is con-
tained in register D0, represented in the same manner as in vq_mouse( ).
The function returns the address of the normal entry point of the system
timer routine, so that your routine may call that routine when it is finished.

Devices required for
Screen

C Binding
int handle;
int *new_addr, *old_addr
vex_timv(handle, new_addr, old_addr);

Inputs

	contrl[0] = 125	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The physical screen device handle (obtained from graf_handle( ) call)
new_addr	contrl[7-8]	Long-word address of the user's mouse button routine.

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout
old_addr	contrl[9-10]	Long-word address of the normal system mouse button routine.

Exchange Mouse Movement Vector
vex_motv( )   Opcode=126

This function allows you to add your own machine language program to the ST mouse movement interrupt handler that executes every time that the mouse changes position. At the time your program executes, register D0 contains the horizontal position of the mouse, and the D1 contains the vertical position. Your routine should preserve all registers, should not call any non-reentrant ROM routines, and should end with an RTS instruction. The function returns the address of the normal entry point of the system timer routine, so that your routine may call that routine when it is finished.

Devices required for
Screen

C Binding
int handle;
int *new_addr, *old_addr
vex_motv(handle, new_addr, old_addr);

Inputs

	contrl[0] = 126	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The physical screen device handle (obtained from graf_handle( ) call)
new_addr	contrl[7-8]	Long-word address of the user's mouse movement routine.

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout
old_addr	contrl[9-10]	Long-word address of the normal system mouse movement routine.

Exchange Cursor Change Vector
vex_curv( )   Opcode=127

This function allows you to add your own machine language program to the
ST mouse pointer interrupt handler that executes every time that the mouse
pointer is to be redrawn. At the point at which your code executes, the hori-
zontal position of the mouse pointer is stored in register D0, and its vertical
poisiton in register D1. Your routine should preserve all registers, should not
call any non-reentrant ROM routines, and should end with an RTS instruc-
tion. The function returns the address of the normal entry point of the sys-
tem timer routine, so that your routine may call that routine when it is
finished.

Devices required for
Screen

C Binding
int handle;
int *new_addr, *old_addr
vex_curv(handle, new_addr, old_addr);

Inputs
	contrl[0] = 127	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The physical screen device handle (obtained from graf_handle( ) call)
new_addr	contrl[7-8]	Long-word address of the user's mouse pointer routine.

Results
	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 0	Number of output integers in intout
old_addr	contrl[9-10]	Long-word address of the normal system mouse pointer redraw routine.

312

Sample Keyboard State Information

vq__key__s( )   **Opcode = 128**

This function is used to learn whether or not the Control, Left Shift, Right Shift, and/or Alt keys are currently being pressed.

Devices required for

Screen

C Binding

int handle, key;
vq__key__s(handle, &key);

Inputs

	contrl[0] = 128	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 1	Number of output integers in intout
key	intout[0]	Keypress status

The status for each key is returned in one of the four low bits of this word. Bits are assigned as follows:

Bit	Value	Key
0	1	Right Shift
1	2	Left Shift
2	4	Control
3	8	Alt

A one bit in any of these places means the key is pressed. For example, a value of 10 means that both the Alt key (8) and the Left Shift (2) are pressed.

Set Clipping Rectangle

vs_clip( )   Opcode=129

This function is used to turn clipping on and off. When clipping is on, output of all of the VDI graphics functions is restricted to a particular rectangular area. Output directed to areas outside of that rectangle is ignored. Clipping is particularly useful for confining output to the within the boundaries of a window.

Devices required for
Screen, Printer, Metafile

C Binding
int handle, clip_flag, points[4];
vs_clip (handle, clip_flag, points);

Inputs

	contrl[0] = 129	Opcode
	contrl[1] = 2	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
clip_flag	intin[0]	Clipping Flag
		0 = clipping off
		<> 0 = clipping on
points[0]	ptsin[0]	Left edge of clipping rectangle
points[1]	ptsin[1]	Top edge of clipping rectangle
points[2]	ptsin[2]	Right edge of clipping rectangle
points[3]	ptsin[3]	Bottom edge of clipping rectangle

Results

contrl[2] = 0	Number of points in ptsout
contrl[4] = 0	Number of output integers in intout

Inquire Face Name and Index

vqt_name( )   **Opcode = 130**

This function returns a character string containing the name and style information about a text font. It also returns the font ID number, which is needed to set this font as the current graphics text font (with a call to vst_font).

Devices required for

Screen, Printer, Plotter

C Binding

```
int handle, number, id;
char name[32];
id = vqt_name (handle, number, name);
```

Inputs

	contrl[0] = 130	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 1	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle
number	intin[0]	Font number. Font numbers are arbitrary numbers that range from 1 (the system font) to the maximum number of fonts. They are assigned in numeric order of ID numbers, so if there are three fonts with ID numbers of 1, 50, and 12, the font with ID number 1 will have a font number of 1, ID 12 will be font 2, and ID 50 will be font 3.

Results

	contrl[2] = 0	Number of points in ptsout
	contrl[4] = 33	Number of output integers in intout
id	intout[0]	Font ID number. This number is actually part of the font information itself, and is assigned when the font is created. It is needed to set this font as the current graphics text font when calling vst_font( ).
name[0–31]	intout[1–32]	The text name string. This string is formatted so that each character is set into a separate 16-bit member of the intout array, with the ASCII value of the character in the low byte, and a zero in the high byte. The first 16 characters of the string contain the name of the fonts, and the last 16 describe its thickness and style (such as whether this is a bold or italic variation).

See also

vst_font( )

Inquire Current Face Information

vqt_font_info( )   Opcode = 131

This function returns information about the size of the current graphics text font, including information about the size changes brought about by special effects.

Devices required for

All

C Binding

int handle, minchar, maxchar, maxwidth, distances[5], effects[3];
vqt_font_info(handle, &minchar, &maxchar, distanches, &maxwidth, effects);

Note: Some versions of the Alcyon and Megamax bindings assign this function the name vqt_fontinfo( ), instead of vqt_font_info( ).

Inputs
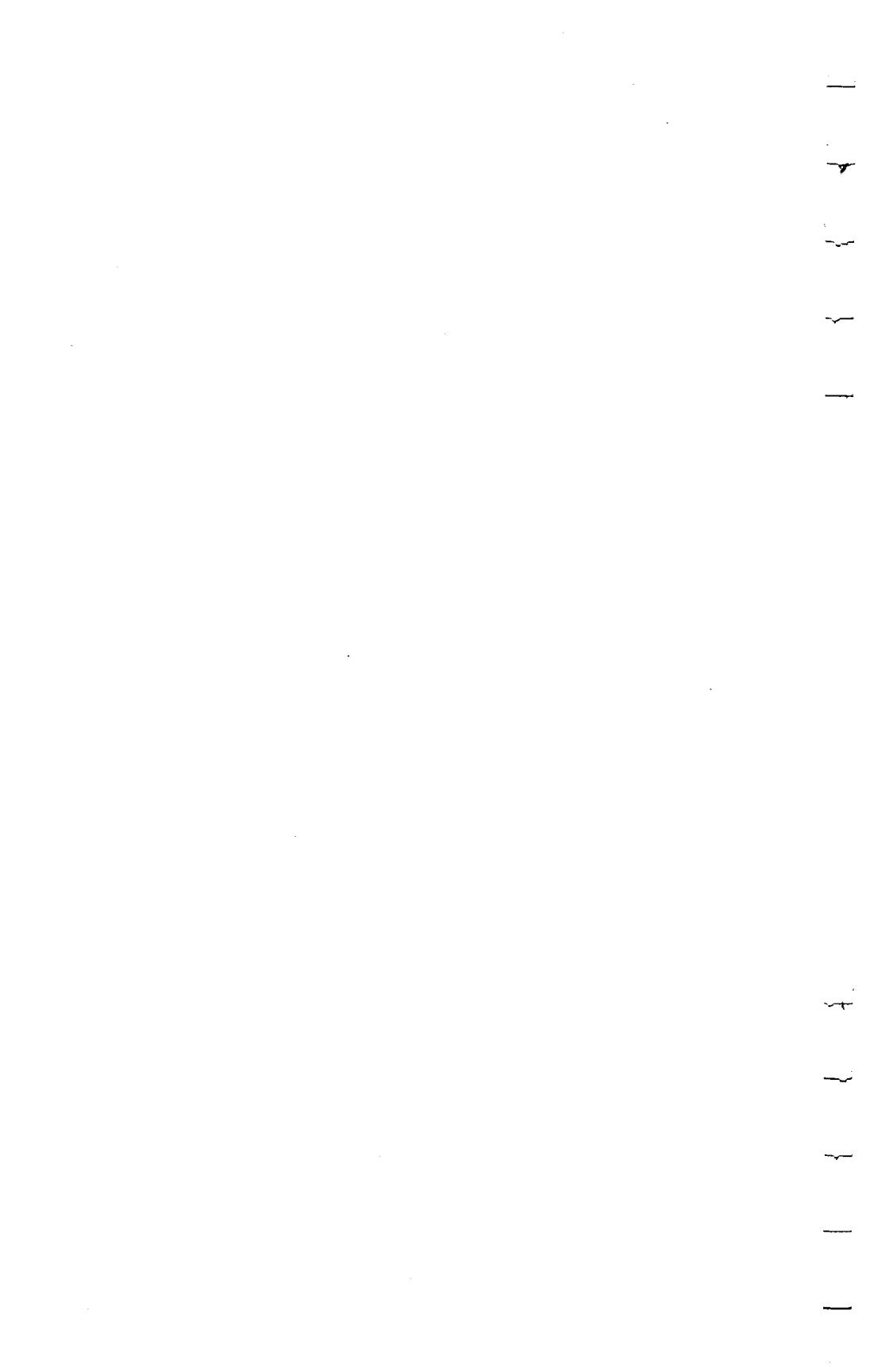
	contrl[0] = 131	Opcode
	contrl[1] = 0	Number of points in ptsin
	contrl[3] = 0	Number of input integers in intin
handle	contrl[6] = n	The (virtual) workstation device handle

Results

	contrl[2] = 5	Number of points in ptsout
	contrl[4] = 2	Number of output integers in intout
minchar	intout[0]	The ASCII character number of the first character in this type face.
maxchar	intout[1]	The ASCII character number of the last character in this type face.
maxwidth	ptsout[0]	The maximum character cell width in this type face, not including special effects.
distances[0]	ptsout[1]	The distance from the baseline to the bottom line.
effects[0]	ptsout[2]	The total increase in character width due to current special effects.
distances[1]	ptsout[3]	The distance from the baseline to the descent line.
effects[1]	ptsout[4]	The increase in character width on the left due to current special effects.
distances[2]	ptsout[5]	The distance from the baseline to the half line.
effects[2]	ptsout[6]	The increase in character width on the right due to current special effects.
distances[3]	ptsout[7]	The distance from the baseline to the ascent line.
	ptsout[8]	0
distances[4]	ptsout[9]	The distance from the baseline to the top line.

Appendix B

Extended Keyboard Codes

The VDI string input functions (vrq_string and vsm_string) may return a two-byte value for every key pressed, rather than a simple one-byte ASCII code. The first byte of this keycode is generally a unique key identifier that refers to the physical key struck, regardless of shift key combinations. The second byte is usually the ASCII value of the key combination, which does depend on the state of the shift keys (Shift, Control, and Alt). The following table shows the keycodes, as 4-digit hexadecimal numbers, for all key and shift combinations..

Main Keyboard

Unshifted		Shift		CTRL	ALT
a	1E61	A	1E41	1E01	1E00
b	3062	B	3042	3002	3000
c	2E63	C	2E43	2E03	2E00
d	2064	D	2044	2004	2000
e	1265	E	1245	1205	1200
f	2166	F	2146	2106	2100
g	2267	G	2247	2207	2200
h	2368	H	2348	2308	2300
i	1769	I	1749	1709	1700
j	246A	J	244A	240A	2400
k	256B	K	254B	250B	2500
l	266C	L	264C	260C	2600
m	326D	M	324D	320D	3200
n	316E	N	314E	310E	3100
o	186F	O	184F	180F	1800
p	1970	P	1950	1910	1900
q	1071	Q	1051	1011	1000
r	1372	R	1352	1312	1300
s	1F73	S	1F53	1F13	1F00
t	1474	T	1454	1414	1400
u	1675	U	1655	1615	1600
v	2F76	V	2F56	2F16	2F00
w	1177	W	1157	1117	1100
x	2D78	X	2D58	2D18	2D00
y	1579	Y	1559	1519	1500
z	2C7A	Z	2C5A	2C1A	2C00

1	0231	!	0221	0211	7800
2	0332	@	0340	0300	7900
3	0433	#	0423	0413	7A00
4	0534	$	0524	0514	7B00
5	0635	%	0625	0615	7C00
6	0736	^	075E	071E	7D00
7	0837	&	0826	0817	7E00
8	0938	*	092A	0918	7F00
9	0A39	(	0A28	0A19	8000
0	0B30	)	0B29	0B10	8100
-	0C2D	_	0C5F	0C1F	8200
=	0D3D	+	0D2B	0D1D	8300
`	2960	~	297E	2900	2960
\	2B5C			2B7C	2B1C	2B5C
[	1A5B	{	1A7B	1A1B	1A5B
]	1B5D	}	1B7D	1B1D	1B5D
;	273B	:	273A	271B	273B
;	273B	:	273A	271B	273B
'	2827	"	2822	2807	2827
,	332C	<	333C	330C	332C
.	342E	>	343E	340E	342E
/	352F	?	353F	350F	352F
Space	3920		3920	3900	3920
Esc	011B		011B	011B	011B
Backspace	0E08		0E08	0E08	0E08
Delete	537F		537F	531F	537F
Return	1C0D		1C0D	1C0A	1C0D
Tab	0F09		0F09	0F09	0F09

Cursor Pad

Unshifted		**Shift**	**CTRL**	**ALT**
Help	6200	6200	6200	(screen print)
Undo	6100	6100	6100	6100
Insert	5200	5230	5200	(left mouse button)
Clr/Home	4700	4737	7700	(right mouse button)
Up-Arrow	4800	4838	4800	(move mouse up)
Dn-Arrow	5000	5032	5000	(move mouse dn)
Rt-Arrow	4B00	4B34	7300	(move mouse rt)
Lft-Arrow	4D00	4D36	7400	(move mouse lft)
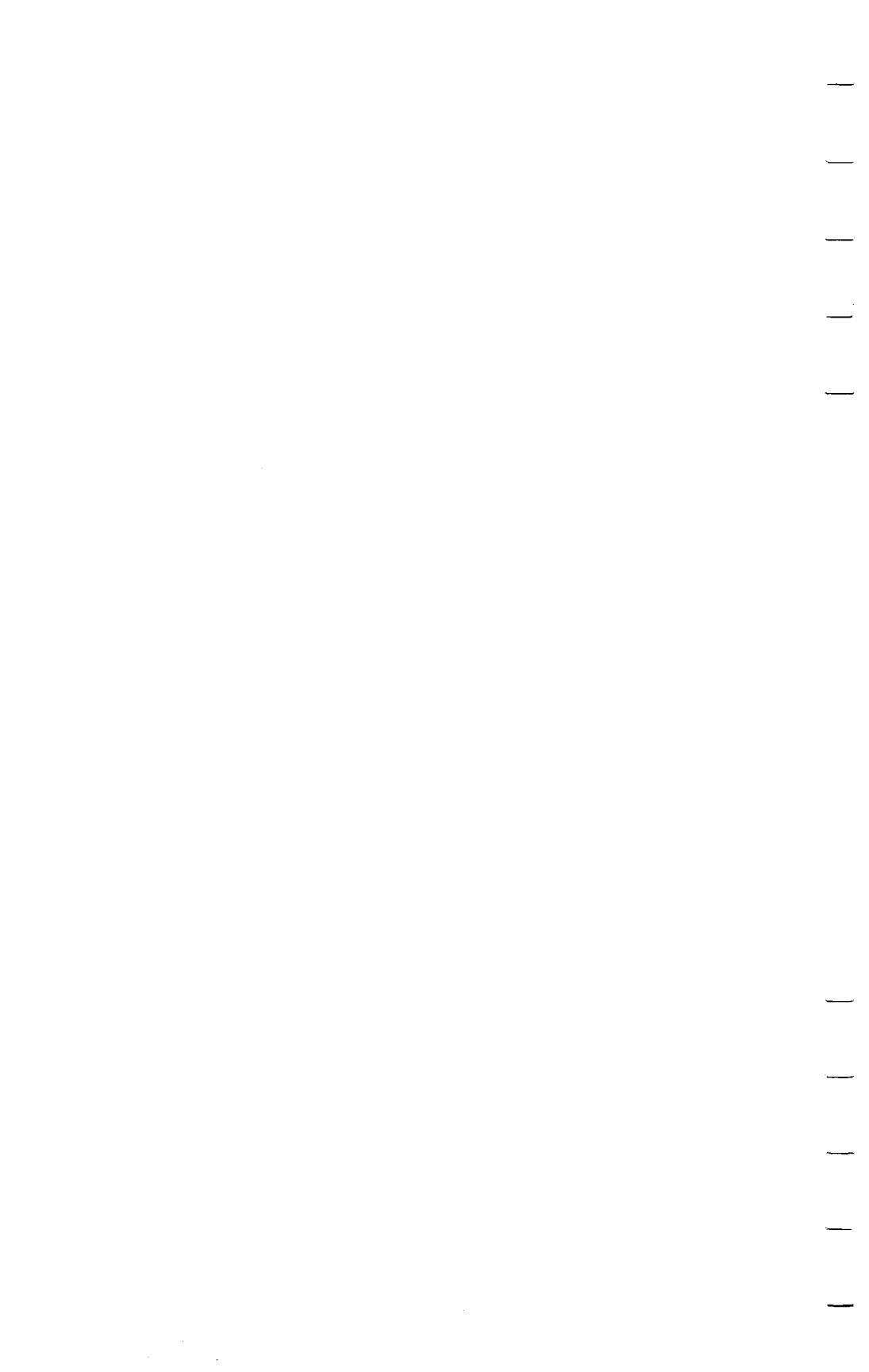
Numeric Pad

Unshifted		Shift	CTRL	ALT
(	6328	6328	6308	6328
)	6429	6429	6409	6429
/	652F	652F	650F	652F
*	662A	662A	660A	662A
–	4A2D	4A2D	4A1F	4A2D
+	4E2B	4E2B	4E0B	4E2B
.	712E	712E	710E	712E
Enter	720D	720D	720A	720D
0	7030	7030	7010	7030
1	6D31	6D31	6D11	6D31
2	6E32	6E32	6E00	6E32
3	6F33	6F33	6F13	6F33
4	6A34	6A34	6A14	6A34
5	6B35	6B35	6B15	6B35
6	6C36	6C36	6C1E	6C36
7	6737	6737	6717	6737
8	6838	6838	6818	6838
9	6939	6939	6919	6939

Function Keys

Unshifted		Shift	CTRL	ALT
F1	3B00	5400	3B00	3B00
F2	3C00	5500	3C00	3C00
F3	3D00	5600	3D00	3D00
F4	3E00	5700	3E00	3E00
F5	3F00	5800	3F00	3F00
F6	4000	5900	4000	4000
F7	4100	5A00	4100	4100
F8	4200	5B00	4200	4200
F9	4300	5C00	4300	4300
F10	4400	5D00	4400	4400

Appendix C

VDI Font Files

Font files for VDI disk-based fonts are divided into

four sections. The first, called the *font header*, contains information about the font such as first and last character in the font, font size, font name, and so on. The font header is 87 bytes long, and is laid out in the following format:

Byte Number	Description
0–1	Font ID. This value is used by the vst_font( ) function to make this the current graphics text font. It's one of the values returned by the vqt_name( ) function.
2–3	Font size (in points).
4–35	Font name and style information. This is a 32-character text string, with each character occupying the low byte of its own 16-bit word. The first 16 characters give the name of the font, while the last 16 describe special characteristics, such as thickness and style. This text string is one of the values returned by vqt_name( ).
36–37	First character. The ASCII value of the first character in the font. This value is returned by vqt_font_info( ).
38–39	Last character. The ASCII value of the last character in the font. This value is returned by vqt_font_info( ).
40–41	Top line distance. The distance in pixels from the baseline to the top line. This value is returned by vqt_font_info( ).
42–43	Ascent line distance. The distance in pixels from the baseline to the ascent line. This value is returned by vqt_font_info( ).
44–45	Half line distance. The distance in pixels from the baseline to the half line. This value is returned by vqt_font_info( ).
46–47	Descent line distance. The distance in pixels from the baseline to the descent line. This value is returned by vqt_font_info( ).
48–49	Bottom Distance. The distance from the baseline to the bottom line. This value is returned by vqt_font_info( ).
50–51	Character Width. The width of the widest character in the font.
52–53	Cell Width. The width of the widest character cell in the font. This value is returned by vqt_font_info( ).
54–55	Left offset. The number of pixels added to left side of character by special effects. This value is returned by vqt_font_info( ).
56–57	Right Offset. The number of pixels added to right side of character by special effects. This value is returned by vqt_font_info( ).
58–59	Thickening width. The number of pixels added to the width of a character by the thickening special effect.

60–61   Underline size. The width in pixels of the line used to underline a character.

62–63   Lightening mask. The 16-bit mask used to remove pixels from the character for the lightening special effect. The pattern 0 X 5555, which removes every other pixel, is the one most commonly used for this purpose.

64–65   Skewing mask. A 16-bit mask used to determine how to shift the character's image data for skewing (italics). The pattern 0 X 5555 is the one most commonly used for this purpose.

66–67   Font flags. Each flag occupies one bit:

Bit	Value	Description
0	1	Set to 1 if this is the default system font
1	2	Set to 1 if there's a horizontal offset table
2	4	Byte-orientation flag. Set to 1 if data is in high-byte, low-byte order used by 6800 processor
3	8	Set to 1 if this is a mono-spaced font.

68–71   Horizontal offset table pointer. The number of bytes from the beginning of the file to the horizontal offset table.

72–75   Character offset table pointer. The number of bytes from the beginning of the file to the character offset table.

76–79   Font data pointer. The number of bytes from the beginning the file to the start of font image data.

80–81   Form width. The number of bytes required to hold the combined widths of all of the characters in the font (total character widths divided by 8).

82–83   Form height. Same as the font height in pixels.

84–87   Pointer to the next font. These four bytes are placeholders for a pointer to the next font which is set by the device driver.

Character table offset. The next section is called the character offset table. This table contains the offset for the characters' image data from the beginning of the image data table. This offset is equal to the sum of the widths of all of the preceding characters. For example, let's say the first character in the font has an ASCII value of 32. Its offset is the first entry in the offset table, which always has a value of zero. If that character is 4 pixels wide, the second entry, for character 33, will be four. The width of character 33 will be added to four to obtain the value for the third entry, which covers character 34. You can find the width of any individual character by subtracting its offset from that of the following character. That
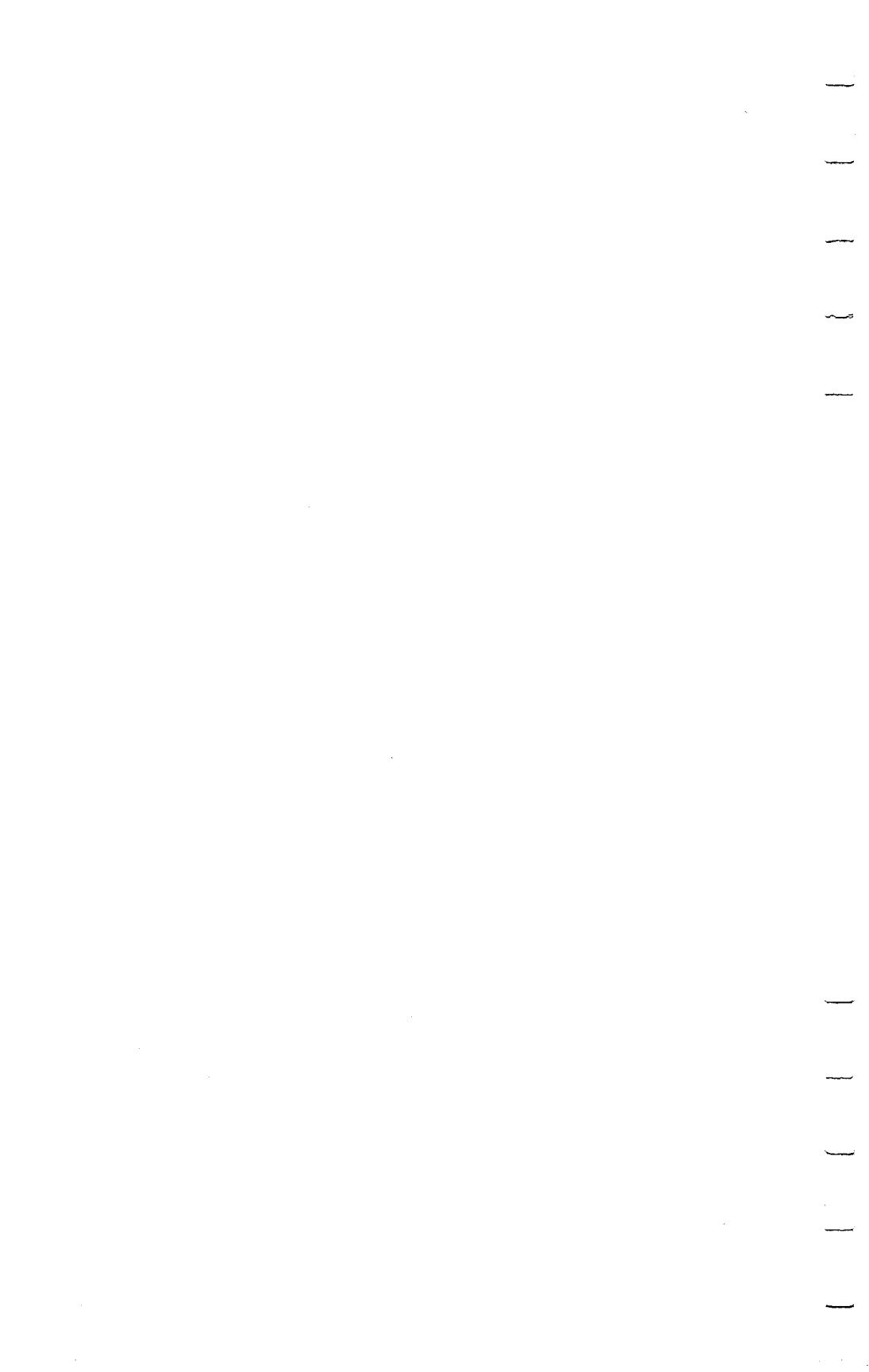
means that there will have to be one more entry in the table than there are characters, since you need to subtract the offset for the last character from that of the following one. Note that this table is necessary even for fonts whose characters all have the same widths (called mono-spaced fonts).

Horizontal offset table. The third section is an optional horizontal offset table. This table contains one entry per character, showing the additional number of pixel spaces (positive or negative) that should be added before the character is output. A bit in the flag word of the header table indicates whether or not there is a horizontal offset table.

Actual image data. The final section is the actual image data for the characters in the font. Character data is formatted with all of the data for each scan line of all of the characters following one after the other. The data for the first line of the first character is followed by the data for the first line of the second character, and so on. Each scan line starts on a word boundary, but within a scan line the characters are not byte- or word-aligned. That means that if each character is six bits wide, the first character uses the first six bits in the first byte, and the second character uses the last two bits of the first byte and the first four bits of the next byte. Only at the end of the scan line is padding added to make the next scan line start on a word boundary.

Important Note
The few disk-based fonts that were available for examination at the time of this writing were arranged in the Intel format used by the IBM version of GEM. This means that 16-bit values are formatted with the low byte first and the high byte second, and 32-bit values are stored with the least significant byte first, followed by increasingly significant bytes. For example, in the font header, if the 16-bit font ID has a value of 2, the number appears in the header with the two-byte followed by the-zero byte.

Appendix D
System Characters

This appendix includes all the system font.
The font supports all characters from 0 through 255.

0	13 ${}^{C}_{R}$	26 **ə**	39 **❚**
1 **⇧**	14 **⫫**	27 ${}^{E}_{S}$	40 **(**
2 **⇩**	15 ****	28 **ᄃ**	41 **)**
3 **⇨**	16 **◘**	29 **⅃**	42 **✳**
4 **⇦**	17 **¦**	30 **⚔**	43 **✚**
5 **▨**	18 **ᘗ**	31 **⚒**	44 **,**
6 **▰**	19 **Ʒ**	32 Space	45 **━**
7 **▨**	20 **Ч**	33 **!**	46 **.**
8 **√**	21 **Ƨ**	34 **"**	47 **/**
9 **◷**	22 **Ƃ**	35 **#**	48 **0**
10 **⚲**	23 **ᄀ**	36 **$**	49 **1**
11 **♪**	24 **8**	37 **%**	50 **2**
12 ${}^{F}_{F}$	25 **9**	38 **&**	51 **3**

52 **4**	66 **B**	80 **P**	94 **∧**
53 **5**	67 **C**	81 **Q**	95 **_**
54 **6**	68 **D**	82 **R**	96 **`**
55 **7**	69 **E**	83 **S**	97 **a**
56 **8**	70 **F**	84 **T**	98 **b**
57 **9**	71 **G**	85 **U**	99 **c**
58 **:**	72 **H**	86 **V**	100 **d**
59 **;**	73 **I**	87 **W**	101 **e**
60 **<**	74 **J**	88 **X**	102 **f**
61 **=**	75 **K**	89 **Y**	103 **g**
62 **>**	76 **L**	90 **Z**	104 **h**
63 **?**	77 **M**	91 **[**	105 **i**
64 **@**	78 **N**	92 ****	106 **j**
65 **A**	79 **O**	93 **]**	107 **k**

System Characters
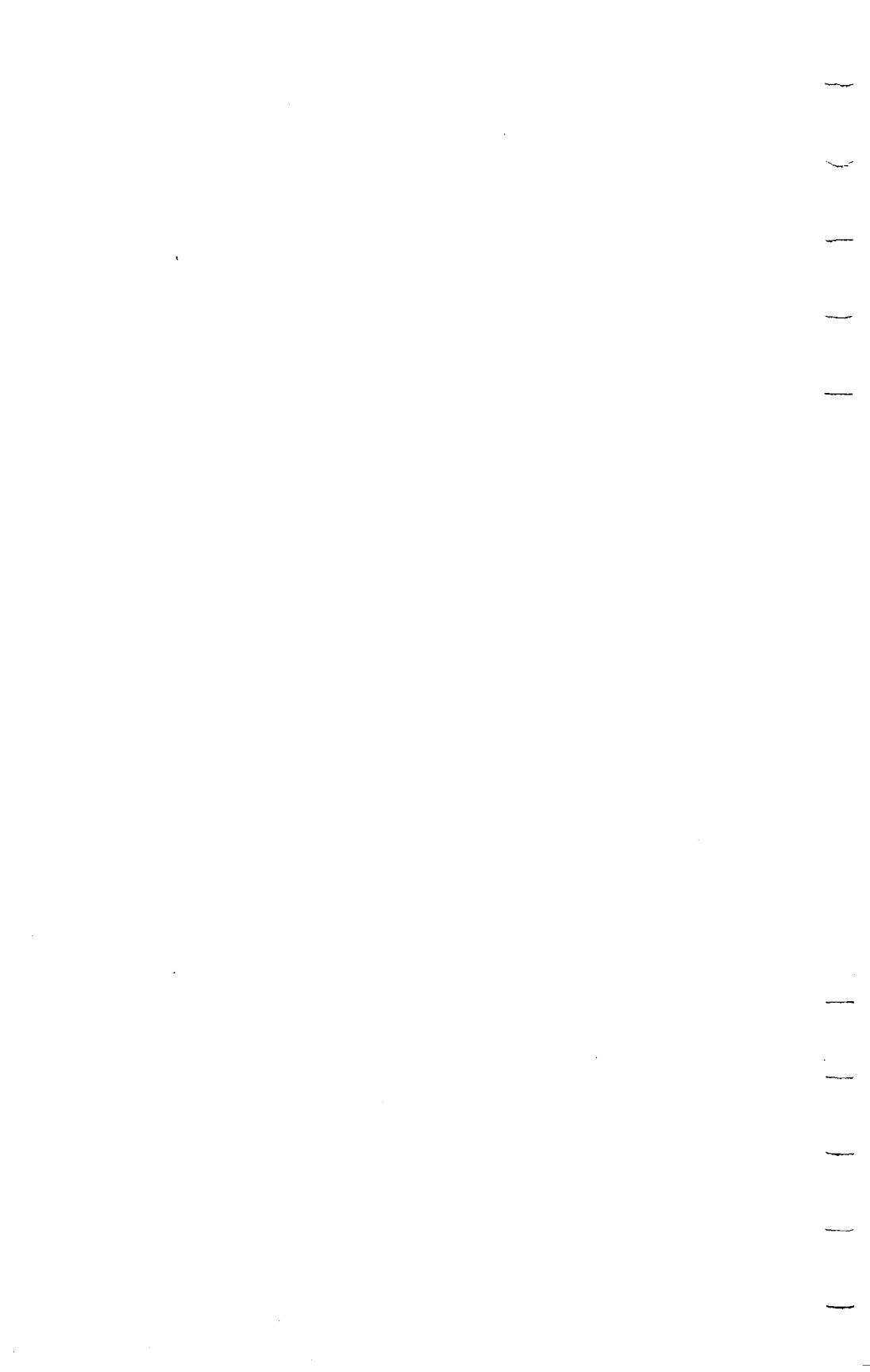
108 l	122 Z	136 ê	150 û
109 m	123 {	137 ë	151 ù
110 n	124 |	138 è	152 ÿ
111 o	125 }	139 ï	153 ö
112 p	126 ~	140 î	154 ü
113 q	127 Δ	141 ì	155 ¢
114 r	128 Ç	142 Ä	156 £
115 s	129 ü	143 Å	157 ¥
116 t	130 é	144 É	158 β
117 u	131 â	145 æ	159 ƒ
118 v	132 ä	146 Æ	160 á
119 w	133 à	147 ô	161 í
120 x	134 å	148 ö	162 ó
121 y	135 ç	149 ò	163 ú

164	ñ	178	⌐	192	ij	206	מ
165	Ñ	179	⌐	193	IJ	207	ן
166	ª	180	œ	194	א	208	ם
167	º	181	Œ	195	ב	209	ﬥ
168	¿	182	À	196	ג	210	נ
169	⌐	183	Ã	197	ד	211	ץ
170	¬	184	Õ	198	ה	212	ק
171	½	185	¨	199	ו	213	ר
172	¼	186	´	200	ז	214	ש
173	¡	187	†	201	ח	215	ת
174	«	188	¶	202	ט	216	ן
175	»	189	©	203	ˈ	217	ר
176	ã	190	®	204	כ	218	ﬥ
177	õ	191	™	205	ל	219	ף

220 ⁇	234 Ω	248 °
221 §	235 δ	249 ●
222 ∧	236 ⱷ	250 ·
223 ∞	237 φ	251 √
224 ∝	238 Є	252 ∩
225 β	239 ∏	253 ²
226 Γ	240 ≡	254 ³
227 π	241 ±	255 —
228 Σ	242 ≥	
229 σ	243 ≤	
230 μ	244 ⌠	
231 τ	245 ⌡	
232 Φ	246 ÷	
233 Θ	247 ≈	

Index by Function

Index by Opcode

340

Index

COMPUTE! Books

Ask your retailer for these **COMPUTE! Books** or order directly from **COMPUTE!**.

Call toll free (in US) **1-800-346-6767** (in NY 212-887-8525) or write COMPUTE! Books, P.O. Box 5038, F.D.R. Station, New York, NY 10150.

Quantity	Title	Price*	Total
_____	COMPUTE!'s ST Artist (070-X)	$18.95	_____
_____	COMPUTE!'s First Book of the Atari ST (020-3)	$16.95	_____
_____	COMPUTE!'s Kids and the Atari ST (038-6)	$14.95	_____
_____	COMPUTE!'s ST Applications Guide: Programming in C (078-5)	$19.95	_____
_____	COMPUTE!'s St Applications (067-X)	$16.95	_____
_____	COMPUTE!'s ST Programmer's Guide (023-8)	$17.95	_____
_____	The Elementary Atari ST (024-6)	$18.95	_____
_____	Elementary St BASIC (034-3)	$14.95	_____
_____	Introduction to Sound and Graphics on the Atari ST (035-1)	$16.95	_____
_____	Learning C: Programming Graphics on the Amiga and Atari ST (064-5)	$18.95	_____

*Add $2.00 per book for shipping and handling.
Outside US add $5.00 air mail or $2.00 surface mail.

NC residents add 5% sales tax. _____

NY residents add 8.25% sales tax _____

Shipping & handling: $2.00/book _____

Total payment _____

All orders must be prepaid (check, charge, or money order).
All payments must be in US funds.
☐ Payment enclosed.
Charge   ☐ Visa   ☐ MasterCard   ☐ American Express

Acct. No._____ Exp. Date _____
                                                        (Required)

Name _____
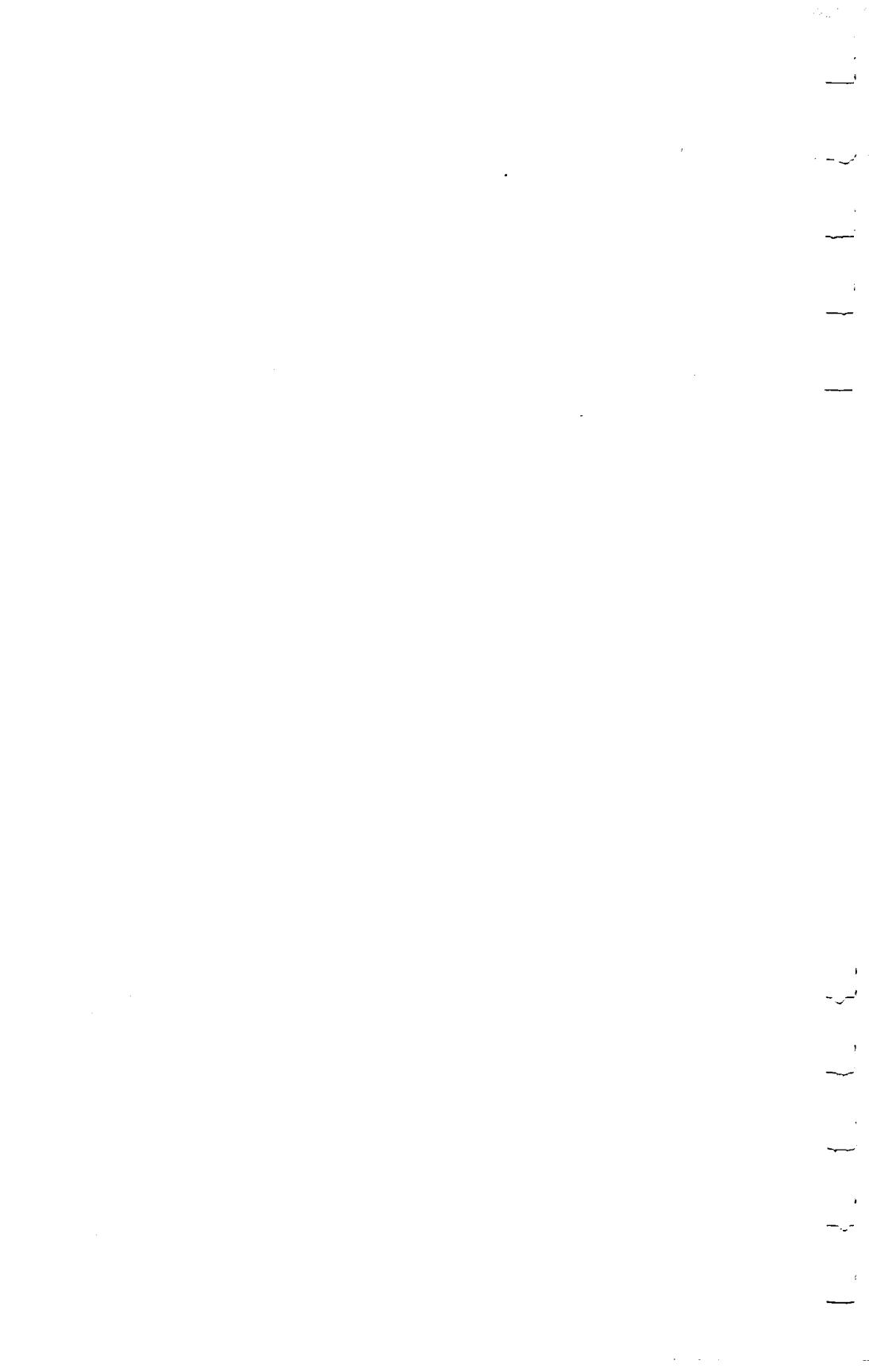
Address _____

City_____ State _____ Zip _____

*Allow 4–5 weeks for delivery.
Prices and availability subject to change.
Current catalog available upon request.
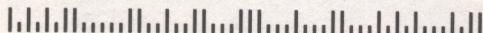
NEW FOR ATARI ST USERS

COMPUTE!'s ATARI ST
DISK & MAGAZINE

Only *COMPUTE!'s Atari ST Disk & Magazine* gives you all this and more in each big issue:

TOP QUALITY PROGRAMS: Application programs for home and business. Utilities. Games. Educational programs for the youngsters. All are already on an enclosed disk and ready to run. For example: a typical disk might contain an elaborate adventure game written in BASIC, a programming utility written in machine language, a dazzling graphics demo in compiled Pascal, and a useful home or business application written in Forth or C.

NEOCHROME OF THE MONTH: What are computer artists doing with the Atari ST? Each issue contains a Neochrome picture file—ready to load and admire.

REGULAR COLUMNS: If you're a programmer—or would like to be—you'll love our col-

umns on ST programming techniques and the C language. Or check out our column on the latest events and happenings throughout the ST community. Or send your questions and helpful hints to our Reader's Feedback column.

REVIEWS: Honest evaluations of the latest, best software and hardware for the Atari ST.

NEWS & PRODUCTS: A comprehensive listing of all the new software and peripherals for your ST.

AND MORE: Interviews with ST newsmakers, reports on the latest industry trade shows, and overviews of significant new product introductions.

Don't miss a single big issue. Subscribe to *COMPUTE!'s Atari ST Disk & Magazine* now through this special money-saving offer. Return coupon above or call **1-800-247-5470 (in Iowa 1-800-532-1272).**

COMPUTE! Publications, Inc.
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies

*RETURN COUPON ABOVE TO ENJOY
CHARTER SUBSCRIPTION PRIVILEGES*

The Complete VDI Reference

If you're going to design and write software for the Atari ST in BASIC, machine language, or C—and take advantage of all the advanced features the computer has to offer—you need *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI*.

The first in a series of three reference guides for the Atari ST personal computer, this book has everything you need to create sophisticated, professional-looking graphics. Here's just a sample of what you'll find inside:

- A complete easy-to-use VDI (Virtual Design Interface) function reference section.
- Numerous sample programs which demonstrate exactly how to implement VDI function calls from C, machine language, and BASIC.
- Drawing and manipulating image blocks.
- How to fill shapes and draw points and lines.
- How font files are organized.
- Three indices that make finding the right information easy and quick.
- A complete listing of extended keyboard codes.

Written in a clear and concise style by the noted ST author Sheldon Leemon, *COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI* is for every intermediate-to-advanced-level BASIC, C, and machine language programmer who wants to tap the true potential of this powerful computer.

COMPUTE!'s Technical Reference Guide, Atari ST Volume One: The VDI is *the* complete tutorial and reference guide to a vital part of all ST software development.

$18.95