SYSTEMS MANAGEMENT ASSOCIATES
3325 Executive Drive
P.O. Box 20025
Raleigh, North Carolina 27619
(919) 878-3600

January 30, 1986

Dear PROMAL Owner:

Here is your new PROMAL GRAPHICS TOOLBOX Version 1.0. We are sure you will enjoy using this software package to enhance your PROMAL application programs with exciting graphics. Bruce Carbrey, the author of PROMAL, personally designed this advanced bit-mapped graphics support system. He and his team have implemented the package with the same standards of excellence found in PROMAL. I know you will be pleased with their results.

You should review the GRAPHICS TOOLBOX Manual to see if the package, as described, meets your needs and expectations. If for any reason you are not satisfied, return the unopened diskette envelope and the manual within 30 days for a refund.

Because this package is not copy-protected, once the sealed disk envelope is opened, the package cannot be returned for a refund.

If you do not have PROMAL 2.0 which is required for the GRAPHICS TOOLBOX, please call us at 919-878-3600 so we can send you an upgrade disk. The charge for upgrading to PROMAL 2.0 from any previous version is only $10.00 plus $2.50 shipping and handling.

Be sure to read the README.T file on the disk, which contains some manual corrections and information not covered in the manual. You can print this file from the PROMAL EXECUTIVE by typing:

```
TYPE README.T>P
```

Thank you for choosing the PROMAL GRAPHICS TOOLBOX.

Sincerely,

John R. Segner
President

P R O M A L

(PROgrammers's Micro Application Language)

THE PROMAL GRAPHICS TOOLBOX

For The APPLE IIe and IIc
-or-
COMMODORE 64

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, North Carolina 27609

January 1986

PROMAL GRAPHICS TOOLBOX

Apple II and Commodore 64
January, 1986

Copyright Notice

Hardware & Software Requirements

The PROMAL Graphics Toolbox runs on Apple IIe with 80 column extended card, or Apple IIc, or Commodore 64, with PROMAL 2.0 or later software. The Apple II version and Commodore version of the Graphics Toolbox provide a high degree of compatibility for the user, but differ substantially internally and are sold separately.

Trademarks

Apple II and ProDOS are trademarks of Apple Computer, Inc.
Commodore 64 is a trademark of Commodore Business Machines, Inc.
PROMAL is a trademark of Systems Management Associates, Inc.

Limited Warranty

The programs and manuals are provided "as is", without warranty of any kind, either express or implied, except that:

SMA warrants the distribution diskette to be free of physical defects in material and workmanship under normal use for a period of thirty (30) days from delivery to you as evidenced by a copy of your purchase receipt. SMA's entire liability and your exclusive remedy shall be the replacement of any diskette not meeting this warranty, by returning it postpaid to the factory.

In no event shall SMA be responsible for any indirect or consequential damages, even if SMA has been advised of the possibility of such damages. Some states do not allow the limitation or exclusion of liability for indirect or consequential damages, so the above limitation may not apply to you.

Disclaimer and Notice

The right is reserved to make any changes to this publication or the product it describes without obligation to notify any person of such revision or changes. **Because the diskette is not copy-protected, this product cannot be returned for a refund once the seal is broken.**

TABLE OF CONTENTS

APPENDICES

(This page is intentionally left blank)

INTRODUCTION

WHAT IS THE PROMAL GRAPHICS TOOLBOX?

The PROMAL GRAPHICS TOOLBOX is a software package for PROMAL programmers to use to develop high-resolution graphics application programs on the Commodore 64 or Apple IIe or IIc computers. The toolbox provides a set of subroutines which you can call from your PROMAL programs to draw points, lines, dashed lines, circles and arcs, rectangles, bars, etc. In addition you can draw text (characters) to annotate your graphics image, with text displayed horizontally or vertically in a variety of sizes. You can fill arbitrary enclosed shapes with a pattern, such as cross-hatching. Images or portions of images can be extracted from the screen and moved to other areas of memory, so that programs can save, restore, or print images. Figure 1 is a sample of the kind of graphics image that can be produced using the PROMAL GRAPHICS TOOLBOX. The complete source program used to produce this image is file SALESDEMO.S on the distribution diskette.

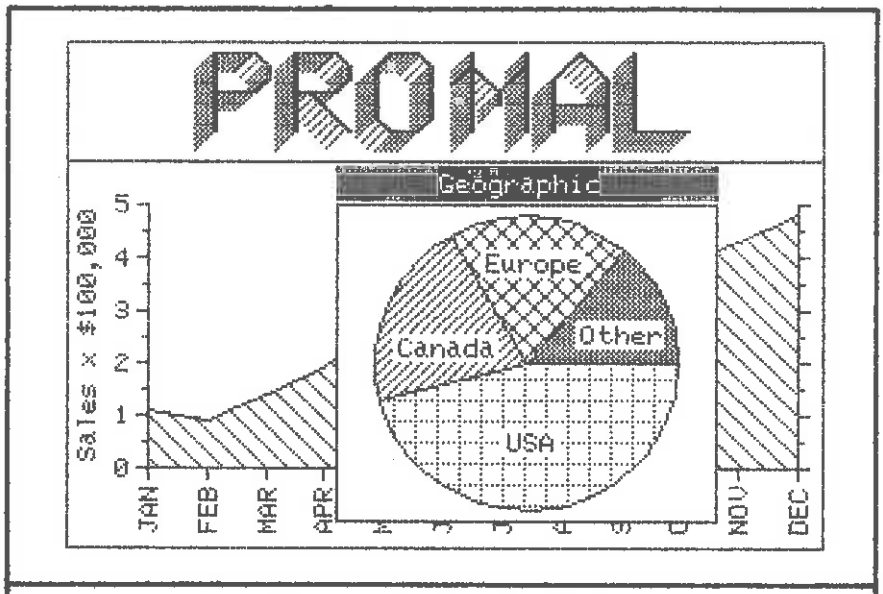


Figure 1: Sample Plot

WHAT CAN YOU DO WITH THE PROMAL GRAPHICS TOOLBOX?

The toolbox provides the primitive functions you need for almost any kind of graphics display. You can write programs to display:

* Bar charts
* Pie charts
* Point plots, scatter plots
* Curves and curve fitting
* Engineering drawings
* Games and animation
* Maps
* Scientific and statistical functions
* Geometric shapes
* Musical scores
* Interactive drawing packages
* More - use your imagination!

COMPATIBILITY AND GRAPHICS MODES SUPPORTED

Despite the fact that the underlying graphics hardware on the Apple and Commodore 64 differs greatly, the PROMAL GRAPHICS PACKAGE provides a very high degree of compatibility for source programs. Therefore you will usually not need to make any significant changes to your graphics application program when "porting" your program between these machines.

The PROMAL GRAPHICS TOOLBOX for the Commodore 64 supports 16 colors in high resolution, 320 by 200 mode. The Apple II version ignores color information because the Apple II does not have a high-resolution graphics mode. Instead, the Apple version supports the standard high-resolution monochrome 280 by 192 mode. Apple color mode is not supported because the resolution is too limited for the kinds of applications the PROMAL GRAPHIC PACKAGE is intended for.

INSTALLATION

No special installation is needed for the PROMAL GRAPHICS TOOLBOX. For safety, we suggest that you promptly make a backup copy of the distribution disk and keep the original in a safe place. Simply copy the files you need on to any working disks you wish when you develop your application program. If a file called README.T is present on your disk, please TYPE it. It will contain additional information not included in this manual.

DEMONSTRATION PROGRAMS

Before studying the PROMAL GRAPHICS PACKAGE in detail, you may wish to run a couple of the demo programs provided on the distribution disk, to get an idea of what kind of results you can expect from your programs. Boot up PROMAL 2.0 (or later) in the usual way using one of your working disks. Then put the Toolbox disk (or better, a copy of the Toolbox disk) in the drive and type:

Apple II	Commodore 64
PREFIX *	UNLOAD
BUFFERS HIRES	BOOTW SALESDEMO
BOOTW SALESDEMO	

This will display an image similar to Figure 1. When the display is completed, press RETURN to exit back to the EXECUTIVE. The distribution disk contains the complete source code for this program, so you may study it later if you wish.

An interesting interactive drawing program is SKETCH, which lets you draw lines, rectangles, circles, text, and fill enclosed areas with patterns. To run SKETCH, type:

Apple II	Commodore 64
BOOTW SKETCH	UNLOAD SALESDEMO
	BOOTW SKETCH

You will see a rectangular piece of "paper" for you to draw on in the center of the screen, with a blinking cross in the center. This is your "pen" position. You can move the "pen" by using the i, j, k, and m keys on the keyboard, as shown on the screen. Notice that these four keys form a "cross" on the keyboard indicating which direction the "pen" will move. Holding down the SHIFT key while pressing these movement keys will make the "pen" move faster. Pressing the RETURN key will "anchor" the pen at the current point. Using the movement keys will then draw a "rubber band line" which you can move around until you get it where you want it. Then press RETURN again to "freeze" the line on the screen. You may draw any number of lines in the same manner. Note that if you place the "rubber band" line over an existing line, the existing line will be temporarily "flipped" to make it invisible. This is normal, and the line will re-appear when you press RETURN or move the rubber band line elsewhere. You can move your anchor point by pressing the space bar.

You can use the function keys (Apple key with a number key on the Apple) to select other shapes, as indicated. Once you have selected "text", you can move to where you want the first letter and then press RETURN. Then type in normally. The backspace (or delete) key can be used to "undo" mistakes.

By pressing a number key, you can choose one of the pre-defined patterns to fill an area with. Move the cursor to the inside of an enclosed shape and press RETURN to fill the area. Caution: If you fill a shape that is not completely enclosed, the fill will "spill out" and fill up the whole screen. Also be careful not to try to fill an area that has already been filled.

The legends on screen indicate a key that can be used to "UNDO". This will restore the screen to the way it was before the most recent command (a command is defined as all the drawing done since the last function key was pressed).

To exit the program, press Q (for quit). The complete source program for the SKETCH demo is provided on the distribution diskette. You may wish to study or modify it after you have read this manual.

SYSTEM COMPONENTS

The PROMAL GRAPHICS PACKAGE is supplied in two main parts: The Screen Graphics Drivers (**SGD**) and the Window Graphics System (**WGS**). The SGD is a collection of very-high-performance, machine language subroutines, totaling about 4.5K bytes in size, callable from your PROMAL application program. The SGD provides the graphic "primitives" for drawing points, lines, etc. on your screen. Coordinates are given in terms of the X and Y locations of the pixels (dots) on your screen. The SGD subroutines are fast enough to perform some animated effects in high resolution mode, a considerable achievement for hardware in the Commodore/Apple class.

The WGS provides a group of higher-level subroutines written in PROMAL, provided both in source and pre-compiled form. These subroutines can be called from your PROMAL application for more sophisticated functions. In particular, the WGS provides support for multiple graphics windows on the screen. With windows, information to be plotted can be described in an arbitrary (X,Y) coordinate system, with automatic scaling to fit in a specified rectangular "viewport" on the screen. Lines are automatically "clipped" at the boundaries of the window, making it easy to write applications which need to "zoom" in to show part of a larger image.

Please note that the term "windows" as used here refers to traditional, display graphics oriented windows. The PROMAL GRAPHICS TOOLBOX does not directly support "pull down windows" for text, nor does it directly support a mouse or other graphics input devices. Also it does not directly support sprites on the Commodore 64, although you can program sprites directly using PROMAL in high resolution mode in conjunction with the PROMAL GRAPHICS PACKAGE, and a Sprite Editor is included on the distribution disk, to make it easy to create sprites.

In addition to the SGD and WGS, a number of graphics utility subroutines are provided on disk in PROMAL source form, which you can include in your program where needed. Included are subroutines for saving and restoring images or parts of images to memory or disk files, printing images, etc. You can easily modify these subroutines to meet any special needs you have.

The following pages explain how to use the PROMAL GRAPHICS TOOLBOX in your own PROMAL application programs. This manual assumes you already have a good working knowledge of PROMAL and your computer.

GRAPHICS FUNDAMENTALS

BIT-MAPPED GRAPHICS AND COORDINATE SYSTEMS

In normal, "text" display mode, the Apple II or Commodore 64 map a portion of memory onto the screen, with each byte of display memory representing one character position on the screen. Thus 1000 bytes are used on the 40x25 Commodore Screen, and 2000 bytes on the 80x25 Apple II screen. The actual conversion from a character code in memory to a pattern of dots on the screen recognizable as a character is done by hardware.

By switching to graphics mode, the normal text display is no longer visible on the screen. Instead, A different, larger area of memory (about 8K bytes) is mapped onto the screen, with each bit of this memory area controlling a single dot on the screen. These dots are called **pixels** (short for "picture elements"). Setting a bit to 1 in this display memory causes the corresponding pixel on the screen to be shown as a foreground (bright) dot on the screen. Setting the bit to 0 causes the pixel to be turned off (to the background color). This is called **bit-mapped graphics.**

The reason a larger area of memory is needed for displaying graphics information is that a single byte of memory can only control at most 8 pixels on the screen. In text mode, a single byte controls about 48 to 64 pixels, depending on the format of displayed characters. For example, the Commodore displays a character in a 8 pixel by 8 pixel cell. The advantage of bit-mapped graphics mode is that you can display any pattern of dots on the screen, not just the characters defined by your computer's hardware.

The term **resolution** is used to describe the total number of pixels which are individually recognizable on your screen. The more pixels, the higher the resolution is. Computers with more pixels have a better quality display because detail can be more accurately represented. This is illustrated by **Figure 2**, which represents a small portion of two bit mapped displays. The one at the right has four times as many pixels (better resolution) as the other. If we try to draw a diagonal line shown, the best we can do is to turn on the pixels which best approximate the true line, as shown. Notice that the approximation on the right looks better because the larger number of pixels reduces the "stair step" effect.

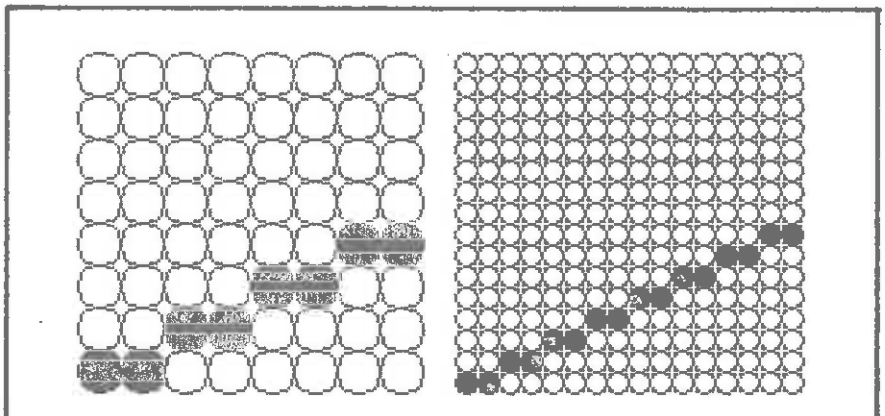


Figure 2: Diagonal Line at Different Resolutions

If you want to draw a line in graphics mode, you need to turn on the pixels which best approximate the line. Deciding which pixels these are and which bits in memory represent them is a non-trivial task, and is one of the most important functions performed by the SGD (Screen Graphics Drivers) of the PROMAL GRAPHICS TOOLBOX. In order to refer to pixels, we need a way to identify them. For the PROMAL GRAPHICS TOOLBOX, we use a simple traditional rectangular coordinate system. Dots on the screen have an X coordinate numbered from left to right and a Y coordinate measured from bottom to top. The notation **(X, Y)** is used to identify the coordinate of a particular pixel. For example, (0, 0) is always at the extreme lower left hand corner of the screen. (2, 7) is the third pixel to the right and the eighth pixel up.

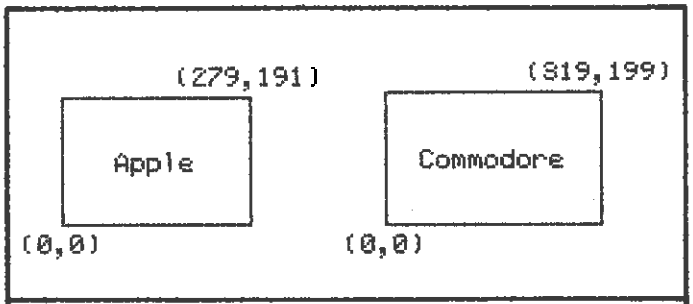


Figure 3: Screen Coordinate Systems

For the Apple, the upper right hand corner of the screen is at (279,191). For the Commodore 64, it is at (319,199), as shown in **Figure 3.**

Please note that for the **SGD** subroutines, **coordinates are expected to be INTegers** (although BYTE or WORD coordinates are also acceptable). You may **not** specify coordinates for the SGD subroutines as **REAL!** If you do, it will not be detected as a compilation error, but will produce very strange results.

THE GRAPHICS CURSOR IS YOUR "PEN"

The SGD works in terms of an imaginary "pen" called the **graphic cursor** which is always somewhere on the screen. Unlike the text cursor, the graphics cursor is invisible. To draw a line, you **move** the graphic cursor to the desired starting point and then **draw** from there to the desired ending point. For example:

```
S_MOVE 50,30
S_DRAW 150,30
```

This draws a horizontal line on the screen from (50,30) to (150,30). (Note: you would actually have to do some initialization first to enable graphics mode, which we have not discussed yet). S_MOVE and S_DRAW are procedures in the SGD. All procedure and function names in the SGD start with "S_" for easy recognition. After the line is drawn, the "pen" remains where it finished (at (150,30). You could complete a 100 by 60 pixel rectangle by adding:

```
S_DRAW 150,90
S_DRAW 50,90
S_DRAW 50,30
```

The globally predefined variables **S_X** and **S_Y** contain the current value of the graphics cursor. The SGD subroutines automatically update the values of S_X and S_Y. You can directly set the values yourself too. For example:

```
S_X = 125
S_Y = 44
```

is equivalent to:

```
S_MOVE 125,44
```

The cursor is very important. The cursor represents the starting point for almost all SGD operations.

USING THE SGD IN YOUR PROGRAMS

As was mentioned in the introduction, the SGD (Screen Graphics Drivers) is a relocatable machine language module containing subroutines for performing the fundamental graphics operations. The subroutines in the SGD are summarized in Table 1. You can call these subroutines directly from your PROMAL application program. In addition, the WGS subroutines and other utility subroutines call the SGD. The SGD subroutines will be explained in detail in the following section. The WGS and Utility subroutines will be described later.

The skeleton program below shows how you need to set up your program to use the SGD subroutines:

```
PROGRAM HASGRAPH OWN
...
INCLUDE LIBRARY
INCLUDE SGD.E
...
BEGIN
...
S_INIT
...
;...Call Graphics Routines as Desired Here...
...
S_END
...
END
```

Table 1
Summary of SGD Subroutines

Name	Description
S_INIT	Initialize graphic system, clear screen, enable graphics mode.
S_CLEAR	Clear the screen and move the graphic cursor to (0,0).
S_DOT	Draw (or erase or flip) a single pixel.
S_MOVE	Move the graphic cursor to a new coordinate.
S_DRAW	Draw a line.
S_PLOT	Draw, erase, or flip a line or dashed line.
S_RELPLOT	Draw, erase, or flip a line or dashed line using relative coordinates.
S_TEXT	Draw text characters.
S_RECT	Draw (or erase or flip) a rectangle.
S_BAR	Draw, erase, or flip a filled rectangular area.
S_END	Exit from graphics mode, restore normal text screen.
S_GETDOTS	Move a row of dots from the screen to an array.
S_PUTDOTS	Move an array to a row of dots on the screen.
S_XYADDR	Compute the address of a specified pixel.
S_FILL	Fill an enclosed shape with a pattern.
S_GRAPHON	Re-enable graphic mode.
S_ARC	Draw (or erase or flip) a circle or octant of arc.
S_SHAPE	Draw (or erase or flip) a figure from a shape table.

In order to use any graphics, you must have the INCLUDE SGD.E declaration. This makes all the definitions of the SGD subroutines (and variables) known to the compiler.

Next, before using any other graphics subroutines, you **must** call S_INIT. This subroutine initializes the graphics system, sets the graphic cursor to (0,0), clears the graphics screen and enables graphics mode. You may then call any graphics subroutines, such as S_MOVE and S_DRAW, to display your desired graphics image. These subroutines are described in detail in the following sections. Finally, you should call S_END to disable graphics mode and return to text mode.

MEMORY MAP AND LOADING CONSIDERATIONS

Graphics programs require some special considerations. First, you need to insure that the 8K byte high resolution display memory is properly allocated.

On the **Apple II**, there are two possible hi-resolution screen memory locations: $2000 and $4000. Normally, the SGD will use the display memory at $2000. Since the PROMAL runtime package and ProDOS buffers normally extend all the way up to $2900, you need to issue a **BUFFERS HIRES** command from the EXECUTIVE before running your graphics program. This alters the PROMAL memory map, reserving 8K bytes at $2000 for display memory.

NOTE: On the Apple II, after a BUFFERS HIRES command, you will not be able

to run the compiler again until you reclaim the 8K display memory by typing a **BUFFERS 3** command from the EXECUTIVE.

On the **Commodore 64**, the SGD will put the 8K byte display memory at $A000, and will put the 1K "color matrix" at $8C00. This arrangement leaves a fairly large area ($5100-8BFF) available for your program. However, it also means that the display memory will overlay the EDITor. Therefore if you EDIT after running a graphics program, the EDITor will be reloaded from disk. If you've been following this closely, you may wonder why there is an unused 4K "hole" in from $9000-9FFF. This space cannot be used for display memory (or for the color matrix) because the VIC video controller "sees" the ROM character fonts at this location and cannot use this memory. However, **you** can freely use this 4K hole for your own purposes. As you will see later, it may be put to good use as buffers for saving parts of images, alternate character fonts, as a fill buffer for cross-hatching, etc. No special commands are needed on the Commodore to prepare for a graphics program. Also, you should have OWN on the program line (otherwise, your variables may wind up allocated in screen memory on the Commodore 64!).

On either computer, you will need to load the SGD into memory **before** executing your program. You could write a "bootstrap" program to load the SGD and your program, as described in the LOADer section of the PROMAL manual. For convenience, two bootstrap programs are included on the TOOLBOX disk. BOOTS will load the SGD and then run any program given as its argument. BOOTW will load the SGD, WGS, and the named program. For example:

```
UNLOAD
BOOTS MYGRAPH
```

will load the SGD and execute the program MYGRAPH.C.

DESCRIPTION OF SGD VARIABLES AND SUBROUTINES

SGD GLOBAL VARIABLES

The SGD defines certain global variables which control the way the graphics operations are performed, or reflect the current state of graphic operations. These variables are automatically defined when you have the statement **INCLUDE SGD.E** in your programming. The most important of these variables are S_X and S_Y, which are the coordinates of the graphics cursor. The SGD global variables are summarized in Table 2. The purpose and use of these variables will be described where appropriate in the following sections.

COORDINATE RANGE CHECKING

The S_INIT subroutine sets the global variables S_XMAX and S_YMAX to the highest pixel coordinate allowed by the high resolution graphics mode and sets the global range checking flag, S_RANGECHK, to TRUE. Thereafter, any SGD subroutine will validate the coordinates before performing the requested graphic function. This is important, because if you accidentally specify a coordinate which is off-screen with range checking disabled, other areas of memory besides display memory will be affected, possibly crashing your program or the system. With range checking enabled,the SGD does not perform true "clipping" of coordinates, but simply forces the coordinates into range by the following simple algorithm:

```
IF S_X < 0
  S_X=0
IF S_X > S_XMAX
  S_X = S_XMAX
IF S_Y < 0
  S_Y = 0
IF S_Y > S_YMAX
  S_Y = S_YMAX
```

You can disable range checking by setting S_RANGECHK to FALSE, which will give a slight increase in drawing speed, especially for points and arcs. However, this is not recommended unless absolutely necessary.

If your graphics program behaves unexpectedly and draws a lot at the extreme ends of the screen, it may mean you have specified coordinates out of range, which are being forced to the limits of the display.

Table 2
SGD Global Variables

Variable Name	Type	Description
S_X	INT	Current X pixel coordinate of cursor, 0 to S_XMAX.
S_Y	INT	Current Y pixel coordinate of cursor, 0 to S_YMAX.
S_XMAX	INT	Maximum allowable X pixel coordinate. Set by S_INIT to 279 for Apple or 319 for Commodore 64.
S_YMAX	INT	Maximum allowable Y pixel coordinate. Set by S_INIT to 191 for Apple or 199 for Commodore.
S_DASHPIC	WORD	Current dash pattern template used for drawing dashed lines. Each 1 bit corresponds to a pixel which should be "on"; each 0 bit a pixel "off".
S_MODE	BYTE	Code for current default "pen" mode. 0=move, 1=draw, 2=erase, 3=flip (reverse current state), 4=dashed line using S_DASHPIC (where applicable).
S_RANGECHK	BYTE	Flag, initialized to TRUE by S_INIT. If set to 0 disables range checking for coordinates (not recommended except where essential for speed).
S_HRPAGE	BYTE	High order 8 bits of the address of the base of the display memory. Initialized by S_INIT to $20 for the Apple and $A0 for the Commodore 64.
S_FONT	WORD	Pointer to the current character font table used by S_TEXT. Initialized by S_INIT to point to the default ASCII 5x7 font.
S_FORECOLOR	BYTE	Current foreground (draw) color. Ignored on Apple, Initialized by S_INIT to the current text color on the Commodore 64.
S_BACKCOLOR	BYTE	Current background (erase) color. Ignored on Apple, Initialized by S-INIT to the hardware background color on the Commodore 64.
S_MASK	BYTE	Bit mask returned by procedure S_XYADDR to extract a pixel from screen memory.

Notes for Table 2:

1. The variables S_XMAX and S_YMAX should be considered "read only" (that is, you should not alter the values). The value of S_HRPAGE can, with care, be changed to cause drawing on a non-visible 8K chunk of display memory.

2. The SGD uses memory locations $0334 through $039F inclusive on both the Apple II and Commodore 64. You should not use this area for any other purpose if you are going to use the PROMAL GRAPHICS TOOLBOX.

TEXT OUTPUT WHILE IN GRAPHICS MODE

It is important to understand that while in graphics mode, no ordinary output (using PUT or OUTPUT, for example) will appear on the screen. Instead, you may use the S_TEXT procedure to display strings of characters on the screen. However, you **may** continue to use normal PUT and OUTPUT statements while in graphics mode; when you execute an S_END or exit or abort the program, the normal screen will be re-displayed along with any accumulated output. This is often useful for debugging purposes. You may also switch back and forth between text and graphics modes without clearing the graphics screen by calling S_END and S_GRAPHON.

In particular you should remember that the GETC function always echoes characters to the text screen. Therefore if you use GETC to get keystrokes while in your graphics program, you will see the result on the screen when you exit graphics mode. If this is undesirable, you may want to use function TESTKEY instead, which does not echo keystrokes. For example:

```
...
BYTE KEY
...
BEGIN
...
WHILE TESTKEY(#KEY) = 0  ; Wait for keystroke, save it in KEY.
  NOTHING
...
```

See the description of S_TEXT for more information about displaying text in graphics mode.

PIXEL PLOTTING MODES

Most SGD subroutines which draw on the screen have an optional **Mode** byte, which is used to select how the pixels involved should be affected. Legal values for Mode are described in **Table 3.**

COLOR SUPPORT (COMMODORE 64 ONLY)

The PROMAL GRAPHICS TOOLBOX supports 16 color graphics on the Commodore 64. The present Apple implementation does not support color because the Apple does not have adequate resolution in the color mode. The color variables may be set on the Apple version, but will be ignored. The rest of this section pertains to the Commodore.

When drawing on the screen, the color for pixels turned on (1) will be the color set in S_FORECOLOR, and the pixels turned off (0) will be the color set in S_BACKCOLOR. You may freely alter the values of these variables to change the color of your "pen" and "paper". The allowable values are given in **Table 4.**

Table 3
Values for MODE

Mode	Meaning
0	Move. No change in the display; however the graphics cursor location is updated.
1	Draw. Normal plotting mode. The affected pixels will be set to 1 in display memory, causing them to appear as bright spots on the screen. It is okay to draw a pixel that is already set to 1.
2	Erase. The affected pixels will be set to 0 in display memory, causing them to appear as dark (background) spots on the screen. It is okay to erase a pixel that is already set to 0.
3	Flip. The affected pixels will be reversed. Pixels which are presently 1 will be set to 0, and pixels which are 0 will be set to 1. Flip mode has the important attribute that performing a flip of any group of pixels **twice** will return them exactly to their original state. Thus it is possible to draw a line in flip mode across some existing graphics, and draw it again in flip mode to remove the line, regardless of what the line crosses. See the description of S_PLOT for more information.
4	Dashed line. This mode is not available for plotting points, circular arcs, or characters. In dashed mode, some of the affected pixels are drawn and some are erased, corresponding to bits in the global variable S_DASHPIC. S_DASHPIC defines a 16 dot pattern, which will be copied and repeated as the line is drawn. See S_PLOT for more information about dash mode.

Table 4
Commodore 64 Colors

0 = Black	8 = Orange
1 = White	9 = Brown
2 = Red	10 = Light Red
3 = Cyan	11 = Gray 1
4 = Purple	12 = Gray 2
5 = Green	13 = Light Green
6 = Blue	14 = Light Blue
7 = Yellow	15 = Gray 3

The S_INIT procedure initializes S_FORECOLOR to the current text color, and S_BACKCOLOR to the text screen background color. The S_CLEAR procedure clears the screen by setting the entire background to S_BACKCOLOR.

It is important to understand that the Commodore 64 hardware cannot control the color of individual pixels on the screen. Instead, the screen is divided up into 1000 square regions of 8 pixels by 8 pixels each. The color of each region is controlled by a byte in a 1000 byte array called the Color Matrix, at address $8C00. Each block of 64 pixels can have any foreground color and background color, but **all** the pixels in the block have the same color choices. With the PROMAL GRAPHIC TOOLBOX, anytime you draw any pixel in one of these blocks, it automatically changes the foreground and background color of all the pixels in the block to the current colors. Therefore it is generally best to use different colors only in different areas of the screen. Otherwise, you may get some unexpected results. For example, suppose you draw two crossed lines, one blue and one yellow, like this:

```
S_BACKCOLOR=0          ; Black
S_FORECOLOR=6           ; Blue
S_MOVE 0,164
S_DRAW 31,164          ; Horizontal line
S_FORECOLOR=7          ; Yellow
S_MOVE 18,152
S_DRAW 18,174          ; Vertical line
```

The result will be a blue horizontal line with a short yellow section at the crossing point, and a completely yellow vertical line. A magnified view of the pixels at the cross point is shown in **Figure 4**, showing the effect of the 8 by 8 color grid. This limitation of color has nothing to due with PROMAL or the GRAPHICS TOOLBOX; it is a hardware limitation of the Commodore 64.

Also, only certain color combinations will provide adequate contrast. Please see page 152 of the Commodore 64 Programmer's Reference Guide for more information. A black background will give you the widest choice of colors.

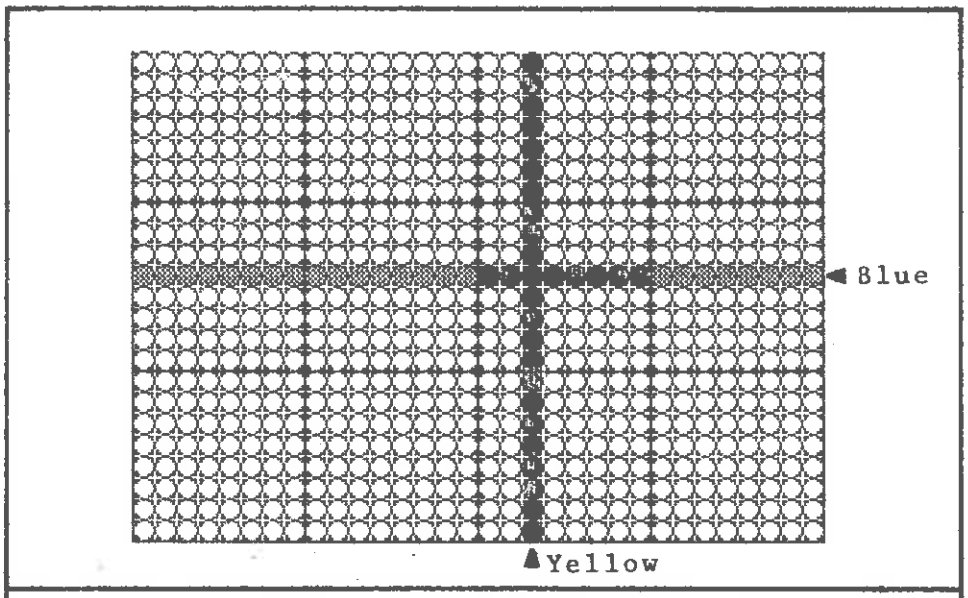


Figure 4: Commodore 8x8 Color Grid Effect

DETAILED DESCRIPTION OF SGD SUBROUTINES

The following section describes the individual SGD procedures and functions in detail, with simple examples. The notation follows the same conventions as the PROMAL Language Manual; in particular, arguments shown in square brackets are optional.

PROC S_INIT INITIALIZE SCREEN GRAPHICS

USAGE:

```
S_INIT [Deferred]
```

DESCRIPTION:

S_INIT initializes the PROMAL Screen Graphics Drivers, sets the SGD global variables to their default values, and normally clears the screen and enables high-resolution graphics mode. However, if the optional BYTE flag variable **Deferred** is specified as TRUE, then S_INIT will not clear the screen or enable graphics mode. Normally, the Deferred argument is omitted.

S_INIT must be called before using any other graphics subroutine. Failure to do so will result in strange behavior and may cause a system crash.

EXAMPLE:

```
PROGRAM GRAPHIT OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BEGIN
S_INIT
...
S_END
...
END
```

NOTES:

1. Setting the Deferred flag is useful when you wish to prepare a graphics image before displaying it. In this case, use **S_INIT TRUE**, and then call S_CLEAR and draw your graphics in the usual manner. When you are ready to display your image, call S_GRAPHON.

2. For the **Apple II**, you can use the Deferred form to select the alternate display page ($4000) and/or to select mixed text and graphics mode (in mixed text and graphics mode, the bottom 4 lines of the text screen replace the bottom of the high-resolution screen image). These cases are outlines below:

Selecting the $4000 page

```
PROGRAM ALTPAGE
INCLUDE LIBRARY
INCLUDE SGD.E
...
EXT BYTE SW_PAGE2 AT $C055
...
BEGIN
...
S_INIT TRUE
S_HRPAGE=$40
SW_PAGE2=TRUE
S_CLEAR
S_GRAPHON
...
S_END
...
END
```

Selecting mixed text and graphics

```
PROGRAM MIXEDTEXT
INCLUDE LIBRARY
INCLUDE SGD.E
...
EXT BYTE SW_MIXED AT $C053
...
BEGIN
...
S_INIT TRUE
SW_MIXED=TRUE
S_CLEAR
S_GRAPHON
...
S_END
...
END
```

One advantage to using the Apple display memory at $4000 instead of $2000 is that you don't have to perform a BUFFERS HIRES command. However, you will have to write your own bootstrap loader for your graphics application, taking care that you only load programs between $2900 and $3FFF and above $6000, not between $4000 and $5FFF.

PROC S_GRAPHON ENABLE GRAPHICS DISPLAY MODE

USAGE:

```
S_GRAPHON
```

DESCRIPTION:

S_GRAPHON enables the graphics display hardware and disables ordinary text display mode. It is normally used after a deferred S_INIT, when it is desired to complete the drawing of an image before displaying it. S_GRAPHON is also used to re-enable a graphics display without clearing the screen after calling S_END.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BEGIN
S_INIT
...
S_END    ; Back to text mode
...
S_GRAPHON  ; Re-enable graphics mode
...
S_END
END
```

NOTES:

1. See the notes for S_INIT for additional usage examples.

PROC S_CLEAR CLEAR HI-RESOLUTION SCREEN

USAGE:

```
S_CLEAR
```

DESCRIPTION:

Procedure S_CLEAR clears the entire high-resolution graphics display by writing zeros into all of graphic memory, starting at the location specified by S_HRPAGE (which is initialized by S_INIT). It is not necessary for graphics mode to be enabled to clear the screen. However, S_INIT must be called prior to this (or any other) subroutine. S_INIT calls S_CLEAR internally unless the DEFERRED flag is specified as TRUE.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BEGIN
S_INIT
...
S_DRAW X, Y
...
S_CLEAR   ; Clear the screen
...
S_END
END
```

NOTES:

1. To only clear a portion of the screen, use S_BAR with mode=erase (see S_BAR for details).

PROC S_END DISABLE HIGH-RES GRAPHICS MODE

USAGE:

S_END

DESCRIPTION:

Procedure S_END disables high-resolution graphics mode on your computer and re-enables the normal text display. When the text display is re-enabled, any output which was generated while graphics mode was enabled (or before) will be made visible. S_END does not erase graphics memory or otherwise alter the global graphic variables, so you may re-display graphics information again by simply calling S_GRAPHON. You may also continue to use graphics calls after calling S_END; the graphics will be performed, but of course will not be visible until you call S_GRAPHON.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BEGIN
S_INIT
...
S_END
PUT NL,"We're back in normal text mode now."
...
END
```

NOTES:

1. Although the EXECUTIVE will automatically disable graphics mode when your program exits to it, you should call S_END before your program exits, so that if you decide to run your graphics program from another program using the LOADer, it will not leave your program "hung up" in graphics mode.

PROC S_DOT PLOT A SINGLE DOT

USAGE:

```
S_DOT X, Y [,Mode]
```

DESCRIPTION:

S_DOT plots a single dot (one pixel) on the high resolution screen, at screen coordinates **(X, Y)**. **X** and **Y** are INTeger values (but BYTE or WORD expressions may also be used). **Mode** is an optional BYTE argument, defaulting to the current setting of S_MODE. Allowable values for **Mode** are 0 (ignore), 1 (draw), 2 (erase) and 3 (flip). Since S_INIT sets the value of S_MODE to 1 (draw), the default mode will normally be draw unless you have altered the global variable S_MODE. After the call to S_DOT, the graphic cursor will be set to **(X, Y)**.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
INT XLOC
INT YLOC
...
BEGIN
S_INIT
S_POINT 20,40    , Draw dot at X=20, Y=40
S_DOT S_XMAX-10, S_YMAX-10  ; Draw dot 10 from upper R.H. corner
S_DOT S_X, S_Y, 2  ; Erase the dot we just drew
XLOC = 0
YLOC = S_YMAX
S_DOT XLOC, YLOC, 3  ; Flip a dot at upper left hand corner
...
S_END
END
```

PROC S_PLOT PLOT A LINE

USAGE:

S_PLOT X, Y [,Mode]

DESCRIPTION:

S_PLOT is the fundamental line drawing subroutine for the PROMAL GRAPHICS TOOLBOX. It plots a line from the present graphic cursor location, (S_X, S_Y), to (**X, Y**). **X** and **Y** are INTegers (BYTE and WORD expressions are also acceptable). After it is finished the graphics cursor will be updated to (**X, Y**). **Mode** is an optional BYTE argument defaulting to the current value of S_MODE. Since S_MODE is initialized to 1 by S_INIT, the normal default mode will be draw mode. Legal values of **Mode** are 0 (move), 1 (draw), 2 (erase), 3 (flip) and 4 (dashed). If dashed mode is selected, the current value of S_DASHPIC will be used for the pattern (described below).

Notice that because S_X and S_Y are set to the value of X and Y before S_PLOT returns, it is easy to draw a series of connected line by only specifying the new endpoint coordinates rather than both endpoints.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
INT XX
INT YY
...
BEGIN
S_INIT
S_X=20     ; Move cursor to (20,30)...
S_Y=30
S_PLOT 50,30  ; Draw triangle...
S_PLOT 40,45
S_PLOT 20,30
IF TOUPPER(GETC)='E' ; E Key pressed?
  S_PLOT 50,30,1  ; Erase the triangle
  S_PLOT 40,45,1
  S_PLOT 20,30,1
S_MOVE 100,100
S_DASHPIC = $AAAA  ; Alternating on-off dot pattern
S_PLOT 200,100,4   ; Draw a horizontal dashed line
...
S_END
END
```

NOTES:

1. When plotting lines, the dot at the graphic cursor is always plotted, and so is the dot at the specified endpoint X,Y. Therefore:

```
S_MOVE  0,0
S_PLOT  2,0
```

will turn on a total of 3 pixels (at X=0, X=1, and X=2), not just two pixels. This is particularly important when using flip mode. For example, if you plot a rectangle in flip mode on a clear screen, the dots at the corners will not be visible when the figure is finished, because each corner pixel gets flipped on when it is the endpoint and then gets flipped back off when it is the starting point of the next side. If you plot a single point, however, using:

```
S_PLOT S_X, S_Y, 3
```

the dot will be visible because it will be flipped only once.

2. For **Mode**=4, a dashed line will be drawn using the 16 bits of S_DASHPIC as the template. Dashed lines are drawn by using the most significant bit (bit 15) of S_DASHPIC for the first pixel drawn, the next most significant bit for the next pixel, etc. After 16 pixels, the pattern repeats. Sample S_DASHPIC values and the resulting lines are shown in **Figure 5**. Dashed lines cannot be drawn in flip mode. The default S_DASHPIC initialized by S_INIT is $F0F0, which produces alternating segments of 4 dots on and 4 dots off. Plotting a dashed line with S_DASHPIC=$FFFF is equivalent to draw mode (though not as fast), and S_DASHPIC=$0000 is equivalent to an erase.

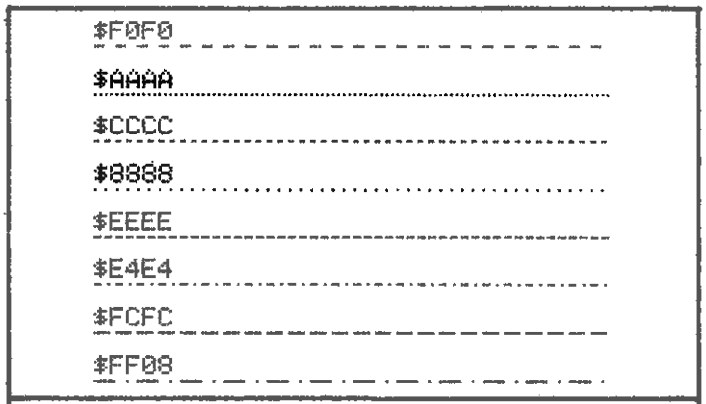


Figure 5: Sample Dashed Line Patterns

3. Flip mode (**Mode** = 3) is very useful for plotting a temporary line across an existing image, because it will show up regardless of whether the area it crosses is bright, dark, or a mix. In addition, you can remove the line and restore whatever was there previously by flipping the same line again. Flip mode is usually ideal for plotting a visible cross or other cursor which you move about the screen.

3. You may wish to try animating your graphics by drawing a line, erasing it, and then repeating it a new location. Also, you can create "rubber band" lines by leaving one end of a line "anchored", then flipping, "un-flipping" and moving the other end. When using these techniques, the line may appear to pulsate, or dots may appear to "run" back and forth, or the line may even appear to be drawn and erased very slowly under some circumstances. This is due to the "strobe" effect of the refresh rate of the screen. The screen is refreshed from top to bottom 60 times per second by hardware. Now suppose you alternately draw and erase a vertical line, which happens to take exactly 1/120th of a second to draw or erase. The result will be that any given point on the line will always be "caught" by the refresh in the same state (either off or on). Since the drawing occurs asynchronously with the screen refresh, this means that the line will not appear to flash but will appear steady. Depending on how "out of sync" it is with the refresh, the line may appear completely visible, completely invisible, or only partially drawn! If the drawing time is changed slightly from 1/120th of a second (by lengthening the line, for instance), the line may appear to be drawn and erased very slowly. Under most circumstances, strobing problems are not very noticeable. If you run into this situation with animation, you may wish to lengthen the time an image is left on the screen by adding a waiting loop for a few milliseconds before erasing.

PROC S_DRAW DRAW A LINE

USAGE:

```
S_DRAW X, Y
```

DESCRIPTION:

Procedure S_DRAW draws a solid line from the current graphic cursor position, (S_X, S_Y), to the specified coordinates (**X, Y**). **X** and **Y** should be type INTeger (but BYTE or WORD expressions are also acceptable). On exit, the graphic cursor coordinates are updated to the new endpoints.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BEGIN
S_INIT
S_MOVE 0,0
S_DRAW S_XMAX, S_YMAX  ; Draw full diagonal line
...
S_END
END
```

NOTES:

1. S_DRAW is exactly equivalent to calling S_PLOT with a MODE of 1.

PROC S_MOVE MOVE THE GRAPHIC CURSOR

USAGE:

S_MOVE X, Y

DESCRIPTION:

Procedure S_MOVE installs the value of **X** in S_X and **Y** in S_Y. **X** and **Y** are INTeger (but BYTE or WORD expressions are acceptable).

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
INT TIME
INT PRICE
...
BEGIN
S_INIT
S_MOVE  100,120      ; Move graphic cursor to X=100, Y=120
...
S_MOVE  TIME, 2*PRICE-4
...
S_END
END
```

PROC S_RELPLOT PLOT A LINE RELATIVE TO THE CURSOR

USAGE:

```
S_RELPLOT DX, DY [,Mode]
```

DESCRIPTION:

Procedure S_RELPLOT plots a line from the present graphic cursor location to a point **DX** pixels to the right and **DY** pixels up from the graphic cursor. **DX** and **DY** are signed INTegers (but BYTE or WORD expressions may be used for positive values). **Mode** is an optional argument of type BYTE specifying the desired drawing mode. If omitted, the default **Mode** will be the current value of S_MODE (which is normally 1=Draw mode unless you have changed it).

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
DATA INT SIZE = 10
...
BEGIN
S_INIT
S_MOVE 40,50
S_RELPLOT 20, 0          ; Draw 20 pixel long horizontal line
S_RELPLOT -SIZE, -SIZE ; Diagonal down & left
S_RELPLOT 0,-20, 4       ; Down 20 with dashed line
...
S_END
END
```

NOTES:

1. S_RELPLOT DX, DY, Mode is exactly equivalent to S_PLOT S_X+DX, S_Y+DY, Mode. In particular note that DX=1 means a two pixels will be drawn, not one.

PROC S_RECT PLOT A RECTANGLE

USAGE:

```
S_RECT DX, DY [,Mode]
```

DESCRIPTION:

Procedure S_RECT plots a rectangle with one corner at the current graphics cursor, (S_X, S_Y), and the opposite corner at (S_X+**DX**, S_Y+**DY**). **DX** and **DY** are INTegers (but BYTE and WORD expressions are also acceptable). **Mode** is an optional BYTE argument specifying the desired drawing mode. If omitted, the mode will be the current value of S_MODE (which is normally 1=draw unless you have changed it). Legal values for MODE are 0=ignore, 1=draw, 2=erase, 3=flip, or 4=dash. On return, the cursor position is unchanged.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD
...
PROC BOX_SCREEN
; Draw a box around the screen without affecting the cursor position
INT XSV
INT YSV
BEGIN
XSV = S_X
YSV = S_Y
S_MOVE 0,0
S_RECT S_XMAX, S_YMAX
S_MOVE XSV, YSV
END
...
BEGIN
S_INIT
BOX_SCREEN
S_END
END
```

NOTES:

1. S_RECT DX,DY MODE is equivalent to:

```
S_PLOT S_X+DX,S_Y,MODE
S_PLOT S_X,S_Y+DY,MODE
S_PLOT S_X-DX,S_Y,MODE
S_PLOT S_X,S_Y-DY,MODE
```

PROC S_BAR PLOT A SOLID RECTANGLE

USAGE:

```
S_BAR DX, DY [,Mode [,Stagger]]
```

DESCRIPTION:

Procedure S_BAR plots a solid rectangular bar by plotting a series of horizontal lines. One corner of the bar will be at the current graphic cursor coordinates, (S_X, S_Y) and the opposite corner will be at (S_X+**DX**, S_Y+**DY**). **DX** and **DY** are INTeger coordinates (although BYTE or WORD expressions may be used for positive values). **Mode** is an optional argument of type BYTE specifying the mode in which the horizontal lines of the bar should be plotted, defaulting to the current value of S_MODE (S_MODE is normally 1=draw unless you have altered it). Legal values for **Mode** are 0=ignore, 1=draw, 2=erase, 3=flip, and 4=dash. The second optional argument, **Stagger**, is used only with Mode=4 and is discussed in Note 2 below. The graphic cursor position is unchanged when S_BAR returns.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BEGIN
S_INIT
S_MOVE 110,100
S_BAR  10,5  ; Draw a solid bar X=110 to 120, Y=100  to 105
...
S_MOVE 0, S_YMAX - S_YMAX/2
S_BAR  S_XMAX,S_YMAX/2,2  ; Clear the top half of the screen
...
S_END
END
```

NOTES:

1. Using S_BAR with Mode=2 is a useful way to clear a portion of the screen. Using S_BAR with mode=3 can be used to "reverse video" any rectangular area of the screen.

2. With Mode=4 (dashed line), **Stagger** is a BYTE expression evaluating to a number between 0 and 15 specifying how many bits the dash pattern the dash pattern, S_DASHPIC, should be rotated before drawing each individual horizontal line of the bar. This is useful for creating patterns for a bar graph, etc. For example, if Stagger is specified as 1, The first line will be drawn using a copy of the S_DASHPIC pattern will be rotated 1 bit left, the second line using a copy of S_DASHPIC rotated two bits left, etc.

3. The bar plotted will encompass both the cursor coordinates and the specified coordinate. Thus if the cursor is at (0,0) and BOX 2,2 is specified, the box drawn will be 3 pixels by 3 pixels, not 2 by 2.

PROC S_TEXT PLOT TEXT

USAGE:

S_TEXT String [,Angle [,Size [,Mode]]]

DESCRIPTION:

Procedure S_TEXT is used to draw text characters in high-resolution graphics mode. **String** is a PROMAL string to be displayed. The text will be displayed starting at the current cursor position, (S_X, S_Y), using the currently - defined character font (see Note 2 below). **Angle** is an optional argument of type BYTE (or WORD or INT) giving a numeric code for the orientation of the text. The default value is 0. Legal values are 0 (horizontal), 1 (90 degrees vertical), 2 (180 degrees, upside down) or 3 (270 degrees, vertical). **Size** is an optional argument of type BYTE (or WORD or INT) specifying the size multiplier for the string. The default value is 1. Legal values may be 1 through 7. A value of 2 will cause double size characters, 3 triple size, etc. **Mode** is an optional argument of type BYTE (or WORD or INT) specifying the drawing mode for the characters. The default mode is 1=draw. Legal values are 0=ignore, 1=draw, 2=erase, or 3=flip (flip mode is not recommended).

The string may contain any ASCII characters ($20 through $7F). A single character can also be specified instead of a string. The characters will be drawn starting with the lower left hand corner of the first character at the cursor. When S_TEXT returns, the cursor will be placed after the last character.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BYTE NUMBUF [6]  ; Buffer for encoding number to ASCII
DATA WORD TITLE = "CONSOLIDATED WIDGETS"
INT MEGABUCKS
...
BEGIN
...
MEGABUCKS = 223
...
S_INIT
S_MOVE 80,10
S_TEXT "Month"  ; Label horizontally
S_MOVE 10,20
S_TEXT "Sales in $Millions", 1 ; Label vertical axis
```

```
S_MOVE 40,150
S_TEXT TITLE, 0, 2  ; Display 2X size title
S_MOVE 12,20
INTSTR (MEGABUCKS, NUMBUF)  ; Convert number to string
S_TEXT NUMBUF   ; Annotate value
...
S_END
END
```

NOTES:

1. The default character font draws characters using a 5 x 7 pixel character in a 6 x 10 character cell. On the Commodore 64, the graphics screen is wide enough for 53 characters (320/6). On the Apple II, 46 characters can be drawn on a full-width line.

2. You can define alternate character fonts, such as foreign language characters, Old English characters, proportional characters, etc. as you desire. **Appendix 1** describes how to define your own character font.

3. Characters are drawn as a series of lines. If you want to place text over the top of some part of an image (such as a cross-hatched piece of a pie chart), you may wish to use S_BAR with Mode=2 to erase an area before calling S_TEXT. Effective titles can be used by using S_TEXT with Mode=2 (erase) over a solid area created with S_BAR. Flip mode (3) is not recommended for S_TEXT because the "vertices" of the characters will not be visible (because the characters are drawn as a series of lines with common endpoints).

PROC S_SHAPE — PLOT A SHAPE FROM A TABLE

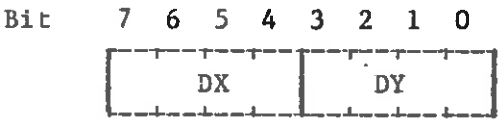
USAGE:

```
S_SHAPE Shapestring
```

DESCRIPTION:

Procedure S_SHAPE plots a shape on the screen from information compactly stored in a special string called a shape table. **Shapestring** is the address of the shape table to be used. Plotting will be begun at the current graphic cursor location, (S_X, S_Y). The shape table is a series of bytes terminated by a $00 byte.

A shape table is particularly useful for drawing pre-defined shapes quickly, especially fairly complex shapes which are not too large. Each byte in the shape table contains either a **vector byte** containing the displacement (DX and DY) for the next point to be drawn, or a special **escape byte**. Vector bytes have the following format:



DX and DY may have the values -7 to +7 only, expressed in twos complement form. Either DX or DY can be 0, but not both. DX and DY are conveniently expressed as two hex digits, using **Table 5.**

Table 5: Hex Digits for Vector Byte Encoding

Value:	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
Hex Digit:	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7

Each byte in the shape table normally causes one line to be plotted in the current mode, S_MODE, starting at the current cursor location, with dimensions specified by DX and DY of the byte. For example, a byte of $60 plots a line to the right by six pixels, and $E9 plots a diagonal line 2 to the left and 7 down.

A byte in the shape table with value **$8n** is an escape code, and causes the following vector byte to be plotted using mode n. For example, $80, $23 causes a move by 2 to the right and up three pixels. Legal values for n are 0 (move), 1 (draw), 2 (erase), 3 (flip). and 4 (dashed). Escape codes only affect the vector byte immediately following the escape code; remaining vector bytes are drawn in the normal mode.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
DATA BYTE CROSS =    ; Shape table for + shape centered on cursor
  $E0,40,$80,$EE,$04,$80,$0E, 0  ; Leaves cursor where it was
...
BEGIN
S_MOVE 40,40
S_SHAPE CROSS ; Draw 4 x 4 cross at (40,40)
...
S_END
END
```

NOTES:

1. Apple users should note that PROMAL shape table definitions differ from the shape table format used in BASIC, which can only draw vertical or horizontal lines. PROMAL shape tables are more versatile because diagonal lines can be drawn.

2. Since shape table definitions are terminated by a zero byte, you may conveniently use the PROMAL string manipulation subroutines (MOVSTR, etc.) on shape tables.

3. The graphic cursor will be left wherever you complete the shape, so it's often a good idea to end at the same place you started at. Don't forget the 0 byte at the end of the shape table!

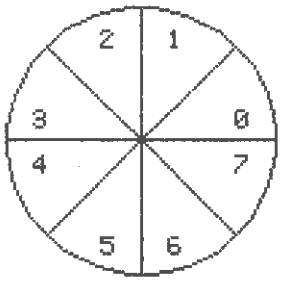
PROC S_ARC PLOT A CIRCLE OR OCTANTS OF ARC

USAGE:

S_ARC Radius, Octmask [,Mode]

DESCRIPTION:

S_ARC is one of several subroutines in the PROMAL GRAPHICS PACKAGE for drawing circles and arcs. The other subroutines are W_ARC1 and W_ARC2, described in the section on the WGS. S_ARC is not as versatile as these subroutines, but is much faster. S_ARC can draw a circle or any combination of eight octants of arc of a circle. An octant of arc is one eighth of a circle; in other words, a 45 degree section of the circumference. The circle is drawn with the center at the current graphic cursor, (S_X, S_Y). **Radius** is an INTeger expression (but BYTE or WORD expressions are acceptable) for the desired radius. **Octmask** is a BYTE expression identifying which octants of the circle are to be plotted. Each bit of **Octmask** corresponds to one octant of the circle, as shown in **Figure 6**. A value of $FF plots the entire circle. A value of $01 plots only the octant from 0 to 45 degrees. **Mode** is an optional argument of type BYTE specifying the plotting mode desired. **Mode** defaults to the current value of S_MODE (which is normally 1=draw unless you have altered it). Legal values for **Mode** are 0=ignore, 1=draw, 2=erase, and 3=flip. The graphic cursor position is not changed when S_ARC returns.



Octant	Bit Mask
0	$01
1	$02
2	$04
3	$08
4	$10
5	$20
6	$40
7	$80

Specify **Octmask** as the sum of the bit mask values above for the octants you wish to be plotted.

Figure 6: Octants for S_ARC

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD
...
BEGIN
S_INIT
S_MOVE 50,50
S_ARC 25, $FF    ; Draw a circle of radius 25 centered at (50,50)
S_MOVE 100,0
S_ARC 10, $0F, 3  ;Flip semi circle 0-180 degrees
...
S_ARC 10, $0F, 3  ;Flip semi circle back off
...
S_END
END
```

NOTES:

1. S_ARC should not be used to plot circles or arcs which are partially off-screen; the portions on-screen may be incorrectly drawn if you do so.

2. Circles may look egg-shaped rather than circular. This is due to the fact that the pixels on the screen are not the same height and width. If this is a problem, you may wish to use W_ARC1 or W_ARC2 instead, which can specify an aspect ratio to correct the appearance of a circle.

3. S_ARC is well-suited to flip mode operation because it draws all the pixels individually, not as a series of line segments.

FUNC BYTE S_FILL FILL AN ENCLOSED AREA WITH A PATTERN

USAGE:

Bytevar = S_FILL (Pattern, Buffer, Buffersize)

DESCRIPTION:

Function S_FILL is one of the most powerful and interesting subroutines in the PROMAL GRAPHICS PACKAGE. It can be used to fill an arbitrary-shaped enclosed area on the screen with a pattern, such as a cross-hatch. The filling will be begun at the current graphic cursor, (S_X, S_Y) and expand in all directions until a solid boundary line (one bits) is encountered. **Pattern** is the address of a 16 word table defining the pattern to be used for fill. This pattern is a 16 dot by 16 dot template which will be used repetitively throughout the filled area. See Note 1 below for the format of the Pattern definition. The file FILLDEFS.S contains DATA statements for 16 different pre-defined fill patterns for your convenience, as shown in **Figure 7. Buffer** is the address of a scratch area of memory that S_FILL can use for calculations while it is running. For most shapes, a buffer of a couple hundred bytes will be sufficient. For filling very complex shapes, a larger buffer may be required. **Buffersize** is the size in bytes of the buffer area. The function returns a BYTE value which is TRUE if the fill operation was successful, and FALSE if the fill could not be completed because the buffer was not large enough. The position of the graphic cursor, (S_X, S_Y) is not altered.

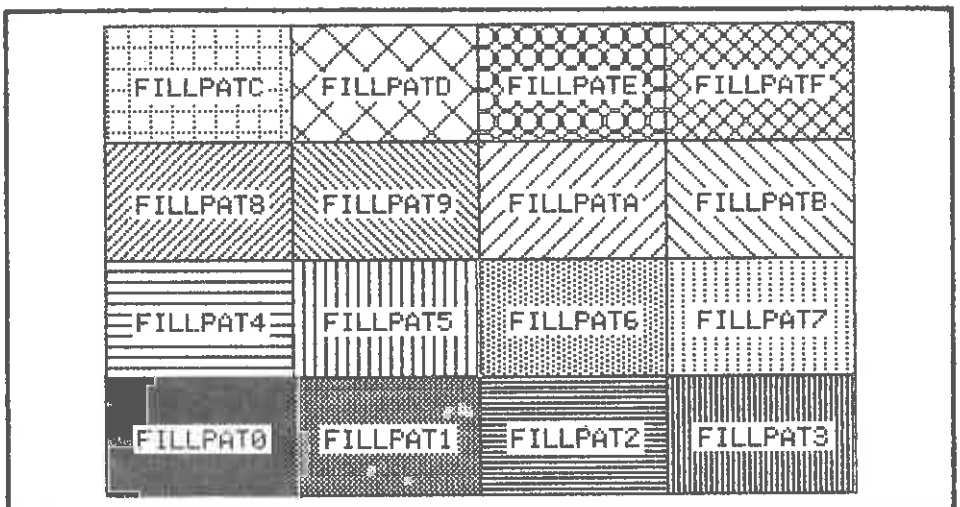


Figure 7: Predefined Fill Patterns in File FILLDEFS.S

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
INCLUDE FILLDEFS    ; 16 Pre-defined fill patterns
CON WORD SCRATCHSZ=200
BYTE SCRATCH [SCRATCHSZ]
...
BEGIN
S_INIT
S_MOVE 100,90
S_ARC 30, $FF       ; Draw two concentric circles
S_ARC 60, $FF
IF NOT S_FILL (FILLPAT0, SCRATCH, SCRATCHSZ)  ; Fill center circle
  ABORT "#CNEED BIGGER SCRATCH FOR S_FILL"
S_MOVE 100,50
IF NOT S_FILL (FILLPAT1, SCRATCH, SCRATCHSZ)  ; Fill outer circle
  ABORT "#CNEED BIGGER SCRATCH FOR S_FILL"
...
S_END
END
```

NOTES:

1. The fill pattern is 16 words of 16 bits each. Each 1 bit turns on a pixel; each 0 bit turns off a pixel. The Utility program MAKEFILL is included to make it easy to define fill patterns of your own using the EDITor to "draw" the patterns desired. This program was used to define the fill patterns in FILLDEFS.S. See the comments in the source program MAKEFILL.S for an explanation of how to use the program. The file FILLDEFS.T was created with the EDITor as input to MAKEFILL.S.

2. If you try to fill an area that is already filled with a pattern, the S_FILL function will just treat it as a very complex shape to be filled and will try to fill in between the existing filled dots. This will probably result in the function running out of scratch buffer space and returning FALSE.

3. Even a tiny opening in an enclosed shape will allow the fill pattern to "spill out" and fill the entire screen.

PROC S_GETDOTS EXTRACT A ROW OF PIXELS FROM THE SCREEN

USAGE:

```
S_GETDOTS DX, Buffer
```

DESCRIPTION:

Procedure S_GETDOTS extracts a horizontal row of dots from the screen and copies them to a an array of bytes, one pixel per bit. This procedure is useful for writing subroutines to save, restore, or print an image or part of an image. The procedure will copy pixels starting at the current cursor position, (S_X, S_Y). **DX** is an INTeger specifying the width of the row of pixels desired. **Buffer** is the address of the destination array. Pixels will be packed into **Buffer** eight pixels per byte, with the left most pixel in the high order bit of the first byte. A total of **DX+1** pixels will be copied; any fractional bytes will be zero-filled. The minimum size required for **Buffer** is (DX+8)/8 bytes. S_GETDOTS will copy only as far a the end of the current horizontal line; it will not "wrap around" to the next line even if you specify a **DX** larger than the number of remaining pixels on the line.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BYTE ROWDOTS [40]  ; Big enough for 320 dots
BEGIN
S_INIT
...
S_MOVE 0,50
S_GETDOTS 100, ROWDOTS  ; Save 101 pixels (0,50) through (100,50)
...
S_END
END
```

NOTES:

1. The subroutines S_SV_IMAGE, S_PUT_IMAGE and S_PRT_IMAGE in file IMAGE.S provide excellent examples of how to use this procedure to save all or part of an image in memory, to a disk file, or to the printer. You may wish to study these subroutines to learn how to use S_GETDOTS effectively.

PROC S_PUTDOTS INSTALL A HORIZONTAL ROW OF PIXELS INTO SCREEN MEMORY

USAGE:

```
S_PUTDOTS Buffer, DX [, Setcolor]
```

DESCRIPTION:

Procedure S_PUTDOTS performs the inverse operation of procedure S_GETDOTS. S_PUTDOTS copies a group of pixels packed as bits in an array onto the screen. The pixels are installed starting at the current graphic cursor location, (S_X, S_Y). **Buffer** is the address of the array of bytes. The buffer has pixels packed 8 per byte, with the most significant bit being the left most pixel. **DX** is an integer specifying the width in pixels to be copied. A total of **DX+1** pixels will be extracted from the array. **Setcolor** is an optional argument of type BYTE, pertaining only to the Commodore 64; it will be ignored on the Apple. If **Setcolor** is TRUE, then the pixels copied onto the screen will be copied in the currently selected color and background, S_FORECOLOR and S_BACKCOLOR. If **Setcolor** is omitted or FALSE, the pixels will be drawn in whatever color combination is currently on the screen at their position.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
BYTE ROWDOTS [40]  ; Big enough for 320 dots
BEGIN
S_INIT
...
S_MOVE 10,100
S_GETDOTS 50, ROWDOTS  ; Save 51 pixels (10,100) thru (60,100)
...
S_CLEAR
...
S_MOVE 10,100
S_PUTDOTS ROWDOTS, 50  ; Restore 51 pixels of old image
...
S_END
END
```

NOTES:

1. By using S_PUTDOTS it is possible to write subroutines to copy all or parts of images from other sources whose format is known (such as other graphic application programs) onto the screen and modify the image using the subroutines in the PROMAL GRAPHIC TOOLBOX. It is also possible to achieve effects such as vertical scrolling of graphics images. The subroutines S_RCL_IMAGE and S_GET_IMAGE in the file IMAGE.S illustrate how to recall images or parts of images from memory or disk. You may find it worthwhile to study these subroutines to see how to use S_PUTDOTS effectively.

FUNC WORD S_XYADDR COMPUTE THE ADDRESS OF THE CURSOR

USAGE:

```
Wordvar = XYADDR
```

DESCRIPTION:

Function S_XYADDR computes the address of the byte in screen memory containing the pixel at the graphic cursor, (S_X, S_Y). It returns the address as the value of the function. In addition, it returns a one-bit mask in the global variable S_MASK. This mask can be ANDed with the byte at the returned address to extract the pixel from memory. A 0 result means the pixel is off; a non-zero result means the pixel is on (1).

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
...
FUNC BYTE ISON ; X , Y
  ; Return TRUE if pixel at (X, Y) is on,
  ; otherwise return FALSE.  Does not affect
  ; the cursor location.
ARG INT X
ARG INT Y
INT XSV
INT YSV
BYTE TEMP
BEGIN
XSV=S_X     ; Save cursor location
YSV=S_Y
TEMP = (S_XYADDR@< AND MASK) <> 0
S_MOVE XSV, YSV
RETURN TEMP
END
...
BEGIN
```

```
S_INIT
...
IF ISON(0,0)
  PUT CR, "The pixel at the origin is on."
...
S_END
END
```

NOTES:

1. For the Commodore 64, the address of the color byte controlling the colors at (S_X, S_Y) can be computed using the following statements:

```
WORD COLORPTR
BYTE FORECOLOR
BYTE BACKCOLOR
...
COLORPTR = ((S_XYADDR - S_HRPAGE:+ << 8 ) >> 3) + $8C00
```

The color codes for the pixel at the cursor can then be extracted like this:

```
FORECOLOR = COLORPTR@< >> 4
BACKCOLOR = COLORPTR@< AND $0F
```

This completes the description of the SGD subroutines. You may wish to study some of the example programs to learn more about how to use the SGD procedures and functions. The following section describes another part of the PROMAL GRAPHICS TOOLBOX, the Window Graphic System (WGS).

WINDOW GRAPHICS SYSTEM (WGS)

INTRODUCTION

The Window Graphics System (WGS) is the name given to a collection of PROMAL subroutines included with the GRAPHICS TOOLBOX. These subroutines are provided both as a pre-compiled module and in source form, so you can add to them or modify them if you wish (CAUTION: If you modify the WGS in any way, be sure to modify a copy of the file and **give it a new name** other than WGS).

So far all the subroutines have dealt with **screen coordinates**, expressed as **INTegers**. The distinguishing feature of the WGS is that coordinates are expressed as type **REAL** values. The big advantage of the WGS is that you give it coordinates in any unit of measure that is appropriate to your application, and it automatically scales the coordinates to the screen. For example, if you were plotting a sine wave, your Y coordinates might range from -1.0 to +1.0, but if you were plotting a map of the USA with data in miles, coordinates might run from 0.0 to 3500. The X and Y axis can be independently scaled. For example, the X axis might run from 0.0 to 0.1 microseconds and the Y axis from 200000.0 to 800000.0 angstroms for some scientific data. In graphics parlance, this kind of coordinate system is called a **world** coordinate system.

A second distinguishing feature of the WGS is the use of **viewports** and **windows**. A viewport is a rectangular area of the screen, in which all WGS graphics will be drawn. The **viewport** is defined by giving the **screen coordinates in pixels (as INTegers)** of where you want the viewport on the screen. It can be as large as full screen or as small as 8 pixels wide by three pixels high.

The **window** tells what dimensions you want associated with the viewport, so that the WGS knows how to scale your data for plotting. To specify a window, you tell the WGS what coordinates should be plotted at the corners of the viewports, **expressed as REAL values** in terms of your data.

An example may help clarify the relation of the viewport and the window. Suppose you want to plot a line graph of sales for the year, with months along the X axis going from 1 to 12, and dollars along the Y axis going from $500,000 to $800,000. Further suppose that you wanted this graph to be plotted with the lower left hand corner at screen coordinate (24, 30) and the upper right hand corner at (216, 150). In this case your **viewport** would be defined as a rectangle with vertices at (24, 30) and (216, 150), and the corresponding window would be defined by (1.0, 500000.) and (12.0, 800000.), as shown in **Figure 8.**

Another very useful characteristic of the WGS is that the lines drawn by the WGS will be **clipped** at the viewport boundaries. This means that lines will not be displayed outside the viewport. Clipping is illustrated in **Figure 9.** Here the WGS was told to plot a diagonal line which extended beyond the window boundaries. However, only the part of the line inside the viewport (represented by the rectangle) was actually drawn; the portion of the line outside the viewport (represented by the dashed line) was **not** drawn.

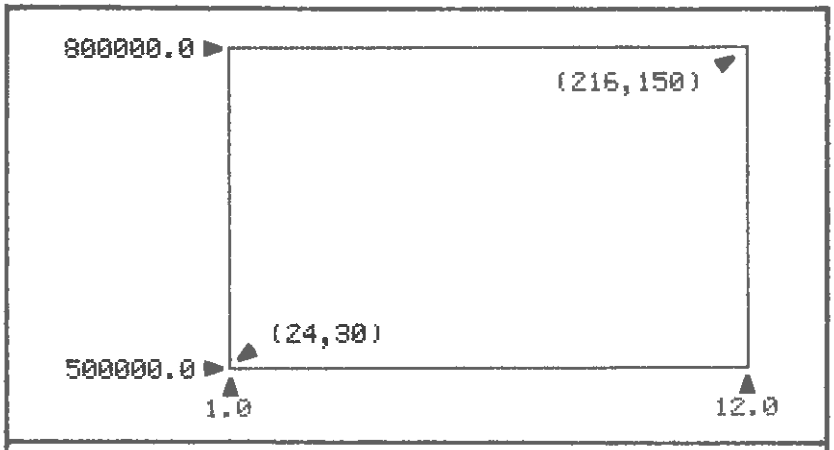


Figure 8: Relationship of Viewport and Window

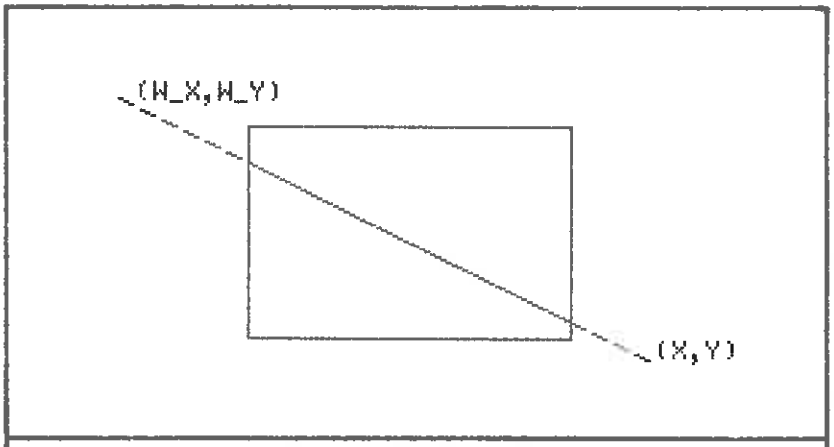


Figure 9: Clipping

The clipping feature is very important for certain kinds of graphics. For example, suppose you plotted a mechanical drawing of a machine part. If you wanted to "zoom in" to only see a portion of the part "close up", you could simply change the window definition and redraw **the exact same data** as before. Only the part of the plotted data defined by the window would be plotted.

WGS CURSOR

Like the SGD, the WGS has an invisible graphic cursor. The current position of the WGS cursor is expressed as (W_X, W_Y). The exported **REAL** variables **W_X** and **W_Y** give the current WGS cursor position, and are updated automatically as you perform WGS drawing. Unlike the SGD, **any** values are legal for W_X and W_Y, not just values within the window. For instance, if your window is defined from X = 1.0 to X = 12.0, it is perfectly legal to set S_X to 15.32, 100000.0, or -261.42. If you draw beyond the boundaries of the window, nothing will be

When you call a WGS subroutine which moves the cursor, it also moves the SGD cursor, provided that the cursor is in the viewport. However, the SGD cursor will always be "clipped" at the viewport boundaries.

Please note that the WGS routines call the SGD routines to actually perform the plotting. You may freely mix SGD calls with WGS calls. Naturally, if you call an SGD routine, it will not be bounded by the viewport or window established for the WGS; the SGD routines work just like they always did. Also note that moving the SGD cursor does **not** update the position of the WGS cursor.

This relationship between the WGS and the SGD is very handy. For example, you can plot your data using the WGS calls, but annotate your axes outside of the viewport by using SGD calls. The SALESDEMO.S file uses this method, as well as the example given in the description of W_AXIS later in this section.

USING THE WGS

In order to use the WGS, you need to set up your program as shown in the skeleton program below:

```
PROGRAM Name OWN
INCLUDE LIBRARY
INCLUDE SGD.E
INCLUDE WGS.E
...
BEGIN
S_INIT
W_INIT
; ... Graphics calls here as needed here...
S_END
END
```

You need to have the line **INCLUDE WGS.E** in your program to make the definitions of the WGS subroutines and variables known to the compiler. Note that you still need to INCLUDE SGD.E as well. In your program, you should call S_INIT first and then W_INIT to initialize the graphics system before using other graphics calls. W_INIT will be described in the following section.

When loading your program, you need to load the SGD first, then the WGS, and then your application program. This can be conveniently accomplished by using the BOOTW command from the EXECUTIVE. For example:

```
UNLOAD
BOOTW MYPROG
```

This loads the SGD, WGS, and then loads and executes MYPROG.C. The source program for BOOTW is included on the disk if you wish to modify a copy of it for your specialized needs.

Table 6
WGS SUBROUTINE SUMMARY

Name	Description
W_INIT	Initialize WGS system and initialize variables.
W_VIEW	Specify desired viewport in screen coordinates.
W_WINDOW	Specify desired window in world coordinates.
W_CLEAR	Clear the current viewport.
W_BOXVP	Draw a rectangle around the current viewport.
W_MOVE	Move the cursor to the specified world coordinates.
W_DOT	Draw a dot at the specified world coordinates.
W_PLOT	Plot a line to the specified world coordinates in a given mode.
W_DRAW	Draw a line to the specified world coordinates.
W_AXIS	Draw axes with optional tic marks
W_ARC1	Draw a circle, ellipse, or arc from a point on the arc.
W_ARC2	Draw a circle, ellipse, or arc from the center.

Table 7
WGS Variables and Constants

Name	Type	Description
W_X	REAL	World cursor location, X coordinate.
W_Y	REAL	World cursor location, Y coordinate.
W_XMIN	REAL	Minimum viewable world X coordinate.
W_YMIN	REAL	Minimum viewable world Y coordinate.
W_XMAX	REAL	Maximum viewable world X coordinate.
W_YMAX	REAL	Maximum viewable world Y coordinate.
W_VXMIN	INT	Screen X coordinate of left edge of viewport.
W_VYMIN	INT	Screen Y coordinate of bottom edge of viewport.
W_VXMAX	INT	Screen X coordinate of right edge of viewport.
W_VYMAX	INT	Screen Y coordinate of top edge of viewport.
W_TIC_DX	REAL	Tic interval for X axis for Procedure W_AXIS
W_TIC_DY	REAL	Tic interval for Y axis for Procedure W_AXIS
W_TIC_SIZE	INT	Tic size in pixels for Procedure W_AXIS

Notes for Table 7:

1. See PROC W_AXIS for information about the following additional EXPORTED CONstant names: W_X_AXIS, W_Y_AXIS, W_POS_TICS, W_NEG_TICS, W_BOTH_TICS.

SUMMARY OF WGS SUBROUTINES AND VARIABLES

Table 6 summarizes the subroutines which are EXPORTed by the WGS, which you may call from your application programs.

The WGS is set up as a separately compiled PROMAL module. It EXPORTs the variables described in **Table 7**, which may be accessed by your application program.

```
**********************************************************************
*                                                                    *
*                          IMPORTANT!                                *
*                                                                    *
*   When using the WGS routines, be very careful to use REAL         *
*   coordinates where expected.  If you use an INTeger, BYTE         *
*   or WORD value where a REAL is expected, it will not be           *
*   detected as a compilation error, but will produce very           *
*   strange results, possibly crashing the system!  In               *
*   particular remember that REAL constants must be written          *
*   with a decimal point.  Therefore W_DRAW 0, 10 is wrong           *
*   but W_DRAW 0., 10. is correct.                                   *
*                                                                    *
**********************************************************************
```

DETAILED DESCRIPTION OF WGS SUBROUTINES

This section describes the WGS subroutines individually. An example is given in the discussion of W_DRAW illustrating W_INIT, W_VIEW, W_WINDOW, W_CLEAR, W_BOXVP, W_MOVE, W_PLOT and W_DRAW. The W_AXIS, W_ARC1 and W_ARC2 discussions have their own examples.

PROC W_INIT INITIALIZE WGS

USAGE:

W_INIT

DESCRIPTION:

Procedure W_INIT initializes the WGS software. It should be called **after** S_INIT but **before** any other calls to WGS routines.

PROC W_VIEW DEFINE CURRENT VIEWPORT

USAGE:

W_VIEW SX0, SY0, SX1, SY1

DESCRIPTION:

Procedure W_VIEW is used to define a new viewport on the screen. **SX0**, **SY0**, **SX1**, and **SY1** are all **INTeger** arguments (but BYTE or WORD expressions are also acceptable) defining the desired coordinates for the lower left hand corner of the viewport, (**SX0**, **SY0**), and upper right hand corner of the viewport, (**SX1**, **SY1**). Thereafter, any plotting done by WGS routines will be performed in this viewport.

NOTES:

1. You may define any number of viewports on the screen. However, only the most recently defined will be active. If you wish to have several viewports on the screen at once and "switch back and forth" between them, you should save the "state" of each viewport (by saving the values for W_VIEW, W_WINDOW, and the cursor coordinates) and make a new call to W_VIEW and W_WINDOW each time you switch.

2. For future compatibility with the IBM version of the PROMAL GRAPHICS PACKAGE (when available), you may wish to make the values of **SX0** and **SX1** evenly divisible by 8. However, they may be any value up to the limits of the screen size for the Apple and Commodore versions.

3. The pixels defining the viewport are considered to be part of the viewport and therefore can be plotted on with WGS calls.

PROC W_WINDOW DEFINE CURRENT WINDOW BOUNDS

USAGE:

W_WINDOW X0, Y0, X1, Y1

DESCRIPTION:

Procedure W_WINDOW defines the world coordinates to be assigned to the limits of the currently-defined viewport. **X0**, **Y0**, **X1**, and **Y1** are all type **REAL** arguments, defining the minimum and maximum values to be associated with the viewport. (**X0**, **Y0**) is the world coordinate to be plotted at the lower left hand corner of the viewport, and (**X1**, **Y1**) is the world coordinate to be defined at the upper right hand corner. Subsequent calls to WGS plotting subroutines will only display values within this range; plotting outside the window limits will cause "clipping".

PROC W_CLEAR MOVE THE VIEWPORT

USAGE:

W_CLEAR

DESCRIPTION:

Procedure W_CLEAR clears the currently defined viewport. It does not affect any of the image on the rest of the screen.

PROC W_BOXVP DRAW A BOX AROUND THE VIEWPORT

USAGE:

W_BOXVP

DESCRIPTION:

Procedure W_BOXVP draws a box around the currently defined viewport.

PROC W_MOVE MOVE THE WORLD CURSOR

USAGE:

W_MOVE X, Y

DESCRIPTION:

Procedure W_MOVE moves the world graphic cursor. **X** and **Y** are **REAL** values for the desired position. Any REAL values may be used. The world cursor, (W_X, W_Y) will be set to (X, Y). If the resulting position is in the window, the screen cursor (S_X, S_Y) will also be updated to the corresponding position.

PROC W_DOT — DRAW A DOT IN WORLD COORDINATES

USAGE:

```
W_DOT X, Y
```

DESCRIPTION:

Procedure W_DOT draws a single pixel at the specified world coordinates. **X** and **Y** are the **REAL** values desired for the coordinates. If (X, Y) is outside the currently defined window, the point will not be drawn; however, the world cursor (W_X, W_Y) will be updated to the new coordinates in any event.

PROC W_PLOT — PLOT A LINE IN WORLD COORDINATES

USAGE:

```
W_PLOT X, Y, MODE
```

DESCRIPTION:

Procedure W_PLOT plots a line from the current world cursor position, (W_X, W_Y), to the specified world coordinates. **X** and **Y** are **REAL** values for the desired coordinates. **Mode** is a BYTE value indicating the desired plotting mode. Legal values for **Mode** are 0=move, 1=draw, 2=erase, 3=flip, and 4=dashed. If **Mode** is 4 then the dashed line pattern used is the current value of S_DASHPIC. Please note that **Mode** is a **required** argument. If all or part of the line to be plotted is not inside the current window boundaries, only the part that is inside the boundaries will be drawn. Clipping will be done at the viewport boundary. The world cursor will be update to the specified coordinates.

NOTES:

1. You must define a viewport and a window before plotting any data.

PROC W_DRAW DRAW A LINE IN WORLD COORDINATES

USAGE:

```
W_DRAW X, Y
```

DESCRIPTION:

Procedure W_DRAW draws a solid line from the current world cursor position, (W_X, W_Y), to the specified coordinates. **X** and **Y** are the desired **REAL** coordinates. If all or part of the line is outside the current window limits, only the part within the window boundaries will be drawn. Clipping will be done at the viewport boundaries. The world cursor, (W_X, W_Y) will be updated to the specified coordinates.

EXAMPLE:

```
PROGRAM SINPLOT OWN
INCLUDE LIBRARY
INCLUDE SGD.E
INCLUDE WGS.E
INCLUDE REALFUNCS  ; Trig functions from End User disk
DATA REAL PI = 3.14159
REAL THETA
...
BEGIN
S_INIT
W_INIT
W_VIEW 20,20, 220,160  ; Define 200 x 140 pixel viewport
W_WINDOW 0.,-1., 2.*PI,1. ; Window X 0 to 2*pi, Y -1 to +1
W_CLEAR                ; Erase viewport
W_BOXVP                ; Frame viewport
W_MOVE 0., 0.
W_PLOT 2.*PI, 0., 4    ; Draw horizontal dashed centerline
THETA = 0.
W_MOVE THETA, SIN(THETA)
REPEAT
  THETA = THETA + PI/16.
  W_DRAW THETA, SIN(THETA)  ; Plot sine wave
UNTIL THETA >= 2.*PI
...
S_END
END
```

The resulting image is shown in **Figure 10.**

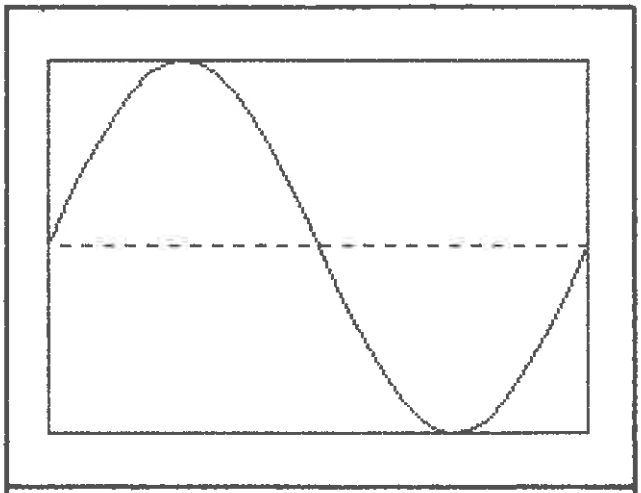


Figure 10: SINPLOT Program Output

PROC W_AXIS DRAW AXIS

USAGE:

W_AXIS Kind

DESCRIPTION:

Procedure W_AXIS draws an X or Y axis (or both) for plotting data, optionally with tic marks. **Kind** is an argument of type **BYTE** giving a code for the kind and number of axes to be drawn. The axes will be drawn in the currently defined window, starting from the world cursor position, (W_X, W_Y). **Kind** should be specified by simply adding any of the following constants (which are each masks for 1-bit flags EXPORTed by WGS) together:

Constant name	Meaning (and defined value)
W_X_AXIS	Plot an axis in the X direction ($01).
W_Y_AXIS	Plot an axis in the Y direction ($02).
W_POS_TICS	Draw tic marks on the positive side of the axis ($04).
W_NEG_TICS	Draw tic marks on the negative side of the axis ($08).
W_BOTH_TICS	Draw tic marks on both sides of the axis ($0C).

Prior to calling W_AXIS, you should set the following EXPORTED variables, if you want tic marks on your axes:

Variable name	Type	Meaning
W_TIC_DX	REAL	Tic interval for X axis.
W_TIC_DY	REAL	Tic interval for Y axis.
W_TIC_SIZE	INT	Tic size in pixels.

EXAMPLE:

```
PROGRAM SALESPLOT OWN
INCLUDE LIBRARY
INCLUDE SGD.E
INCLUDE WGS.E
BYTE NUMBUF [10]
REAL Y
WORD I
BYTE MODE
DATA REAL SALES [] =  ; 1980 - 1990 sales in $millions (projected after 85)
  4.9, 4.3, 4.7, 5.1, 5.8, 5.9, 6.3, 6.8, 7.3, 7.8, 8.8
BEGIN
S_INIT
W_INIT
W_VIEWPORT 40,20, 260,170    ; Viewport with room around outside
W_WINDOW 1980.,4., 1990.,9.  ; Years in X, 4 to 9 in Y
W_MOVE 1980.,4.
W_TIC_DX = 1.0          ; 1 year between X tic marks
W_TIC_DY = 0.5          ; 0.5 between Y tic marks
W_TIC_SIZE = 4          ; 4 pixel tic marks
W_AXIS W_X_AXIS + W_Y_AXIS + W_NEG_TICS ; X & Y axis with outside tics
FOR I = 1980 TO 1990    ; For all X tic marks
  W_MOVE I:., 4.        ; Move to tic mark
  WORDSTR I-1900, NUMBUF    ; Convert year to string
  S_RELPLOT -6,-12,0    ; Move down 12 & left 6 pixels
  S_TEXT NUMBUF         ; Annotate year centered at tic mark
Y = 4.
REPEAT
  REALSTR Y, NUMBUF, 4, 1  ; Make Y into string, e.g., " 5.5"
  W_MOVE 1980., Y
  S_RELPLOT -30, -3, 0 ; Left & center of Y tic marks
  S_TEXT NUMBUF         ; Annotate tic mark
  Y = Y + 0.5
UNTIL Y > 9.0
MODE = 1                ;   Draw
S_DASHPIC=$3333
W_MOVE 1980., SALES [0]
FOR I = 1980 TO 1990
  W_PLOT I:., SALES [I-1980], MODE
  IF I >= 1985
    MODE = 4 ; Dashed line for projected sales
S_MOVE 8,20
S_TEXT "Annual Sales in $Million", 1
S_MOVE 100,150
S_TEXT "WIDGETS",0,2    ; 2X Title
IF GETC      ; Wait for any key
  S_END
END
```

This program produced the output in Figure 11.

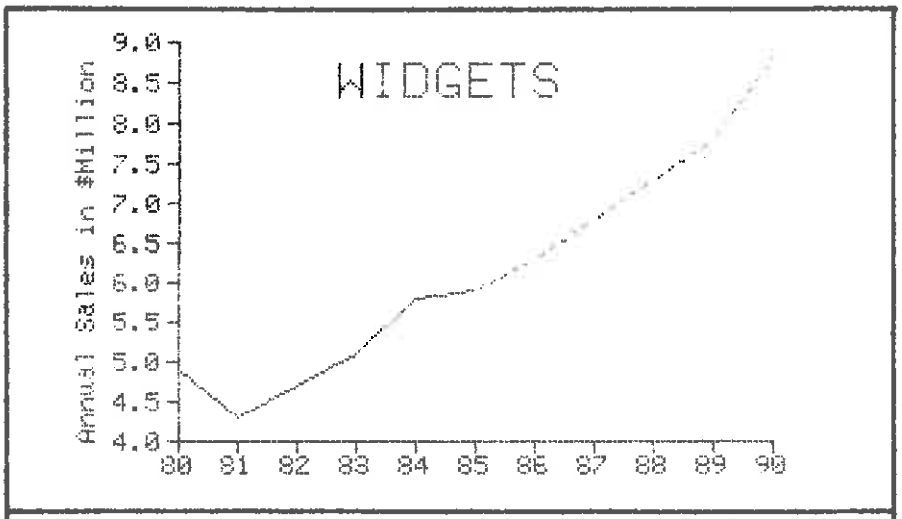


Figure 11: Output from W_AXIS Example

PROC W_ARC1	DRAW AN ARC FROM AN ENDPOINT

USAGE:

W_ARC1 XC, YC, Angle, Aspect

DESCRIPTION:

Procedure W_ARC1 draws a circle, an ellipse, or any part of a circle or ellipse, with the present world cursor location, (W_X, W_Y) as the starting point for the arc. **XC** and **YC** are the desired **REAL** center coordinates for the circle or ellipse (which does not have to be inside the window). **Angle** is the desired **REAL** angle measured in **degrees** to be drawn (e.g., 360. for a full circle). The arc will be drawn counter-clockwise if **Angle** is positive. **Aspect** is the desired aspect ratio (ratio of Y to X), normally 1.0 for a circle. On completion, the world cursor (W_X, W_Y) will be left at the end point of the arc. The arc will be clipped at the window boundaries if necessary.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
INCLUDE WGS.E
...
BEGIN
S_INIT
W_INIT
W_VIEW 0,0, 200,150
W_WINDOW 0.,0., 200.,150.
W_MOVE 150., 75.
W_ARC 100.,75., 360., 1.   ; Draw circle centered at (100,75)
W_ARC 0.,75., 30., 1.      ; Draw 30 degree arc centered at (0,75)
...
S_END
END
```

NOTES:

1. For very small circles, S_ARC will generally give a better appearing circle than W_ARC1 or W_ARC2.

2. W_ARC1 or W_ARC2 may seem to take some time to plot large circles which are partially out of the window, since all the points on the circle are computed even if they are not plotted.

3. The aspect ratio can be used to plot ellipses or correct the oblong appearance of circles. Remember that the selection of viewport dimensions and windows also effects the appearance of circles.

PROC W_ARC2 DRAW AN ARC GIVEN THE CENTER

USAGE:

W_ARC2 Radius, Angle1, Angle2, Aspect

DESCRIPTION:

Procedure W_ARC2 is similar to W_ARC1 except the world cursor is at the center of the circle. **Radius** is the desired **REAL** value for the radius. **Angle1** is the desired **REAL** starting angle for the arc. **Angle2** is the desired ending angle for the arc. Both angles are measured in **degrees**. **Aspect** is the **REAL** aspect ratio (ratio of Y to X), normally 1.0. When the arc is completed, the world cursor will be left at the ending point on the arc. The arc will be clipped at the window boundaries if necessary.

EXAMPLE:

```
PROGRAM GRAPHIX OWN
INCLUDE LIBRARY
INCLUDE SGD.E
INCLUDE WGS.E
DATA PIE [] = 0., 30., 100., 180., 250., 360.
...
BEGIN
S_INIT
W_INIT
W_VIEW 0,0, 160,160
W_WINDOW -1.,-1., 1., 1.
W_MOVE 0., 0.
FOR I = 0 TO 5
  W_ARC2 0.8, PIE[I]  ; Draw a pie chart from data
  W_DRAW 0., 0.
...
S_END
END
```

UTILITY SUBROUTINES

In addition to the SGD and WGS, a group of graphics utility subroutines is provided in source form in the file **IMAGE.S**. To use these subroutines, you may INCLUDE IMAGE in your program, or use the EDITor to extract only the subroutines you need. The comments in the file explain the operation of the subroutines in some detail, so you can modify them to meet your special needs. These subroutines require the SGD, but not the WGS. The IMAGE subroutines are described briefly below.

PROC S_SV_IMAGE COPY IMAGE TO MEMORY

USAGE:

```
S_SV_IMAGE Buf, DX, DY
```

DESCRIPTION:

Procedure S_SV_IMAGE copies all or part of high-resolution screen memory into a memory array. The graphics cursor, (S_X, S_Y) is at the lower left hand corner of the rectangular area to be saved. **Buf** is the address of an array of type BYTE to hold the copy of the image. The size of **Buf** must be at least DY*((DX+8)/8) bytes. **DX** and **DY** are INTeger values for the size of the rectangle to be saved, measured in pixels. These values should be positive and not exceed the remaining screen size.

EXAMPLE:

```
BYTE FIG_SAVE [2000]
...
S_MOVE 0,0
S_SV_IMAGE FIG_SAVE, S_XMAX/2, S_YMAX/2 ; Save 1/4th of screen
...
```

NOTES:

1. Color information is not saved (Commodore 64).

PROC S_RCL_IMAGE RESTORE IMAGE FROM MEMORY

USAGE:

```
S_RCL_IMAGE Buf, DX, DY
```

DESCRIPTION:

Procedure S_RCL_IMAGE restores an image or part of an image previously saved using S_SV_IMAGE. **Buf** is the address of the array, **DX** and **DY** are the size of the saved image in pixels (which must be the same as when it was saved), and the graphic cursor is at the desired lower left hand corner for the image (which does not have to be the same).

EXAMPLE:

```
S_RCL_IMAGE FIG_SAVE, S_XMAX/2, S_YMAX/2
```

PROC S_PUT_IMAGE COPY IMAGE TO DISK

USAGE:

```
S_PUT_IMAGE Handle, DX, DY
```

DESCRIPTION:

Procedure S_PUT_IMAGE copies all or a rectangular portion of an image to a disk file. **Handle** is the file handle for a file previously opened in write mode. **DX** and **DY** are INTeger values for the size of the image to be save in pixels, and should be positive. The graphics cursor, (S_X, S_Y) will be used as the lower left hand corner of the image.

EXAMPLE:

```
WORD MYFILE
...
MYFILE=OPEN("IMAGE1.G",'W')
...
S_MOVE 0,0
S_PUT_IMAGE S_XMAX, S_YMAX  ; Save whole screen
...
```

NOTES:

1. See the source code comments for the file format.
2. Color information is not saved in the present implementation.

PROC S_GET_IMAGE RECALL IMAGE FROM DISK

USAGE:

S_GET_IMAGE Handle, Move

DESCRIPTION:

Procedure S_GET_IMAGE recalls an image previously saved with S_PUT_IMAGE on disk. **Handle** is an open file handle for the file to read. **Move** is a flag. If FALSE, the image will be displayed in the same location it was stored at. If TRUE, the image will be displayed with the lower left hand corner at the current location of the graphic cursor.

EXAMPLE:

```
WORD MYFILE
...
MYFILE=OPEN("IMAGE1.G")
...
S_GET_IMAGE MYFILE, FALSE
...
```

PROC S_PRT_IMAGE PRINT IMAGE

USAGE:

S_PRT_IMAGE Handle, DX, DY, Margin

DESCRIPTION:

Procedure S_PRT_IMAGE prints all or a rectangular portion of the high-resolution graphics screen on a printer. **Handle** is the open file handle for the printer. **DX** and **DY** are positive integers giving the desired size to print in pixels. **Margin** is and integer giving the desired left hand margin for the printer in terms of dots. The image will be printed with the lower left hand corner at the current graphics cursor position, (S_X, S_Y).

EXAMPLE:

```
WORD PRTR
...
PRTR=OPEN("P",'W')
...
S_MOVE 0,0
S_PTR_IMAGE PRTR, S_XMAX, S_YMAX, 50 ; Print screen, 50 dot margin
```

NOTES:

1. **IMPORTANT:** Unfortunately, there is no "standard" way to tell a printer to print graphics. Although the methods are usually similar, each printer manufacturer chooses his own way to implement graphics commands. Since it is impossible for SMA to develop individual printer drivers for the hundreds of printers available, we have instead included the source code to work with **one** printer in the file IMAGE.S. In all likelihood, **this subroutine will not work with your printer until you modify it for the particular commands your printer expects.** Also, some printers (such as the Commodore 1526) are simply not capable of printing bit-mapped graphics. You will need to consult your printer manual to find out what your printer needs. In general, you will need to change:

a. The escape sequence used to change the line spacing. In the furnished subroutine, the sequence is ESC, n, ESC, T16. If you wind up with horizontal bands of whitespace in your printed image, it may be because you have not set the vertical line spacing to match graphics mode.

b. The escape sequence to enter graphics mode. In the furnished subroutine, the sequence is ESC, G, nnnn, where nnnn is four ASCII decimal digits giving the number of graphic bytes to follow. Some printers only send 3 digits or don't send ASCII values, in which case you will need to alter the WORDSTR call in the furnished procedure.

c. The sequence for enabling uni-directional mode. Most printers have more satisfactory vertical alignment in uni-directional rather than bi-directional mode.

d. The sequence for restoring normal mode when the image is printed. In the furnished subroutine, this sequence is ESC, L000, ESC, N, ESC, A, ESC, (.

e. On the Apple, ProDOS normally intercepts printer output and process some data bytes as special commands. The furnished print driver defeats this so that the data will be passed through. If you have a Commodore, you will need to remove the statements which set APLPALF.

f. On the Commodore 64, PROMAL normally converts printer data output from ASCII to "Commodore ASCII". When sending ESCape sequences or graphics data, you will need to defeat this by setting C64PUL (BYTE at address $0DF4) to 0 at the start of the subroutine and restoring it to its entry state when the procedure ends.

2. If you abort a program (or if it otherwise terminates) while printing a graphics image, you will probably have to turn the printer off and back on to get it to operate normally again.

3. The procedure S_PRINT in file IMAGE.S prints the entire screen.

APPENDIX A

USER-DEFINED TEXT FONTS

The S_TEXT procedure is capable of drawing text using other character sets besides the one provided. You may devise your own fonts, which may define characters with any size and shape. Characters are drawn by S_TEXT using lines. The format for the font is very similar to the format for the shape table used by procedure S_SHAPE. The file SGDFONT.A on the distribution disk contains an assembly language definition of the standard font used by S_TEXT. It contains comments explaining the format of a font table. You do not have to use an assembler to generate a new font; it merely was a convenient way of generating all the necessary data bytes for use with the SGD. You could define your font with PROMAL DATA statements, read it from a separate file, etc.

By studying the comments and the sample font in SGDFONT.A, you should be able to devise your own font. An average font table takes about 1K of memory if all ASCII characters are defined. You can define a font which only has a subset of the ASCII set (for example, only numeric digits for a big on-screen scoreboard). You may have as many different fonts in memory at once as there is room for. To change fonts, just change the SGD global variable S_FONT to point to the start of the desired font table in memory. S_FONT should point to the very first byte of the header for the font, as described in SGDFONT.A. Be sure to fill in the name field of the font header with a name for your font.

When designing your font, bear in mind that the graphic cursor, (S_X, S_Y) will be left wherever you complete a character. Therefore it is a good idea to make the last part of each character definition a MOVE to the normal starting position of the next character.

APPENDIX B

COMMODORE 64 SPRITE EDITOR

The Commodore 64 version of the PROMAL GRAPHICS PACKAGE disk includes a PROMAL program called SPRITEGEN (the source code is in file SPRITEGEN.S with include files SPRITEG2.S through SPRITEG6.S). This program allows you to interactively define the shape of a sprite and see the sprite on the screen. When you have defined your sprite, SPRITEGEN can write a file on disk containing the necessary PROMAL DATA statement to define the sprite in a program.

To use SPRITEGEN, execute it from the EXECUTIVE in the usual manner (it does not use any graphics routines, so you don't need the SGD). Select a mode by pressing a function key. You'll be guided by prompts on the screen. Here are the modes:

DRAW The sprite is drawn using the four cursor movement keys. As the cursor is moved, it draws, erases, or just moves, depending on the current "pen state". The pen state is set by pressing **D** (draw), **E** (erase) or **M** (move). The current state, which remains in effect until changed, is shown on the screen. The actual size sprite will be displayed at the lower right.

WRITE This permits a file to be written containing a PROMAL DATA statement that defines a 64 byte array of data that will cause the commodore hardware to properly render the sprite. You'll be asked for an array name and file name. If the file name is entered without an extension, ".S" is added. Prior to starting the write, you are told whether the file already exists and given the opportunity to cancel the write.

READ This permits a previously created sprite definition file to be read. Then, further editing can be done, and finally the updated sprite rewritten using **WRITE** mode.

EXIT Exits back to the EXECUTIVE.

SPRITEGEN writes each sprite definition to a separate file. You may later use the EDITor to combine several sprite DATA statements into a single file, or simply INCLUDE each file in your application program source. A sample DATA definition generated by SPRITEGEN is shown below:

```
DATA BYTE MISSLE1 [] =
  $FF,$FF,$FF, $80,$00,$01,
  $80,$00,$01, $80,$00,$01,
  $80,$00,$01, $80,$00,$01,
  $80,$00,$01, $80,$00,$01,
  $80,$00,$01, $80,$00,$01,
  $80,$00,$01, $80,$F4,$01,
  $80,$F4,$01, $80,$F4,$01,
  $80,$00,$01, $80,$00,$01,
  $80,$00,$01, $80,$00,$01,
  $80,$00,$01, $80,$00,$01,
  $FF,$FF,$FF, $00
```