

**S.A.L.A.D.
Still Another Line A Document**

COPYRIGHT

Copyright 1987 by Atari Corp. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of Atari Corp., 1196 Borregas Ave., Sunnyvale, CA 94086.

DISCLAIMER

ATARI CORP. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Atari Corp. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Atari Corp. to notify any person of such revision or changes.

TRADEMARKS

Atari, ST, and TOS are trademarks of Atari Corp.

Still Another Line A Document was printed in the United States of America.

First Edition: December 17, 1987

This document was written on an Atari Mega 4 computer using Microsoft Write and an Atari SLM804 laser printer.



Section 1: Introduction to "Line A"

TOS provides two software interfaces to its graphics routines: VDI and Line A. This document describes the "Line A" interface to the ST's low-level graphics primitives, as provided in all ST computers with TOS in ROM. It assumes you are familiar with the ST's operating system and 68000 assembly language.

While VDI is appropriate for many applications, it is sometimes advantageous to give up some convenience for speed and additional features, like support for all 16 Bit Blt logic operations in TextBlt. Line A provides an interface for simple graphics operations, with these additional features.

Line A Opcodes

Since Line A is an "underlying" portion of TOS, its interface to the world is designed more for the convenience of the operating system than for humans. The Line A interface consists of 16 opcodes, each of which is one word in length. The upper 4 bits are 1010 (A in hex, hence Line "A") and the lower 12 bits are used as the opcode field. The 15 opcodes are:

Initialization	(\$A000)	Return Line A pointers
Put Pixel	(\$A001)	Draw a pixel
Get Pixel	(\$A002)	Return the value of a pixel
Arbitrary Line	(\$A003)	Draw an arbitrary line
Horizontal Line	(\$A004)	Draw a horizontal line
Filled Rectangle	(\$A005)	Draw a filled rectangle
Filled Polygon	(\$A006)	Draw one hline of a filled polygon
BitBlt	(\$A007)	Move/copy a section of memory
TextBlt	(\$A008)	Move text to the screen
Show Mouse	(\$A009)	Show the mouse pointer
Hide Mouse	(\$A00A)	Hide the mouse pointer
Transform Mouse	(\$A00B)	Transform the mouse pointer
Undraw Sprite	(\$A00C)	Undraw software "sprite"
Draw Sprite	(\$A00D)	Draw software "sprite"
Copy Raster	(\$A00E)	Copy raster memory form
Seedfill	(\$A00F)	Seedfill

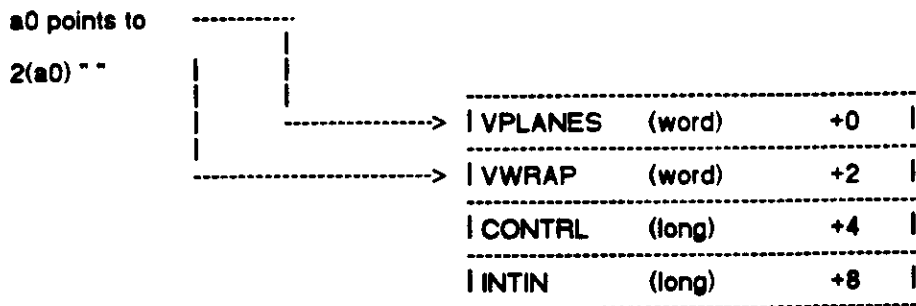
To use Line A (in two easy steps):

- Set up the input variables for the functions you need.
- Declare a constant word of \$A00X, where X is the number of your function.

When the 68000 encounters the \$A00X opcode, it performs a "Line A" exception, executes the Line A function, and returns control to your code.

Section 1: Introduction to "Line A"

Most Line A functions depend on a structure in memory for input parameters. The Line A Init call, \$A000, returns a pointer to this structure in both a0 and d0. The Line A variables are accessed relative to this value. Here's a diagram:



Once you have the pointer to this structure, you can use the Line A "offsets" to address specific variables. Some variables, like VPLANES (number of bit planes in current resolution), are global variables that reflect the current state of the system. Others, like X1 and Y1, are areas for passing parameters to Line A, which you must set up yourself. For a list of these offsets, see Section 3, "The Line A Variable Structure".

Line A allows your application to set "clipping boundaries" for many of its functions. These boundaries are set with the XMINCL, XMAXCL, YMINCL, and YMAXCL variables, which limit where Line A is allowed to draw. This prevents it from "coloring outside the lines." This is handy if you want to have an onscreen "window", and make sure Line A doesn't disturb anything outside its boundaries.

Many Line A functions support clipping. A few that do not are Init (obviously), PutPixel, GetPixel, Line, Horizontal Line, and BitBlt.

Section 2: Line A Function Reference

\$A000 -- Initialization

Returns several useful pointers, including the pointer to the Line A Variable Structure.

Input: None.

Returns:

- D0 = pointer to Line A variable structure
- A0 = pointer to Line A variable structure
- A1 = pointer to a null terminated array of pointers to the system font headers, allowing you to point to custom fonts in the TextBit call.
- A2 = pointer to a null terminated array of pointers to the Line A routines, allowing you to call the routines directly without incurring the overhead of processing a Line A exception. (You MUST be in supervisor mode to do this.)

Notes:

In order to make calls to most Line A routines, you will need to make this call once to get a pointer to the Line A Variable Structure. Once you have that pointer, save it somewhere, and you needn't use Line A init again.

Example:

```
dc.w    $A000                ; Make the Line A Init call
```

Section 2: Line A Function Reference

\$A001 -- Put Pixel

Plots a single pixel at the given X and Y coordinates.

Input:

INTIN[0] = Color value to use when plotting the pixel
PTSIN[0] = x coordinate for pixel
PTSIN[1] = y coordinate for pixel

Returns: Nothing.

Notes:

This function receives its parameters via the INTIN and PTSIN arrays. The Line-A Variable Structure contains pointers to these arrays, at offsets +B and +C, respectively. Offsets within the INTIN and PTSIN arrays are given in words.

The function itself is straightforward. Build a PTSIN array containing the x and y coordinates for the pixel, build an INTIN array containing the color you want for the pixel, set Line-A's INTIN and PTSIN variables to point to your arrays, and perform the PutPixel call.

Example:

; Plot a pixel at (10,10) with color 1

dc.w	\$A000	; Make Line A Init call
move.l	*int,INTIN(a0)	; Address of INTIN array
move.l	*point,PTSIN(a0)	; Address of PTSIN array
dc.w	\$A001	; Put Pixel

.data

; Define the INTIN and PTSIN arrays.

int:	dc.w	1
point:	dc.w	10.10

Section 2: Line A Function Reference

\$A002 -- Get Pixel

Gets the value of a single pixel at the given X and Y coordinates. Returns this value in d0.

Input:

PTSIN[0] = X coordinate of pixel
PTSIN[1] = Y coordinate of pixel

Returns:

D0 = value of the pixel

Notes:

Code for using this function is very similar to that for PutPixel; Instead of putting a pixel on the screen, it returns the value of a pixel in d0.

Example:

```
-----  
: Return the value of pixel at  
: (10,10) in d0
```

```
dc.w    $A000           ; Init Line-A  
move.l  *point,PTSIN(a0) ; Address of PTSIN  
dc.w    $A002           ; Get Pixel
```

.data

```
-----  
: Define the PTSIN array...
```

```
point:  dc.w    10,10
```

Section 2: Line A Function Reference

\$A003 -- Arbitrary Line

Draws a line between the (X1,Y1) and (X2,Y2). The line can be vertical, horizontal, or diagonal. If you know the line is horizontal, Horizontal Line (\$A004) is slightly faster.

Input:

COLBIT0	= bit value for plane 0	+024 \$018	word
COLBIT1	= bit value for plane 1	+026 \$01A	word
COLBIT2	= bit value for plane 2	+028 \$01C	word
COLBIT3	= bit value for plane 3	+030 \$01E	word
LSTLIN	= Draw last pixel of line? (0=yes 1=no)	+032 \$020	word
LNMASK	= line style mask (line pattern)	+034 \$022	word
WMODE	= writing mode	+036 \$024	word
X1	= X1 coordinate	+038 \$026	word
Y1	= Y1 coordinate	+040 \$028	word
X2	= X2 coordinate	+042 \$02A	word
Y2	= Y2 coordinate	+044 \$02C	word

Side Effects:

LNMASK is rotated to align with the right-most endpoint.

Returns: Nothing.

Notes:

LNMASK is a one-word mask containing the pattern of the line. WMODE determines what mode a line is drawn in, replace, transparent, reverse transparent, or XOR mode.

LSTLIN determines if the last pixel of a line is drawn. If LSTLIN is nonzero, the last pixel will NOT be drawn. This helps prevent two connected lines from XORing a common endpoint out of existence.

Example:

```

-----
; draw a solid line from (0,0) to (100,100)

dc.w      $A000                ; Make Line A Init call
move.w    *1,COLBIT0(a0)       ; set COLBIT variables
move.w    *1,COLBIT1(a0)
move.w    *1,COLBIT2(a0)
move.w    *1,COLBIT3(a0)
move.w    *0,LSTLIN(a0)        ; draw last pixel of line
move.w    *$FFFF,LNMASK(a0)    ; line style mask
move.w    *0,WMODE(a0)         ; writing mode (replace)
move.w    *0,X1(a0)            ; (x1,y1) and (x2,y2) into
move.w    *0,Y1(a0)            ; appropriate variables
move.w    *100,X2(a0)
move.w    *100,Y2(a0)
dc.w      $A003                ; Arbitrary Line

```


Section 2: Line A Function Reference

\$A004 -- Horizontal Line

Draw a horizontal line between (X1,Y1) and (X2,Y1). Horizontal line is slightly faster than the Arbitrary Line function.

Input:

COLBIT0	= bit value for plane 0	+024 \$018	word
COLBIT1	= bit value for plane 1	+026 \$01A	word
COLBIT2	= bit value for plane 2	+028 \$01C	word
COLBIT3	= bit value for plane 3	+030 \$01E	word
WMODE	= Writing mode	+036 \$024	word
X1	= X1 coordinate	+038 \$026	word
Y1	= Y1 coordinate	+040 \$028	word
X2	= X2 coordinate	+042 \$02A	word
PATPTR	= Pointer to fill pattern	+046 \$02E	long
PATMSK	= Pattern index	+050 \$032	word
MFILL	= multi-plane pattern flag	+052 \$034	word

Returns: Nothing.

Notes:

PATPTR points to an array of line patterns.

The line pattern is chosen from the array of line patterns based on Y1 AND PATMSK. PATMSK should equal the number of line patterns in the array minus one.

If MFILL is nonzero, all planes will be filled with the values in the COLBITs. This overrides WMODE, since a multi-plane fill will happily perform a REPLACE of the destination bitplanes with no regard for the WMODE.

Example:

```

-----
; Draw a dashed line from (0,10) to (10,100)
dc.w      $A000                ; Line A Init
move.w    *1,COLBIT0(a0)      ; set COLBIT variables
move.w    *1,COLBIT1(a0)
move.w    *1,COLBIT2(a0)
move.w    *1,COLBIT3(a0)
move.w    *0,WMODE(a0)        ; writing mode (replace)
move.w    *0,X1(a0)           ; x1, y1, and x2 into
move.w    *10,Y1(a0)          ; appropriate variables
move.w    *100,X2(a0)
move.l    *pat,PATPTR(a0)     ; pattern pointer
move.w    *0,PATMSK(a0)       ; Pattern length n-1=0
move.w    *0,MFILL(a0)        ; Multiple Plane fill off
dc.w      $A004                ; Horizontal Line

.data
pat:      dc.w      $F0F0      ; Pattern for line.

```

Section 2: Line A Function Reference

\$A005 -- Filled Rectangle

Draw a filled rectangle with upper left corner at (X1,Y1), and lower right corner at (X2,Y2).

Input:

COLBIT0	=	bit value for plane 0	+024 \$018	word
COLBIT1	=	bit value for plane 1	+026 \$01A	word
COLBIT2	=	bit value for plane 2	+028 \$01C	word
COLBIT3	=	bit value for plane 3	+030 \$01E	word
WMODE	=	writing mode	+036 \$024	word
X1	=	X1 coordinate	+038 \$026	word
Y1	=	Y1 coordinate	+040 \$028	word
X2	=	X2 coordinate	+042 \$02A	word
Y2	=	Y2 coordinate	+044 \$02C	word
PATPTR	=	Pointer to the fill pattern	+046 \$02E	long
PATMSK	=	Fill pattern index	+050 \$032	word
MFILL	=	Multi-lane fill pattern flag	+052 \$034	word
CLIP	=	clipping flag	+054 \$036	word
XMINCL	=	X minimum for clipping	+056 \$038	word
XMAXCL	=	X maximum for clipping	+058 \$03A	word
YMINCL	=	Y minimum for clipping	+060 \$03C	word
YMAXCL	=	Y maximum for clipping	+062 \$03E	word

Returns: Nothing.

Example:

Section 2: Line A Function Reference

```

:-----
: Draw a filled rectangle with its upper
: left corner at (0,0) and its lower right
: corner at (100,100). Clip the rectangle
: to within (0,0) and (50,50).

```

```

dc.w      $A000          ; Line A init
move.w    *1,COLBIT0(a0) ; bit planes for shape
move.w    *1,COLBIT1(a0)
move.w    *1,COLBIT2(a0)
move.w    *1,COLBIT3(a0)
move.w    *0,WMODE(a0)   ; writing mode (replace)
move.w    *0,X1(a0)      ; X and Y values for shape
move.w    *0,Y1(a0)
move.w    *100,X2(a0)
move.w    *100,Y2(a0)
move.l    *fuji,PATPTR(a0) ; pattern pointer to fuji
move.w    *15,PATMSK(a0) ; Pattern length = 16-1 = 15
move.w    *0,MFILL(-?)   ; multi-plane fill (off)
move.w    *1,CLIP(a0)    ; clipping status (on)
move.w    *0,XMINCL(a0)  ; clipping boundaries
move.w    *50,XMAXCL(a0)
move.w    *0,YMINCL(a0)
move.w    *50,YMAXCL(a0)
dc.w      $A005          ; Filled Rectangle

```

```

.data
fuji:

```

```

dc.w      $0000          : 0000000000000000
dc.w      $05A0          : 0000010110100000
dc.w      $05A0          : 0000010110100000
dc.w      $05A0          : 0000010110100000
dc.w      $05A0          : 0000010110100000
dc.w      $0DB0          : 0000110110110000
dc.w      $0DB0          : 0000110110110000
dc.w      $1DB8          : 0001110110111000
dc.w      $399C          : 0011100110011100
dc.w      $799E          : 0111100110011110
dc.w      $718E          : 0111000110001110
dc.w      $718E          : 0111000110001110
dc.w      $6186          : 0110000110000110
dc.w      $4182          : 0100000110000010
dc.w      $0000          : 0000000000000000
dc.w      $0000          : 0000000000000000

```

Section 2: Line A Function Reference

\$A006 -- Filled Polygon

Draws a filled polygon line-by-line.

Input:

PTSIN[]	=	Pointer to an array of polygon vertices. ((x1,y1),(x2,y2)...(xn,yn),(x1,y1))		
CONTRL[1]	=	n = number of vertices		
COLBIT0	=	bit value for plane 0	+024 \$018	word
COLBIT1	=	bit value for plane 1	+026 \$01A	word
COLBIT2	=	bit value for plane 2	+028 \$01C	word
COLBIT3	=	bit value for plane 3	+030 \$01E	word
WMODE	=	writing mode	+036 \$024	word
Y1	=	y coordinate of scan-line to fill	+040 \$028	word
PATPTR	=	Pointer to the fill pattern	+046 \$02E	long
PATMSK	=	Fill pattern index	+050 \$032	word
MFILL	=	Multi-plane fill pattern flag	+052 \$034	word
CLIP	=	clipping flag	+054 \$036	word
XMINCL	=	X minimum for clipping	+056 \$038	word
XMAXCL	=	X maximum for clipping	+058 \$03A	word
YMINCL	=	Y minimum for clipping	+060 \$03C	word
YMAXCL	=	Y maximum for clipping	+062 \$03E	word

Side Effects:

A0 is destroyed.
X1 and X2 are destroyed.

Returns: Nothing.

Notes:

The first vertex must be repeated at the end of the list of n endpoints.

Filled polygon requires CONTRL to point to an array in which the second word contains the number of vertices. This is handled in the example program by the line "control: dc.w 0,NUMVERTS".

PTSIN must point to an array of vertices in the format X1, Y1, X2, Y2, X3, Y3, and so on. Each coordinate is a word in length. In the example, this is handled by the array starting at "verts:".

The polygon is drawn one line at a time. The Y coordinate is contained in Y1. To fill an entire polygon, a simple loop increments Y and performs the Filled Polygon call repeatedly. In the example, this is done by a routine called "loop".

Section 2: Line A Function Reference

Example:

```

-----
; Draw a polygon with vertices at (0,0),
; (319,120), and (25,199). One line is
; drawn at a time. The clipping variables
; are set but not evaluated, since CLIP
; is set to 0 (off).

```

```

TOP      equ      0          ; Uppermost Y in polygon
BOTTOM   equ      199       ; Lowermost Y in polygon
NUMVERTS equ      3        ; number of vertices
PATLENGTH equ     15       ; length of pattern

        dc.w      $A000     ; Line-A Init
        move.l    *verts,PTSIN(a0) ; Address of verts
        move.l    *control,CONTRL(a0) ; Address of "control"
        move.w    *1,COLBIT0(a0)
        move.w    *1,COLBIT1(a0)
        move.w    *1,COLBIT2(a0)
        move.w    *1,COLBIT3(a0)
        move.w    *0,WMODE(a0) ; writing mode (replace)
        move.l    *fuji,PATPTR(a0) ; address of pattern
        move.w    *PATLENGTH,PATMSK(a0) ; length of fill pattern
        move.w    *0,MFILL(a0) ; Multi-plane fill (off)
        move.w    *0,CLIP(a0) ; clipping status (off)
        move.w    *0,XMINCL(a0) ; clipping boundaries
        move.w    *100,XMAXCL(a0)
        move.w    *0,YMINCL(a0)
        move.w    *100,YMAXCL(a0)

```

```

-----
; Main loop to draw the polygon

```

```

        move.l    a0,a5 ; Save a0 in a5.
        move.w    *TOP,Y1(a5) ; Maximum y to Y1
        move.w    *BOTTOM,d4 ; Maximum y into d4
        sub.w     *TOP,d4 ; minus minimum y
loop:    dc.w      $A006 ; Filled Polygon (one line)
        addq     *1,Y1(a5) ; Decrement current Y
        dbra     d4,loop

```

Section 2: Line A Function Reference

.data			
control:	dc.w	0,NUMVERTS	; CONTRL[1] = no. of verts
verts:	dc.w	0,0,319,120,25,199,0,0	
fuji:	dc.w	\$0000	: 0000000000000000
	dc.w	\$05A0	: 0000010110100000
	dc.w	\$05A0	: 0000010110100000
	dc.w	\$05A0	: 0000010110100000
	dc.w	\$05A0	: 0000010110100000
	dc.w	\$0DB0	: 0000110110110000
	dc.w	\$0DB0	: 0000110110110000
	dc.w	\$1DB8	: 0001110110111000
	dc.w	\$399C	: 0011100110011100
	dc.w	\$799E	: 0111100110011110
	dc.w	\$718E	: 0111000110001110
	dc.w	\$718E	: 0111000110001110
	dc.w	\$6186	: 0110000110000110
	dc.w	\$4182	: 0100000110000010
	dc.w	\$0000	: 0000000000000000
	dc.w	\$0000	: 0000000000000000

Section 2: Line A Function Reference

\$A007 -- BitBlit

Perform a BIT BLock Transfer

Input:

a6 = pointer to the BitBlit parameter block

The BitBlit routines receive information through their own parameter block, the address of which must be put into a6 before the bitblt call is made. The format of this block is shown below. This block must be 76 bytes long, including 24 bytes at the end for use by the Blt. Variables marked with a (D) may be destroyed during the blit.

B_WD	+00 (\$00)	(word)	Width of block to blit (in pixels)
B_HT	+02 (\$02)	(word)	Height of block to blit (in pixels)
PLANE_CT	+04 (\$04)	(word)	Number of consecutive planes to blit (D)
FG_COL	+06 (\$06)	(word)	Foreground color (logic op index:hi bit) (D)
BG_COL	+08 (\$08)	(word)	Background color (logic op index:lo bit) (D)
OP_TAB	+10 (\$0A)	(long)	Logic ops for all fore and background combos
S_XMIN	+14 (\$0E)	(word)	Minimum X: source
S_YMIN	+16 (\$10)	(word)	Minimum Y: source
S_FORM	+18 (\$12)	(long)	Source form base address
S_NXWD	+22 (\$16)	(word)	Offset to next word in line (in bytes)
S_NXLN	+24 (\$18)	(word)	Offset to next line in plane (in bytes)
S_NXPL	+26 (\$1A)	(word)	Offset from start of current plane to next plane
D_XMIN	+28 (\$1C)	(word)	Minimum X: destination
D_YMIN	+30 (\$1E)	(word)	Minimum Y: destination
D_FORM	+32 (\$20)	(long)	Destination form base address. (For example, Physbase of the screen.)
D_NXWD	+36 (\$24)	(word)	Offset to next word in line (in bytes)
D_NXLN	+38 (\$26)	(word)	Offset to next line in plane (in bytes)
D_NXPL	+40 (\$28)	(word)	Offset from start of current plane to next plane.
P_ADDR	+42 (\$2A)	(long)	Address of pattern buffer (0=no pattern)

Section 2: Line A Function Reference

P_NXLN	+46 (\$2E)	(word)	Offset to next line in pattern (in bytes)
P_NXPL	+48 (\$30)	(word)	Offset to next plane in pattern (in bytes)
P_MASK	+50 (\$32)	(word)	Pattern index mask
SPACE	+52 (\$34)	24 bytes	Extra Space, required by the blit. Be sure to define this or the next 24 bytes of memory will be clobbered, resulting in a mangled image or worse!

S_FORM and D_FORM point to the first words of the source memory form and destination memory forms, respectively. These addresses must be on word boundaries.

S_NXWD and D_NXWD are offsets to the next word in a plane of the memory form. For example, in a monochrome mode screen the value is 2, in medium resolution, 4, and in low resolution, 8.

S_NXLN and D_NXLN are form widths for source and destination. These widths must be even byte values, since they represent the offset from one row of the form to the next and forms must be word aligned and an integral number of words wide. (hint: The hi rez screen value is 90 while low and medium rez values are 160.)

S_NXPL and D_NXPL are offsets from the start of one plane to the start of the next plane. Because of the ST screen's interleaved plane structure, this value is always two. Alternative universes allow for a series of contiguous planes where NXPL values are the number of bytes in each plane. Thus, it is possible to BLT from the contiguous universe into the interleaved ST universe and vice versa.

The actual bit aligned blocks of memory are defined within the form by an upper left anchor point, a pixel width, and a pixel height: (S_XMIN, S_YMIN, B_WD, and B_HT). The location in the destination form is defined by an anchor point (D_XMIN, D_YMIN). No harm will come if these two areas overlap. Note that no clipping is performed and there is no checking to determine whether the bit blocks fall within the confines of the encompassing memory forms. Finally, the number of planes to be transferred (the number of iterations of the BLT algorithm) is contained in the PLANE_CT word.

OP_TAB is a table of four RASTER OP codes. Each of the byte wide entries in OP_TAB contain a code for one of the sixteen logical operations between source and destination blocks. For each plane, the logical operation is chosen by indexing into the OP_TAB with a value derived from the FG_COL and BG_COL words. For a given plane "n", bit "n" of FG_COL is the hi bit of the two bit index value and bit "n" of BG_COL is the lo bit of the index value:

<u>FG(n)</u>	<u>BG(n)</u>	<u>OP_TAB entry</u>
0	0	first byte
0	1	second byte
1	0	third byte
1	1	fourth byte

For each unique combination of FG and BG, a specific logic operation can be defined with OP_TAB.

BitBit Logic Ops

S = Source pixel
D = Destination pixel
D' = Destination after operation

Section 2: Line A Function Reference

OP	Combination Rule
0	D' = 0
1	D' = S AND D
2	D' = S AND [NOT D]
3	D' = S (Replace Mode)
4	D' = [NOT S] AND D (Erase Mode)
5	D' = D
6	D' = S XOR D (XOR mode)
7	D' = S OR D
8	D' = NOT [S OR D]
9	D' = NOT [S XOR D]
A	D' = NOT D
B	D' = S OR [NOT D]
C	D' = NOT S
D	D' = [NOT S] OR D
E	D' = NOT [S AND D]
F	D' = 1

Patterns

Patterns are word-wide, word-aligned images that are logically ANDed with the source prior to the logical combination of source and destination.

Patterns are packed in an imaginary grid anchored at the upper left corner (0,0) of the destination memory form.

Patterns are 16 bits wide and repeated every 16 pixels horizontally.

Patterns are an integral power of 2 in height and repeat vertically at that frequency. (1,2,4,8,...)

The source is shifted into alignment with the destination rectangle prior to the combination of source with pattern. Thus, the relationship between source and pattern is dependent upon the X,Y positioning of the destination rectangle.

P_ADDR points to the first word of the pattern. If this pointer is 0, a pattern is not combined with the source rectangle.

P_NXLN is the offset (in bytes) between consecutive words in the pattern. This number should be an integral power of two (2, 4, 8...)

P_NXPL is the offset (in bytes) from the beginning of a plane to the beginning of the next plane. In the case of a single plane pattern used in a multi-plane environment, this value would be zero. Thus, the same pattern is repeated through all planes.

P_MASK works with P_NXLN to specify the length of the pattern. The length (in words) of the pattern must be an integral power of two.

$$\text{if } P_NXLN = 2^n \\ \text{then } P_MASK = (\text{length in words} - 1) \ll n$$

To BLT from a single plane source to multi-plane destination, S_NXPL = 0. The same source plane is BLT'd to all destination planes. To map 1s to foreground color and 0s to background color, set OP_TAB to:

Section 2: Line A Function Reference

<u>Offset</u>	<u>Logic Op</u>	
+00	00	All zeros
+01	04	D' <- [NOT S] AND D
+02	07	D' <- S OR D
+03	15	All ones

Load foreground color into FG_COL and background color into BG_COL.

To map 1s to foreground color and make 0s transparent set OP_TAB to:

<u>Offset</u>	<u>Logic Op</u>	
+00	04	D' <- [NOT S] AND D
+01	04	D' <- [NOT S] AND D
+02	07	D' <- S OR D
+03	07	D' <- S OR D

Load foreground color into FG_COL; BG_COL is irrelevant. Be sure S_NXPL is set to 0.

To BLT a pattern without Source to the Destination, define a word of ones, and set S_FORM at it. Set S_NXLN, S_NXPL, S_NXWD, S_XMIN, and S_YMIN to 0. Set up the pattern as you usually would. The BLT will create a pattern-filled rectangle.

To make a simple sprite-like device, build a monoplane mask. Everywhere there is a 1 in the mask, the background will be removed. Wherever a 1 falls, the background is left intact. Set OP_TAB to:

<u>Offset</u>	<u>Logic Op</u>	
+00	04	D' <- [NOT S] AND D
+01	04	D' <- [NOT S] AND D
+02	07	D' <- S OR D
+03	07	D' <- S OR D

Load foreground color into FG_COL; BG_COL is irrelevant.

Take a monoplane form (or multi-plane form) and "OR" it (OP 7) into the area that you just scooped out with the mask.

Example:

Section 2: Line A Function Reference

```
-----
: BitBlt a monochrome invertibrate
: to the screen.
```

```

    move.w    *2,-(sp)           ; get screen base address
    trap     *14
    addq     *2,sp
    move.l   d0,screen
    lea     blit,a6             ; address of blit parameter block
    dc.w    $A007               ; BitBlt
```

```
.data
```

```
-----
: BitBlt Parameter Block
```

```

blit:    dc.w    $0030           ; width of source in pixels
         dc.w    $0014           ; height of source in pixels
         dc.w    $0001           ; number of consecutive planes to blit
         dc.w    $0001           ; fg color (logic op index: hi bit)
         dc.w    $0000           ; bg color (logic op index: lo bit)
         dc.l    $07070707       ; logic ops for all fg and bg combos
         dc.w    $0000           ; minimum X: source
         dc.w    $0000           ; minimum Y: source
         dc.l    slug            ; source form base address
         dc.w    $0002           ; byte offset to next word in line
         dc.w    $0006           ; byte offset to next line in plane
         dc.w    $0002           ; offset to next plane (in bytes)
         dc.w    $00FF           ; minimum X: destination
         dc.w    $0064           ; minimum Y: destination
screen:  dc.l    $00000000       ; destination form base address
         dc.w    $0002           ; byte offset to next word in line
         dc.w    $0050           ; byte offset to next line in plane
         dc.w    $0002           ; offset to next plane (in bytes)
         dc.l    $00000000       ; address of pattern buffer
         ; (0=no pattern)
         dc.w    $0000           ; byte offset to next line in pattern
         dc.w    $0000           ; byte offset to next plane in pattern
         dc.w    $0000           ; pattern index mask

         dc.w    $0000, $0000, $0000, $0000
         dc.w    $0000, $0000, $0000, $0000
         dc.w    $0000, $0000, $0000, $0000
```

Section 2: Line A Function Reference

Image definition from:
NEOchrome cut buffer contents (left justified):

pixels/scanline = \$0030 (bytes/scanline: \$0006)
* scanlines (height) = \$0014
Monochrome mask (1 plane; background=0/non-background=1)

slug:	dc.w	\$0000,	\$0000,	\$0030,	\$0000,	\$0000,	\$0066,	\$0000,	\$0000
	dc.w	\$006C,	\$0000,	\$0000,	\$00CE,	\$0000,	\$0000,	\$00CC,	\$0000
	dc.w	\$0000,	\$0198,	\$0000,	\$0000,	\$03B0,	\$0000,	\$0000,	\$0770
	dc.w	\$0000,	\$0000,	\$0760,	\$0000,	\$0000,	\$0EE0,	\$0000,	\$0000
	dc.w	\$7FC0,	\$0000,	\$0003,	\$FFC0,	\$0000,	\$003F,	\$FFC0,	\$0000
	dc.w	\$00FF,	\$FFE0,	\$0000,	\$1FFF,	\$FFF0,	\$01FF,	\$FFFF,	\$FEF0
	dc.w	\$0FFF,	\$FFFF,	\$FF70,	\$1FFF,	\$FFFF,	\$FF80,	\$FFFF,	\$FFFF
	dc.w	\$FFE0,	\$FFFF,	\$FFFF,	\$FFC0				

Note: This example might not be admissable in a programming class, since it changes some of its "dc.w"s. In the real world, you'd probably want to copy all this into your bss, then make the changes.

Section 2: Line A Function Reference

\$A008 -- TextBlit

Perform a TEXT BLock Transfer of 1 character

Input:

WMODE	= Writing mode (0-3 =>VDI modes, 4-19 =>BitBlit modes)	+036 \$024	word
CLIP	= clipping flag	+054 \$036	word
XMINCL	= X minimum for clipping	+056 \$038	word
YMINCL	= Y minimum for clipping	+058 \$03A	word
XMAXCL	= X maximum for clipping	+060 \$03C	word
YMAXCL	= Y maximum for clipping	+062 \$03E	word
XDDA	= accumulator for x dda	+064 \$040	word
DDAINC	= fractional amount to scale up or down	+066 \$042	word
SCALDIR	= scale direction flag (0=down)	+068 \$044	word
MONO	= mono spaced font flag	+070 \$046	word
SOURCEX	= x coord of character (in font form)	+072 \$048	word
SOURCEY	= y coord of character (in font form)	+074 \$04A	word
DESTX	= x coord of character (in destination form)	+076 \$04C	word
DESTY	= y coord of character (in destination form)	+078 \$04E	word
DELX	= width of character	+080 \$050	word
DELY	= height of character	+082 \$052	word
FBASE	= Pointer to start of font data (font form)	+084 \$054	long
FWIDTH	= Width of font form	+088 \$058	word
STYLE	= TextBlit special effects flags	+090 \$05A	word
LITEMASK	= mask for lightening text	+092 \$05C	word
SKEWMASK	= mask for skewing text	+094 \$05E	word
WEIGHT	= width by which to thicken text	+096 \$060	word
ROFF	= offset above character baseline when skewing	+098 \$062	word
LOFF	= offset below character baseline when skewing	+100 \$064	word
SCALE	= scaling flag (0 => no scaling)	+102 \$066	word
CHUP	= character rotation vector	+104 \$068	word
TEXTFG	= Text foreground color	+106 \$06A	word
SCRTPHP	= pointer to start of text special effects buffer	+108 \$06C	long
SCRPT2	= offset of scaling buffer in SCRTPHP buffer (midpoint)	+112 \$070	word
TEXTBG	= Text background color	+114 \$072	word

Notes:

Most of the effort for TextBlit goes into setting up its variables, as shown above. The information you need about the font itself is contained in the font header, as described in the VDI manual under "Font Format". Not all of the variables are always evaluated, as the example shows. Check Section 3, "The Line A Variable Structure," for more information on the TextBlit variables and their uses.

After TextBlit outputs one character, it automatically increments its X coordinate by the width of the character printed.

Section 2: Line A Function Reference

TextBit allows the four VDI writing modes, as well as the BitBit modes. VDI modes 1-4 are TextBit 0-3, and BitBit modes 0-15 are TextBit modes 4-19.

When using special effects, make sure the buffer pointed to by SCRTCHP is large enough to contain the worst case (largest) result of the effects * 2. SCRPT2 must be an offset from the beginning to the midpoint of this buffer.

Example:

: Font Header Offsets

```
first_ade equ 36 ; header offset to value of first displayable
                ; character in font
off_table equ 72 ; header offset to pointer to offset table
data_table equ 76 ; header offset to pointer to font data
form_width equ 80 ; header offset to total width of font
form_height equ 82 ; header offset to total height of font
```

: Print a null-terminated string using
: TextBit

```
dc.w $A000 ; Line A Init
move.w *2,WMODE(a0) ; writing mode (VDI XOR)
move.w *0,CLIP(a0) ; Clipping status (off)
move.w *0,XMINCL(a0) ; clipping boundaries
move.w *125,XMAXCL(a0)
move.w *0,YMINCL(a0)
move.w *200,YMAXCL(a0)
move.w *1,TEXTFG(a0) ; Foreground color
move.w *0,TEXTBG(a0) ; Background color
move.w *100,DSTX(a0)
move.w *100,DSTY(a0)
move.w *4,STYLE(a0)
move.w *0,SCALE(a0)
move.w *1,MONO(a0)
```

: Find the system fonts

```
move.l 4(a1),a1 ; Address of 8x8 font
move.l data_table(a1),FBASE(a0) ; Address of font data
move.w form_width(a1),FWIDTH(a0) ; font form width
move.w form_height(a1),DELY(a0) ; height of font
```

: Print the string

Section 2: Line A Function Reference

```
print:    lea.l    string,a2                ; a2 -> string to print
          move.l  off_table(a1),a3    ; address of offset table
          clr.l   d0                   ; make sure d0 is clear
          move.b  (a2)+,d0             ; character from string
          ble     dienow              ; end of string, exit
          sub.w   first_ade(a1),d0    ; letter's offset in font
          lsl.w   *1,d0               ; x2 for _word_ offset in
          ; offset table
          move.w  0(a3,d0),SRCX(a0)   ; x of desired character
          move.w  2(a3,d0),d0         ; x of next character
          sub.w   SRCX(a0),d0         ; minus x of desired char
          move.w  d0,DELX(a0)         ; width of desired char
          clr.w   SRCY(a0)            ; start at top of char
          movem.l a0-a2,-(a7)         ; push everything on the stack
          dc.w   $A008                ; TextBlit
          movem.l (a7)+,a0-a2        ; put everything back
          bra     print               ; print next character

dienow:   rts

.data
string:   dc.b    "Welcome to TextBlit, Space Guy!",0
```

Section 2: Line A Function Reference

\$A009 -- Show Mouse

Show the mouse cursor.

Input:

INTIN[0] (optional, see Notes, below.)

Returns: Nothing.

Useful Variables:

The depth at which the mouse cursor is hidden is held in HIDE_CNT at offset -598 (-\$256). This variable will be zero if the mouse is shown, and non-zero if hidden. The number is how many "shows" must be performed to show the mouse cursor.

The x and y coordinates of the mouse cursor are held in GCURX and GCURY, at -602 (-\$25A) and -600 (-\$258).

The mouse button status is held in MOUSE_BT at -596 (-\$254). See Section 3, "The Line A Variable Structure", for more information.

Notes:

If Hide Mouse has been used more than once, an equivalent number of Show Mouse calls must be made to be effective. To force the mouse cursor to be shown regardless of how many hides have occurred, put a word of zero into INTIN[0].

Example:

```
dc.w    $A009
```


Section 2: Line A Function Reference

\$A00A -- Hide Mouse

Hide the mouse cursor.

Input: None.

Returns: Nothing.

Notes:

If you use more than one Hide Mouse, it must be countered with an equivalent number of Show Mouse calls to show again. This is explained in "Show Mouse", above.

Example:

dc.w \$A00A

Section 2: Line A Function Reference

\$A00B -- Transform Mouse

Transform the mouse's form.

Input:

INTIN = pointer to an array of parameters

Returns: Nothing.

Notes:

This function gets its parameters and data from an array pointed to by INTIN. This array contains information like the "hot spots" for the mouse pointer, colors for the new mouse pointer, and the actual shape of the new mouse pointer.

The existing mouse pointer information is contained in the Line A Variable Structure, starting at -856 (-\$356). This information can be saved away before you change the mouse cursor and be used to restore it to its former self.

Also, mouse_flag at -339 (-\$153) determines if the mouse interrupts are enabled, and can be used to prevent the mouse cursor from being updated while changing its form. (Be sure to restore mouse_flag to its previous value when you're done.)

Example:

: Replace the familiar arrow with a
: short message.

```
HOTX    equ    0           ; Mouse hot-spots
HOTY    equ    0
MASKC   equ    0           ; color data
DATAC   equ    1

        dc.w    $A000      ; Init Line A
        move.l  *mouse,INTIN(a0) ; address of mouse data
        dc.w    $A00A      ; Hide Mouse
        dc.w    $A00B      ; Transform Mouse
        dc.w    $A009      ; Show Mouse
```

: mouse data

```
mouse:  dc.w    HOTX        ; x hot spot
        dc.w    HOTY        ; y hot spot
        dc.w    1           ; Reserved, must be 1...
        dc.w    MASKC       ; Mask color index
        dc.w    DATAC       ; Data color index
        dc.w    $0000,$0000,$0000,$0000,$FFFF,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$0000,$0000,$0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000,$0002,$632A,$50AA,$5798
        dc.w    $638A,$4030,$0000,$0000,$0000,$0000,$0000,$0000
```

Section 2: Line A Function Reference

\$A00C -- Undraw Sprite

Undraw the previously drawn sprite.

Sprites are useful for animating small objects, since Line-A takes care of the housekeeping for you. Sprites are 16x16, and consist of two "layers": an image and a mask.

When a sprite is drawn, the screen image "under" it is copied into the sprite save block. When that sprite is undrawn, the screen is restored to its original state.

When using multiple sprites, undraw in reverse order of drawing. If any one sprite intersected another, it will have copied part of the underlying sprite away into the sprite save block. If you undraw in order, the underlying sprite will be restored to background, erasing the "top" sprite. When the top sprite is undrawn, it will restore a part of the underlying sprite. This causes what is called (in computer graphics) a "mess".

Input:

A2 = Pointer to sprite save block

Side Effects:

A6 is destroyed.

Notes:

The sprite save block is used to save the screen underneath the sprite. Its size is 10 bytes + 64 bytes per plane: $(10 + (VPLANES * 64))$ bytes.

Example:

See draw sprite, below.

Section 2: Line A Function Reference

\$A00D -- Draw Sprite

Draw a software sprite.

Input:

D1 = Y hot_spot
A0 = pointer to sprite definition block
A2 = pointer to sprite save block

Side Effects:

A6 is destroyed.

Returns: Nothing.

Notes:

The sprite save block is used to save the screen underneath the sprite. Its size is 10 bytes + 64 bytes per plane: $(10 + (VPLANES * 64))$ bytes.

The Sprite Definition Block is laid out as follows:

<u>Offset</u>	<u>Size</u>	<u>Description</u>
000 \$000	word	X offset of sprite hot-spot
002 \$002	word	Y offset of sprite hot-spot
004 \$004	word	Format flag (see below)
006 \$006	word	Background color (Physical pixel color)
008 \$008	word	Foreground color (Physical pixel color)
010 \$00A	64 bytes	Sprite image.

The format flag determines how the sprite will be drawn. There are two modes, VDI and XOR. If the format flag is 1, VDI format is used, if -1, XOR is used. The two modes are compared below.

	<u>FG bit</u>	<u>BG bit</u>	<u>Action</u>
VDI Mode	0	0	Destination (screen) color
	0	1	Background color plotted
	1	0	Foreground color plotted
	1	1	Foreground color plotted
XOR Mode	0	0	Destination (screen) color
	0	1	Background color plotted
	1	0	Invert destination (screen) color
	1	1	Foreground color plotted

The sprite image is designated as alternating words of background and foreground image, like:

word 0 = background line 0
word 1 = foreground line 0
word 2 = background line 1
word 3 = foreground line 1

Example:

Section 2: Line A Function Reference

```

GCURX   equ    -602           ; Current mouse X position
GCURY   equ    -600           ; Current mouse Y position
MOUSE_BT equ    -596           ; Mouse button status

```

```

loop:   move.w  *37,-(sp)      ; Wait for VSYNC
        trap   *14
        addq   *2,sp
        dc.w   $A00A           ; Hide Mouse
        lea   save,a2         ; image save area
        dc.w   $A00C           ; Undraw Sprite

```

; Draw a sprite tied to the mouse position

```

draw:   dc.w    $A000           ; Init Line-A
        move.w  GCURX(a0),d0    ; x position
        move.w  GCURY(a0),d1    ; y position
        lea    sprite,a0        ; sprite image data
        lea    save,a2         ; image save area
        dc.w   $A00D           ; Draw Sprite
        dc.w   $A000           ; Init Line-A
        move.w  MOUSE_BT(a0),d3 ; Mouse button status
        bst    d3,*1           ; Check right button
        bne    loop           ; If not, loop
        dc.w   $A009           ; Show Mouse
        rts

```

.bss

; Sprite save block

```

save:   ds.w    5               ; Storage for misc. info
        ds.w    32             ; Storage for sprite image

```

.data

; Sprite data

```

sprite: dc.w    0               ; x offset of hot spot
        dc.w    0               ; y offset of hot spot
        dc.w    1               ; format flag
        dc.w    0               ; background color
        dc.w    1               ; foreground color
        dc.w    %0000111111111000,%0000011111110000
        dc.w    %0001111111111100,%0000111111111000
        dc.w    %0011111111111110,%0001111111110100
        dc.w    %0011111111111110,%0001100000000100
        dc.w    %0011111111111110,%0001100000000100
        dc.w    %0011111111111110,%0001000000000100
        dc.w    %0011111111111110,%0001111000111100
        dc.w    %0011111111111110,%0001011101010100
        dc.w    %0011111111111110,%0001000100000100
        dc.w    %0001111111111100,%0000101100101000
        dc.w    %0001111111111100,%0000110111011000
        dc.w    %0000111111111100,%0000011000101000
        dc.w    %0000111111111000,%0000011111010000
        dc.w    %0111111111111000,%0010111000010000
        dc.w    %0111111111110000,%0011100111100000
        dc.w    %0111110000000000,%0011100000000000

```

Section 2: Line A Function Reference

Section 2: Line A Function Reference

\$A00E -- Copy Raster

Same as VDI's Copy Raster functions, but with the Line-A call you needn't open a virtual workstation.
See VDI manual under "Raster Operations"

Section 2: Line A Function Reference

\$A00F -- Seedfill

Same as VDI's Contour Fill function, with the following exceptions:

You needn't open a virtual workstation

You **MUST** set the clipping variables correctly. They are evaluated regardless of the state of the clipping flag.

SEEDABORT is a vector to a routine called at the end of each line fill. Seedfill aborts or continues based on the value returned in D0; if zero, it continues, if nonzero, it aborts.

Section 3: The Line A Variable Structure

The following is a chart of the Line A Input Variables Structure. It shows the name of the variable, its offset from the beginning of the table (in decimal and hex), its size, and a brief description of its function. The top of this chart is lower in memory than the bottom. Note: variables that begin with "V_" are used by the ST BIOS character output routines.

<u>NAME</u>	<u>OFFSET</u>	<u>SIZE</u>	<u>DESCRIPTION</u>
	-910 -\$38E to -906 -\$38A		RESERVED
CUR_FONT	-906 -\$38A	long	pointer to current font header
	-902 -\$386 to -856 -\$356		RESERVED
The next 37 words contain mouse cursor information, including the mask, form, hot spot, and writing mode.			
M_POS_HX	-856 -\$356	word	Mouse hot spot x coordinate within the 16x16 mouse cursor
M_POS_HY	-854 -\$354	word	Mouse hot spot y coordinate within the 16x16 mouse cursor
M_PLANES	-852 -\$352	word	Writing mode for mouse cursor.

1 indicates "normal" mode, -1 indicates XOR mode. There are two "planes" of information in the mouse cursor, representing the F(oreground) and B(ackground). The table below shows the displayed result for the four possible combinations in these "planes" for both the "normal" and "XOR" modes.

<u>F</u>	<u>B</u>	<u>Norm</u>	<u>XOR</u>	
0	0	Dest.	Dest.	Destination color is unchanged
0	1	B	B	"Background" mouse color shown
1	0	F	NOT Dest.	"Foreground" color shown, or destination color is inverted
1	1	F	F	"Foreground" color shown

M_CDB_BG	-850 -\$350	word	Mouse background physical pixel color
M_CDB_FG	-848 -\$34E	word	Mouse foreground physical pixel color
MASK_FORM	-846 -\$34C		Location of system mouse cursor mask and form.

Alternating words of background and foreground data, like: background word 0, foreground word 0...background word 15, foreground word 15.

INQ_TAB	-782 -\$30E	words	45 words, containing the information returned by the vq_extnd VDI call. (See VDI manual.)
---------	-------------	-------	---

Section 3: The Line A Variable Structure

DEV_TAB	-692 -\$2B4	words	45 words, containing the information returned by the v_opnwk VDI call. (See VDI manual.)
GCURX	-602 -\$25A	word	Current mouse cursor x position
GCURY	-600 -\$258	word	Current mouse cursor y position
M_HID_CT	-598 -\$256	word	Depth at which the mouse cursor is currently "hidden".

When the mouse cursor is hidden, this variable contains a non-zero value. An application can check this location to determine how deep the cursor is hidden. An application can also force the mouse cursor to be shown regardless of how deep it is hidden via the "Show Mouse" call.

MOUSE_BT	-596 -\$254	word	Current mouse button status
----------	-------------	------	-----------------------------

Bit 0 = left button status (0=up, 1=down)

Bit 1 = right button status (0=up, 1=down)

One way to check the mouse button status. Another is CUR_MS_STAT.

REQ_COL	-594 -\$252	words	3*16 words of internal data for vq_color (See VDI manual.)
SIZ_TAB	-498 -\$1F2	words	15 words, containing text, line, and marker sizes in device coordinates: 0 min char width 1 min char height 2 max char width 3 max char height 4 min line width 5 reserved 6 max line width 7 reserved 8 min marker width 9 min marker height 10 max marker width 11 max marker height 12-14 RESERVED
	-468 -\$1D4	word	RESERVED
	-466 -\$1D2	word	RESERVED
CUR_WORK	-464 -\$1D0	long	Pointer to current virtual workstation attributes
DEF_FONT	-460 -\$1CC	long	Pointer to default font header
FONT_RING	-456 -\$1C8	longs	

Section 3: The Line A Variable Structure

FONT_RING is an array of four longword pointers to linked lists of font headers. The first entry is the head pointer to the font list, the second and third are continuation fields, and the fourth is a null terminator. When the VDI searches through the list and encounters a null pointer in the link field of a font header, it continues the search from the next continuation field in FONT_RING. If this field is zero, the search ends. The first two pointers in FONT_RING are initialized for resident font lists, and the third is the pointer to the GDOS fonts, which is normally reinitialized during each VDI call. FONT_RING[3] is always zero to end VDI's quest for fonts.

When an application requests a specific font size and type, the system searches its lists for the first occurrence of the requested style. When found, VDI searches for the correct height. This search is terminated when VDI encounters another style in the header, or when a zero is found in FONT_RING. All fonts of the same style must be linked together in ascending order.

The first font header in a set of user installed fonts should be pointed to by FONT_RING[0], and the link field in the header of the last user-installed font should contain the pointer it finds in FONT_RING[0].

FONT_COUNT	-440 -\$1B8	word	Number of fonts in the FONT_RING lists
	-438 -\$1B6 to -348 -\$15C		RESERVED
CUR_MS_STAT	-348 -\$15C	byte	Mouse status
			Bit 0 = left mouse button status (0=up, 1=down) Bit 1 = right mouse button status (0=up, 1=down) Bit 2 = reserved Bit 3 = reserved Bit 4 = reserved Bit 5 = mouse movement flag (0=no movement, 1=movement) Bit 6 = right mouse button change flag (0=no change, 1=change) Bit 7 = left mouse button change flag (0=no change, 1=change)
	-347 -\$15B	byte	RESERVED
V_HID_CNT	-346 -\$15A	word	Hide depth of alpha cursor
CUR_X	-344 -\$158	word	Mouse cursor X position
CUR_Y	-342 -\$156	word	Mouse cursor Y position
CUR_FLAG	-340 -\$154	byte	Nonzero = draw mouse form on VBLANK.

Section 3: The Line A Variable Structure

CUR_X, CUR_Y, and CUR_FLAG make up a Communication block to the VBLANK mouse cursor draw routines. The X and Y at which the mouse cursor will be drawn are followed by a flag indicating if the mouse cursor should be drawn on the next VBLANK.

MOUSE_FLAG	-339 -\$153	byte	Non-zero if mouse interrupt processing is enabled
	-338 -\$152	long	RESERVED
V_SAV_XY	-334 -\$14E	word	Saved alpha cursor X coordinate
	-332 -\$14C	word	Saved alpha cursor Y coordinate
SAVE_LEN	-330 -\$14A	word	height of saved form (number of lines saved from screen)
SAVE_ADDR	-328 -\$148	long	Screen address of first word saved from screen
SAVE_STAT	-324 -\$144	word	Save Status
	bit 0 =>		1 = info in buffer is valid. 0 = info in buffer is not valid.
	bit 1 =>		If 1, double-word wide area was saved. If zero, word wide area was saved.
	bits 2-15		RESERVED

SAVE_AREA	-322 -\$142		Save up to 4 planes, 16 longwords per plane.
-----------	-------------	--	--

SAVE_LEN, SAVE_ADDR, SAVE_STAT, and SAVE_AREA are used by the system to save the screen from under the mouse cursor.

USER_TIM	-066 -\$042	long	
NEXT_TIM	-062 -\$03E	long	

USER_TIM is a pointer to a user installed routine executed on each system timer tick. When done, this routine should jump to the address held in NEXT_TIM. For more information, see the VDI manual under "Exchange Timer Interrupt Vector."

USER_BUT	-058 -\$03A	long	User button vector
USER_CUR	-054 -\$036	long	User cursor vector
USER_MOT	-050 -\$032	long	User motion vector
V_CEL_HT	-046 -\$02E	word	Height of alpha cell in pixels
V_CEL_MX	-044 -\$02C	word	Maximum alpha cell X Number of cells across -1

Section 3: The Line A Variable Structure

V_CEL_MY	-042 -\$02A	word	Maximum cell Y Number of cells high -1
V_CEL_WR	-040 -\$028	word	Byte displacement to next vertical alpha cell
V_COL_BG	-038 -\$026	word	Physical color index of background color
V_COL_FG	-036 -\$024	word	Physical color index of foreground color
V_CUR_AD	-034 -\$022	long	Current alpha cursor address
V_CUR_OF	-030 -\$01E	word	Byte offset from screen base to top of first cell
V_CUR_XY	-028 -\$01C -026 -\$01A	word word	Alpha cursor position: cell x Alpha cursor position: cell y
V_PERIOD	-024 -\$018	byte	Alpha cursor flash period (in frames)
V_CUR_CT	-023 -\$017	byte	Alpha cursor countdown timer to next toggle of the cursor form
V_FNT_AD	-022 -\$016	long	Address of monospace font data
V_FNT_ND	-018 -\$012	word	Last ASCII code in font
V_FNT_ST	-016 -\$010	word	First ASCII code in font
V_FNT_WD	-014 -\$00E	word	Width of font form in bytes
V_REZ_HZ	-012 -\$00C	word	Horizontal pixel resolution
V_OFF_AD	-010 -\$00A	long	Address of font offset table (per VDI spec)
	-006 -\$006	word	RESERVED
V_REZ_VT	-004 -\$004	word	Vertical pixel resolution
BYTES_LIN	-002 -\$002	word	Width of destination memory form: set with same value as in WIDTH
PLANES	+000 \$000	word	Number of bit planes in current resolution
WIDTH	+002 \$002	word	Contains the width of the destination memory form (usually screen) in bytes.

Low resolution: \$A0 (160 decimal)
 Medium resolution: \$A0 (160 decimal)
 High resolution: \$50 (80 decimal)

CONTRL	+004 \$004	long	Pointer to CONTRL array
--------	------------	------	-------------------------

Section 3: The Line A Variable Structure

INTIN	+008 \$008	long	Pointer to INTIN array
PTSIN	+012 \$00C	long	Pointer to PTSIN array
INTOUT	+016 \$010	long	Pointer to INTOUT array
PTSOUT	+020 \$014	long	Pointer to PTSOUT array
COLBIT0	+024 \$018	word	Current color bit-plane values for plane 0, 1, 2, and 3, respectively.
COLBIT1	+026 \$01A	word	
COLBIT2	+028 \$01C	word	
COLBIT3	+030 \$01E	word	

Many Line A functions use the COLBITs to determine what color to use while drawing. Each of the COLBITs corresponds to one bit plane in the image, and are labelled for which bit plane they affect. COLBIT0 affects bit plane 0, COLBIT1 bit plane 1, etc. If the value in a COLBIT is zero, the bit is cleared in the affected plane. If the value is nonzero, the bit is set in the affected plane.

LSTLIN	+032 \$020	word
--------	------------	------

If LSTLIN is zero, the last pixel in a line is drawn. Nonzero, and the last pixel is not drawn. This is provided in case you are drawing a series of connected lines, using a writing mode like XOR, where if two lines try to plot the same endpoint it will disappear.

LNMASK	+034 \$022	word	Equivalent to VDI's Polyline Type, described in the VDI manual, under "Set Polyline Line Type".
--------	------------	------	---

WMODE	+036 \$024	word	Equivalent to VDI's Writing Mode, described in the VDI manual, under "Set Writing Mode".
-------	------------	------	--

The four VDI writing modes are:

- (0) Replace Mode -- Ignores the currently displayed image, replaces it with Fore AND Mask. i.e. $New = (fore \text{ AND } mask)$
- (1) Transparent Mode -- Only affects the pixels where the mask is 1. These are changed to the foreground value. i.e. $New = (fore \text{ AND } mask) \text{ OR } (old \text{ AND NOT } mask)$
- (2) XOR Mode -- Reverses the bits representing the color. i.e. $New = mask \text{ XOR } old$
- (3) Reverse Transparent Mode -- Only affects the pixels where the mask is 0. These are changed to the foreground value. i.e. $New = (old \text{ AND } mask) \text{ or } (fore \text{ AND NOT } mask)$

There are several additional writing modes available for functions like TextBit:

Section 3: The Line A Variable Structure

<u>OP</u>	<u>Combination Rule</u>
0	D' = 0
1	D' = S AND D
2	D' = S AND [NOT D]
3	D' = S (Replace Mode)
4	D' = [NOT S] AND D (Erase Mode)
5	D' = D
6	D' = S XOR D (XOR mode)
7	D' = S OR D
8	D' = NOT [S OR D]
9	D' = NOT [S XOR D]
A	D' = NOT D
B	D' = S OR [NOT D]
C	D' = NOT S
D	D' = [NOT S] OR D
E	D' = NOT [S AND D]
F	D' = 1

X1	+038 \$026	word	x1 coordinate
Y1	+040 \$028	word	y1 coordinate
X2	+042 \$02A	word	x2 coordinate
Y2	+044 \$02C	word	y2 coordinate

These variables are often used when a Line A routine needs X and Y coordinates as input, as with Line and Filled Rectangle.

PATPTR	+046 \$02E	long	Pointer to the current fill pattern
--------	------------	------	-------------------------------------

Functions like Horizontal Line and Filled Rectangle look here for the address of their fill pattern.

PATMSK	+050 \$032	word	Fill pattern "mask".
--------	------------	------	----------------------

This value is ANDed with Y1, and the result used as the offset into the fill pattern.

This maintains alignment of the pattern in relation to the screen. In most cases, this also acts as the length of the pattern minus one, thus a one-word pattern would merit a zero, a sixteen word pattern a fifteen, etc. Usually, the pattern should be a power of two in length.

MFILL	+052 \$034	word	Multi-plane fill flag:
-------	------------	------	------------------------

If MFILL is zero, the fill pattern is single plane. If MFILL is nonzero, the fill pattern is multiple plane.

CLIP	+054 \$036	word	Clipping flag: 0=clipping disabled, nonzero=clipping enabled
------	------------	------	--

Section 3: The Line A Variable Structure

XMINCL	+056 \$038	word	Minimum X clipping value
YMINCL	+058 \$03A	word	Minimum Y clipping value
XMAXCL	+060 \$03C	word	Maximum X clipping value
YMAXCL	+062 \$03E	word	Maximum Y clipping value
XDDA	+064 \$040	word	Accumulator for textbit x dda. Should be initialized to \$8000 before each invocation of TextBit that requires scaling.
DDAINC	+066 \$042	word	Fractional amount to scale up or down. If scaling up, set DDAINC to $256 * (\text{Intended size} - \text{Actual size}) / \text{Actual size}$. If scaling down, set DDAINC to $256 * (\text{Intended size}) / \text{Actual size}$.
SCALDIR	+068 \$044	word	Scale direction flag (0=down, 1=up)
MONO	+070 \$046	word	Current font monospaced? 0 = current font is not monospaced OR special effects may increase/decrease the size of the form. 1 = current font is monospaced AND thickening is the only special effect allowed.
SOURCEX	+072 \$048	word	X coord of character in font form
SOURCEY	+074 \$04A	word	Y coord of character in font form
			SOURCEX can be computed from information held in the font header. (See VDI manual for font format)
			temp = character value; temp -= fnt_ptr->first_ade; SOURCEX = fnt_ptr->off_table(temp);
			SOURCEY is usually set to zero (top line of the font form.)
DESTX	+076 \$04C	word	X coordinate of character on screen
DESTY	+078 \$04E	word	Y coordinate of character on screen
DELX	+080 \$050	word	Width of character
DELY	+082 \$052	word	Height of character

DELX and DELY can be computed from the font header.

```
temp = character value;
temp -= fnt_ptr->first_ade;
SOURCEX = fnt_ptr->off_table(temp);
DELX = fnt_ptr->offtable(temp+1)-SOURCEX;
DELY = fnt_ptr->form_height;
```


Section 3: The Line A Variable Structure

FBASE	+084 \$054	long	Pointer to font data
FWIDTH	+088 \$058	word	Width of font form

FBASE and FWIDTH can be retrieved from the font header.

```
FBASE = fnt_ptr->dat_table;  
FWIDTH = fnt_ptr->form_width;
```

STYLE	+090 \$05A	word	TextBit special effects flags bit 0 = Thicken flag bit 1 = Lighten flag bit 2 = Skewing flag bit 3 = Underline flag (Not handled by TextBit) bit 4 = Outline flag Set the bits to select the desired effects. Underlining is done by the application.
LITEMASK	+092 \$05C	word	Mask used to lighten text (typically \$5555)
SKEWMASK	+094 \$05E	word	Mask used to skew text (typically \$5555)
WEIGHT	+096 \$060	word	Width by which to thicken text
ROFF	+098 \$062	word	Offset above character baseline when skewing
LOFF	+100 \$064	word	Offset below character baseline when skewing.

The above five input variables can be computed from the font header.

```
LITEMASK = fnt_ptr->lighten;  
SKEWMASK = fnt_ptr->skew;  
WEIGHT = fnt_ptr->thicken;
```

```
if (skewing) {  
  ROFF = fnt_ptr->right_offset;  
  LOFF = fnt_ptr->left_offset;  
}
```

```
else {  
  ROFF = 0;  
  LOFF = 0;  
}
```

SCALE	+102 \$066	word	Scaling flag (0=no scaling)
--------------	-------------------	-------------	---------------------------------------

Section 3: The Line A Variable Structure

CHUP	+104 \$068	word	Character rotation vector 0 = normal horizontal orientation 900 = rotated 90 degrees clockwise 1800 = rotated 180 degrees clockwise 2700 = rotated 270 degrees clockwise
TEXTFG	+106 \$06A	word	Text foreground color
SCRTP	+108 \$06C	long	Pointer to two contiguous special effects buffers for TextBit
SCRPT2	+112 \$070	word	Offset to beginning of the second buffer in above form.

Each of these special effects buffers must be large enough to contain the worst-case (largest) result of any special effects you may be using. Determine that size, then set aside twice that amount, with SCRTP pointing to the beginning of the buffers. Set SCRPT2 to indicate the offset to the beginning of the second buffer.

TEXTBG	+114 \$072	word	Text background color
COPYTRAN	+116 \$074	word	Copy raster form type flag zero = opaque Nonzero = transparent
SEEDABORT	+118 \$076	long	Pointer to a routine called from within seedfill to allow the fill to be aborted. The routine is called after each horizontal line fill. Initialized to point to a dummy routine that returns FALSE (0). Returning TRUE (nonzero) aborts the seedfill.