Introduction to Bresenham's Line Algorithm Using the SBIT Instruction; Series 32000® Graphics Note 5

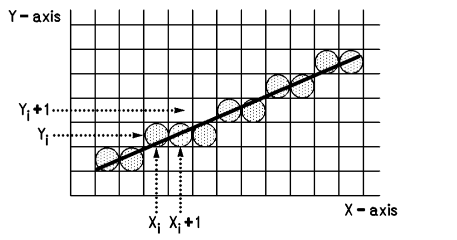
National Semiconductor
Application Note 524
Nancy Cossitt
April 1988

1.0 INTRODUCTION

Even with today's achievements in graphics technology, the resolution of computer graphics systems will never reach that of the real world. A true real line can never be drawn on a laser printer or CRT screen. There is no method of accurately printing all of the points on the continuous line described by the equation $y = mx + b$. Similarly, circles, ellipses and other geometrical shapes cannot truly be implemented by their theoretical definitions because the graphics system itself is discrete, not real or continuous. For that reason, there has been a tremendous amount of research and development in the area of discrete or raster mathematics. Many algorithms have been developed which "map" real-world images into the discrete space of a raster device. Bresenham's line-drawing algorithm (and its derivatives) is one of the most commonly used algorithms today for describing a line on a raster device. The algorithm was first published in Bresenham's 1965 article entitled "Algorithm for Computer Control of a Digital Plotter". It is now widely used in graphics and electronic printing systems. This application note will describe the fundamental algorithm and show an implementation on National Semiconductor's Series 32000 microprocessor using the SBIT instruction, which is particularly well-suited for such applications. A timing diagram can be found in *Figure 8* at the end of the application note.

2.0 DESCRIPTION

Bresenham's line-drawing algorithm uses an iterative scheme. A pixel is plotted at the starting coordinate of the line, and each iteration of the algorithm increments the pixel one unit along the major, or x-axis. The pixel is incremented along the minor, or y-axis, only when a decision variable (based on the slope of the line) changes sign. A key feature of the algorithm is that it requires only integer data and simple arithmetic. This makes the algorithm very efficient and fast.



TL/EE/9665–1

FIGURE 1

The algorithm assumes the line has positive slope less than one, but a simple change of variables can modify the algorithm for any slope value. This will be detailed in section 2.2.

2.1 Bresenham's Algorithm for $0 <$ slope < 1

Figure 1 shows a line segment superimposed on a raster grid with horizontal axis X and vertical axis Y. Note that x_i and y_i are the integer abscissa and ordinate respectively of each pixel location on the grid.

Given (x_i, y_i) as the previously plotted pixel location for the line segment, the next pixel to be plotted is either $(x_i + 1, y_i)$ or $(x_i + 1, y_i + 1)$. Bresenham's algorithm determines which of these two pixel locations is nearer to the actual line by calculating the distance from each pixel to the line, and plotting that pixel with the smaller distance. Using the familiar equation of a straight line, $y = mx + b$, the y value corresponding to $x_i + 1$ is

$$y = m(x_i + 1) + b$$

The two distances are then calculated as:

$$d1 = y - y_i$$
$$d1 = m(x_i + 1) + b - y_i$$
$$d2 = (y_i + 1) - y$$
$$d2 = (y_i + 1) - m(x_i + 1) - b$$

and,

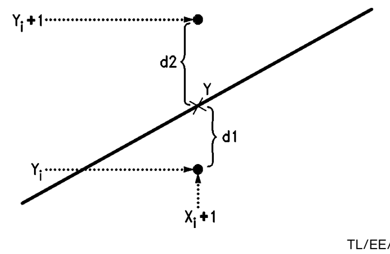
$$d1 - d2 = m(x_i + 1) + b - y_i - (y_i + 1) + m(x_i + 1) + b$$
$$d1 - d2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

Multiplying this result by the constant dx, defined by the slope of the line $m = dy/dx$, the equation becomes:

$$dx(d1-d2) = 2dy(x_i) - 2dx(y_i) + c$$

where c is the constant $2dy + 2dxb - dx$. Of course, if $d2 > d1$, then $(d1-d2) < 0$, or conversely if $d1 > d2$, then $(d1-d2) > 0$. Therefore, a parameter p_i can be defined such that

$$p_i = dx(d1-d2)$$
$$p_i = 2dy(x_i) - 2dx(y_i) + c$$



TL/EE/9665–2

Distances d1 and d2 are compared.
The smaller distance marks next pixel to be plotted.

FIGURE 2

If $p_i > 0$, then d1 > d2 and y_{i+1} is chosen such that the next plotted pixel is $(x_i + 1, y_i)$. Otherwise, if $p_i < 0$, then d2 > d1 and $(x_i + 1, y_i + 1)$ is plotted. (See *Figure 2*.)

Similarly, for the next iteration, p_{i+1} can be calculated and compared with zero to determine the next pixel to plot. If $p_{i+1} < 0$, then the next plotted pixel is at $(x_{i+1} + 1, y_{i+1})$; if $p_{i+1} > 0$, then the next point is $(x_{i+1} + 1, y_{i+1} + 1)$. Note that in the equation for p_{i+1}, $x_{i+1} = x_i + 1$.

$$p_{i+1} = 2dy(x_i + 1) - 2dx(y_{i+1}) + c$$

Subtracting p_i from p_{i+1}, we get the recursive equation:

$$p_{i+1} = p_i + 2dy - 2dx(y_{i+1} - y_i)$$

Note that the constant c has conveniently dropped out of the formula. And, if $p_i < 0$ then $y_{i+1} = y_i$ in the above equation, so that:

$$p_{i+1} = p_i + 2dy$$

or, if $p_i > 0$ then $y_{i+1} = y_i + 1$, and

$$p_{i+1} = p_i + 2(dy - dx)$$

To further simplify the iterative algorithm, constants c1 and c2 can be initialized at the beginning of the program such that $c1 = 2dy$ and $c2 = 2(dy-dx)$. Thus, the actual meat of the algorithm is a loop of length dx, containing only a few integer additions and two compares *(Figure 3)*.

2.2 For Slope < 0 and |Slope| > 1

The algorithm fails when the slope is negative or has absolute value greater than one ($|dy| > |dx|$). The reason for this is that the line will always be plotted with a positive slope if x_i and y_i are always incremented in the positive direction, and the line will always be "shorted" if $|dx| < |dy|$ since the algorithm executes once for every x coordinate (i.e., dx times). However, a closer look at the algorithm must be taken to reveal that a few simple changes of variables will take care of these special cases.

For negative slopes, the change is simple. Instead of incrementing the pixel along the positive direction (+1) for each iteration, the pixel is incremented in the negative direction. The relationship between the starting point and the finishing point of the line determines which axis is followed in the negative direction, and which is in the positive. *Figure 4* shows all the possible combinations for slopes and starting points, and their respective incremental directions along the X and Y axis.

Another change of variables can be performed on the incremental values to accommodate those lines with slopes greater than 1 or less than −1. The coordinate system containing the line is rotated 90 degrees so that the X-axis now becomes the Y-axis and vice versa. The algorithm is then performed on the rotated line according to the sign of its slope, as explained above. Whenever the current position is incremented along the X-axis in the rotated space, it is actually incremented along the Y-axis in the original coordinate space. Similarly, an increment along the Y-axis in the rotated space translates to an increment along the X-axis in the original space. *Figure 4a., g.* and *h.* illustrates this translation process for both positive and negative lines with various starting points.

3.0 IMPLEMENTATION IN C

Bresenham's algorithm is easily implemented in most programming languages. However, C is commonly used for many application programs today, especially in the graphics area. The Appendix gives an implementation of Bresenham's algorithm in C. The C program was written and executed on a SYS32/20 system running UNIX on the NS32032 processor from National. A driver program, also written in C, passed to the function starting and ending points for each line to be drawn. *Figure 6* shows the output on an HP laser jet of 160 unique lines of various slopes on a bit map of 2,000 x 2,000 pixels. Each line starts and ends exactly 25 pixels from the previous line.

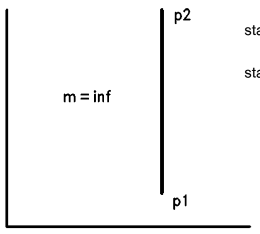
The program uses the variable *bit* to keep track of the current pixel position within the 2,000 x 2,000 bit map *(Figure 5)*. When the Bresenham algorithm requires the current position to be incremented along the X-axis, the variable *bit* is incremented by either +1 or −1, depending on the sign of the slope. When the current position is incremented along the Y-axis (i.e., when p > 0) the variable *bit* is incremented by +warp or −warp, where *warp* is the vertical bit displacement of the bit map. The constant *last bit* is compared with *bit* during each iteration to determine if the line is complete. This ensures that the line starts and finishes according to the coordinates passed to the function by the driver program.

```
do while count < > dx
   if (p < 0) then p+ = c1
   else

                    p+ = c2
                    next_y = prev_y + y_inc
   next_x = prev_x + x_inc
   plot(next_x,next_y)
   count + = 1

/* PSEUDO CODE FOR BRESENHAM LOOP */
```

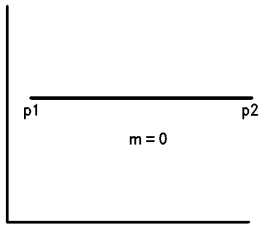
FIGURE 3



start p1: x_inc = y′_inc = 0
y_inc = x′_inc = +1
start p2: x_inc = y′_inc = 0
y_inc = x′_inc = − 1

TL/EE/9665–3

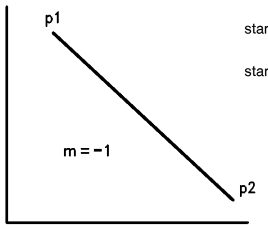
a.



start p1: x_inc = +1
y_inc = 0
start p2: x_inc = −1
y_inc = 0

TL/EE/9665–4

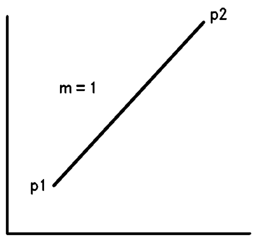
b.



start p1: x_inc = +1
y_inc = −1
start p2: x_inc = −1
y_inc = +1

TL/EE/9665–5

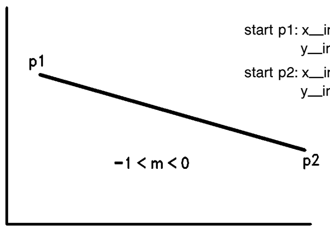
c.



start p1: x_inc = +1
y_inc = +1
start p2: x_inc = −1
y_inc = −1

TL/EE/9665–6

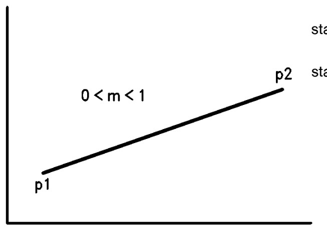
d.



start p1: x_inc = +1
y_inc = −1
start p2: x_inc = −1
y_inc = +1

TL/EE/9665–7

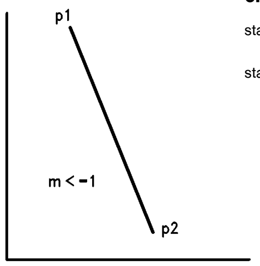
e.



start p1: x_inc = +1
y_inc = +1
start p2: x_inc = −1
y_inc = −1

TL/EE/9665–8

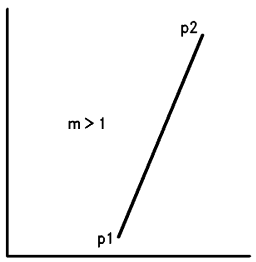
f.



start p1: x_inc = y′_inc = +1
y_inc = x′_inc = −1
start p2: x_inc = y′_inc = −1
y_inc = x′_inc = +1

TL/EE/9665–9

g.



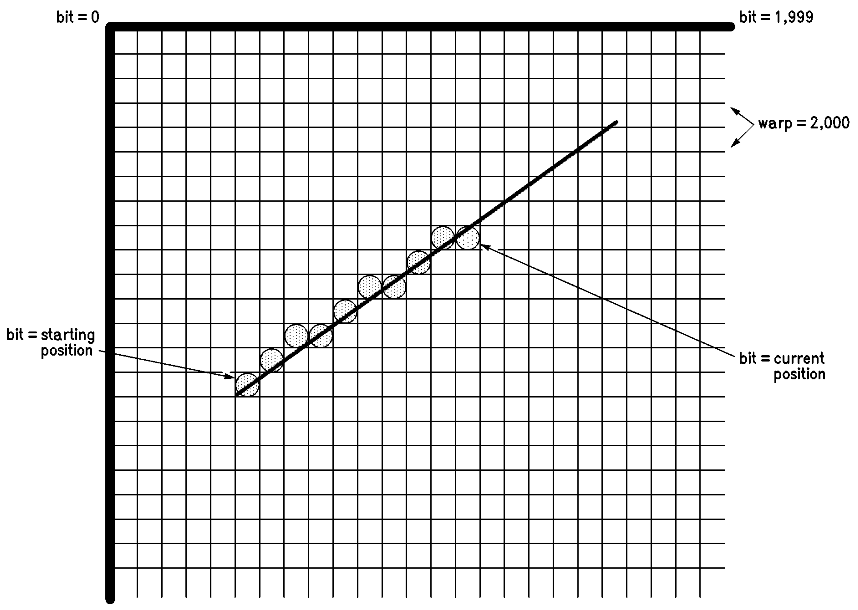
start p1: x_inc = y′_incl = −1
y_inc = x′_inc = +1
start p2: x_inc = y′_inc = +1
y_inc = x′_inc = −1

TL/EE/9665–10

h.

Note: a., g., and h. are rotated 90 degrees left and x′, y′ refer to the original axis.

FIGURE 4

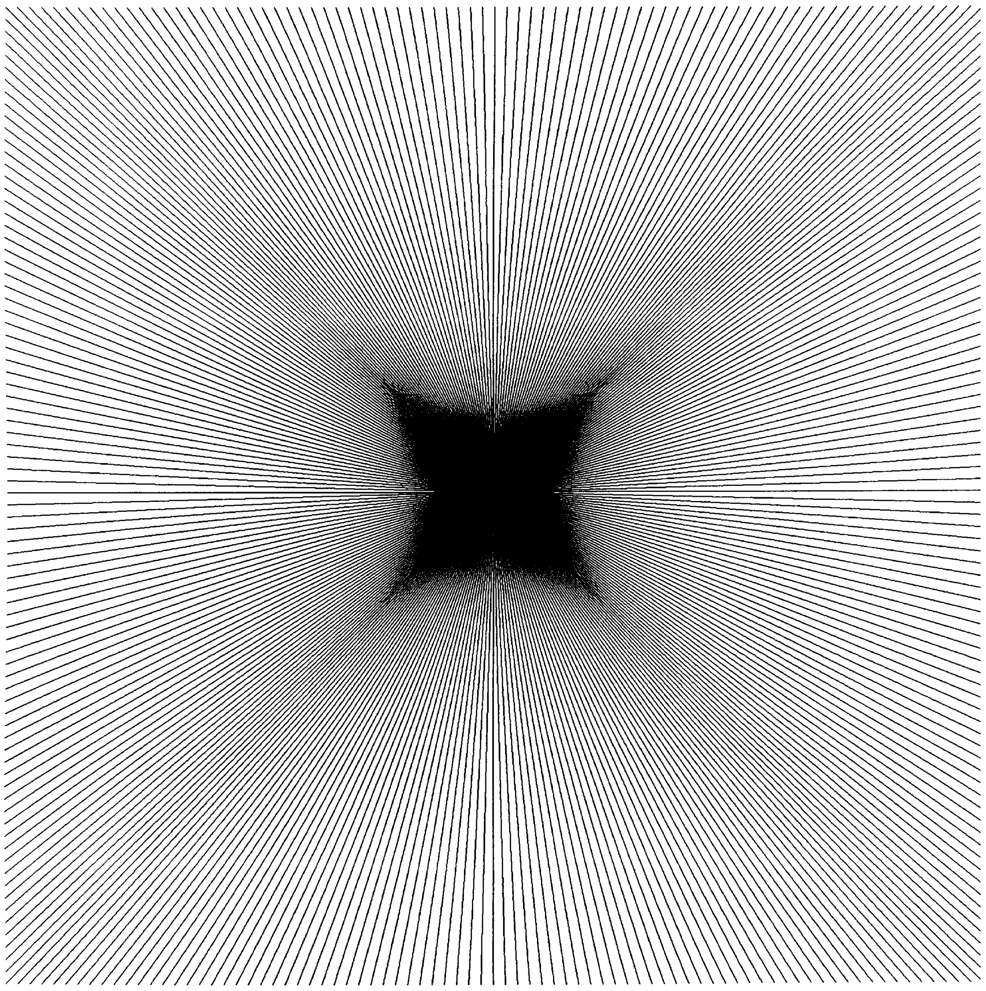


TL/EE/9665–11

Bit Map is 500 kbytes, 2k x 2k Bits
Base Address of Bit Map is 'Bit__Map'

FIGURE 5

Graphics Image (2000 x 2000 Pixels), 300 DPI



TL/EE/9665–12

FIGURE 6. Star-Burst Benchmark—This Star-Burst image was done on a 2k x 2k pixel bit map. Each line is 2k pixels in length and passes through the center of the image, bisecting the square. The lines are 25 pixel units apart, and are drawn using the LINE__DRAW.S routine. There are a total of 160 lines. The total time for drawing this Star-Burst is 2.9 sec on 10 MHz NS32C016.

4.0 IMPLEMENTATION IN SERIES 32000 ASSEMBLY: THE SBIT INSTRUCTION

National's Series 32000 family of processors is well-suited for the Bresenham's algorithm because of the SBIT instruction. *Figure 7* shows a portion of the assembly version of the Bresenham algorithm illustrating the use of the SBIT instruction. The first part of the loop, handles the algorithm for $p < 0$ and .CASE2 handles the algorithm for $p > 0$. The main loop is unrolled in this manner to minimize unnecessary branches (compare loop structure of *Figure 7* to *Figure 3*). The SBIT instruction is used to plot the current pixel in the line.

The SBIT instruction uses *bit__map* as a base address from which it calculates the bit position to be set by adding the offset *bit* contained in register r1. For example, if *bit*, or R1, contains 2,000*, then the instruction:

sbitd r1,@ *bit__map*

will set the bit at position 2,000, given that *bit__map* is the memory location starting at bit 0 of this grid. In actuality, if *base* is a memory address, then the bit position set is:

offset MOD 8

within the memory byte whose address is:

base + (*offset* DIV 8)

So, for the above example,

2,000 MOD 8 = 0

bit__map + 2,000 DIV 8 = *bit__map* + 250

Thus, bit 0 of byte (*bit__map* + 250) is set. This bit corresponds to the first bit of the second row in *Figure 5*.

*All numbers are in decimal.

The SBIT instruction greatly increases the speed of the algorithm. Notice the method of setting the pixel in the C program given in the Appendix:

bit__map[*bit*/8] | = *bit__pos*[(bit & 7)]

This line of code contains a costly division and several other operations that are eliminated with the SBIT instruction. The SBIT instruction helps optimize the performance of the program. Notice also that the algorithm can be implemented using only 7 registers. This improves the speed performance by avoiding time-consuming memory accesses.

5.0 CONCLUSION

An optimized Bresenham line-drawing algorithm has been presented using the SYS32/20 system. Both Series 32000 assembly and C versions have been included. *Figure 8* presents the various timing results of the algorithm. Most of the optimization efforts have been concentrated in the main loop of the program, so the reader may spot other ways to optimize, especially in the set-up section of the algorithm.

Several variations of the Bresenham algorithm have been developed. One particular variation from Bresenham himself relies on "run-length" segments of the line for speed optimization. The algorithm is based on the original Bresenham algorithm, but uses the fact that typically the decision variable p has one sign for several iterations, changing only once in-between these "run-length" segments to make one vertical step. Thus, most lines are composed of a series of horizontal "run-lengths" separated by a single vertical jump. (Consider the special cases where the slope of the line is exactly 1, the slope is 0 or the slope is infinity.) This algorithm will be explored in the NS32CG16 Graphics Note 5, AN-522, "Line Drawing with the NS32CG16", where it will be optimized using special instructions of the NS32CG16.

```
# Main loop of Bresenham algorithm
.LOOP: #p < 0: move in x direction only
            cmpqd           $0,r4
            ble             .CASE2
            addd            r0,r4
            addd            r5,r1
            sbitd           r1,@_bit_map
            cmpd            r3,r1
            bne             .LOOP
            exit            [r3,r4,r5,r6,r7]
            ret             $0
            .align 4
.CASE2: #P > 0: move in x and y direction
            addd            r2,r4
            addd            r7,r1
            addd            r5,r1
            sbitd           r1,@_bit_map
            cmpd            r1,r3
            bne             .LOOP
            exit            [r3,r4,r5,r6,r7]
            ret             $0
```

```
   Register and Memory
        Contents
r0 = c1 constant
r1 = bit current
     position
r2 = c2 constant
r3 = last_bit
r4 = p decision var
r5 = x_inc increment
r6 = unused register
r7 = y_inc increment
_bit_map = address of
first byte in bit map
```

FIGURE 7

Note: Instructions followed by the letter 'd' indicate "double word" operations.

Timing Performance
2k x 2k Bit Map
2k Pix/Vector 160 Lines per Star-Burst

Version	**NS32000 Assembly with SBIT**	
Parameter	**NS32C016-10**	**NS32C016-15**
Set-up Time Per Vector	45 µs	30 µs
Vectors/Sec	54	82
Pixels/Sec	109,776	164,771
Total Time Star-Burst Benchmark	2.9s	1.9s

FIGURE 8

Set-up time per line is measured from the start of LINE__DRAW.S only. The overhead of calling the LINE__DRAW routine, starting the timer and creating the endpoints of the vector are not included in this time. Set-up time does include all register set-up and branching for the Bresenham algorithm up to the entry point of the main loop.

Vectors/Second is determined by measuring the number of vectors per second the LINE__DRAW routine can draw, not including the overhead of the DRIVER.C and START.C routines, which start the timer and calculate the vector endpoints. All set-up of registers and branching for the Bresenham algorithm are included.

Pixels/Second is measured by dividing the Vectors/Second value by the number of pixels per line.

Total Time for the Star-Burst benchmark is measured from start of benchmark to end. It does include all overhead of START.C and DRIVER.C and all set-up for LINE__DRAW.S. This number can be used to approximate the number of pages per second for printing the whole Star-Burst image.

```
#       National Semiconductor Corporation.
#       CTP version 2.4    -- line_draw.s --

.file   "line_draw.s"
        .comm  _bit_map,499750
        .globl  _line_draw
        .set    WARP,2000
        .align 4
_line_draw:                                # initialize
        enter    [r3,r4,r5,r6,r7],12
        movd     12(fp),r5                 # r5=ys
        movd     8(fp),r6                  # r6=xs
        movd     r5,r1                     # initialize starting 'bit'
        muld     $(WARP),r1                # bit=warp*ys+xs
        addd     r6,r1                     # r1=bit
        movd     20(fp),r4                 # r4=yf
        subd     r5,r4                     # r4=dy
        absd     r4,r3                     # r3=|dy|
        movd     16(fp),r2                 # r2=xf
        subd     r6,r2                     # r2=dx
        absd     r2,r6                     # r6=|dx|
        cmpd     r3,r6                     # branch if slope<1
        ble      .LL1                      # must rotate axis for slope>1
        cmpqd    $(0),r4                   # if dy<0 want x_inc<0
        bge      .LL2                      # else x_inc is pos
        addr     WARP,r5                   # x_inc=+/-warp because of rotate
        br       .LL3
        .align 4
.LL2:
        addr     -WARP,r5
.LL3:
        cmpqd    $(0),r2                   # if dx<0 want y_inc<0
        bge      .LL4                      # else y_inc is pos
        movqd    $(1),r7                   # y_inc=+/-1 becaue of rotate
        br       .LL5
        .align 4
.LL4:
        movqd    $(-1),r7
.LL5:                                      # calculate c1,c2 and p
        movd     r6,r0
        addd     r0,r0                     # r0=c1=2*|dx| because of rotate
        subd     r3,r6                     # r6=|dx-dy|    r2=2*r6=c2
        addr     0[r6:w],r2                # this muls r6 by 2 and puts in r2
        movd     r0,r4
        subd     r3,r4                     # r4=c2-|dy|=p in rotated space
        movd     20(fp),r3                 # calculate last_bit
        muld     $(WARP),r3
        addd     16(fp),r3                 # r3=last_bit
        br       .LL6
        .align 4
.LL1:                                      # slope<1 use original axis
        cmpqd    $(0),r4                   # dy determines y_inc
        bge      .LL7
        addr     WARP,r7                   # dy>0 then y_inc=+warp
        br       .LL8
        .align 4
.LL7:
        addr     -WARP,r7                  # dy<0 then y_inc=-warp
.LL8:
        cmpqd    $(0),r2
        bge      .LL9
        movqd    $(1),r5                   # dx>0 then x_inc=+1
        br       .LL10
        .align 4
.LL9:
        movqd    $(-1),r5                  # dx<0 then x_inc=-1
```

TL/EE/9665–13

```
.LL10:                                     # calculate c1,c2,p
        addr     0[r3:w],r0                # r0=2*r3=c1
        movd     r3,r2
        subd     r6,r2
        addd     r2,r2                     # r2=2*|dy-dx|=c2
        movd     r0,r4
        subd     r6,r4                     # p=2*dy-dx=r4
        movd     20(fp),r3                 # calculate last_bit=r3
        muld     $(WARP),r3
        addd     16(fp),r3
.LL6:                                      # main loop for algorithm
        cmpqd    $(0),r4                   # check sign of p
        ble      .LL11                     # branch if pos
        addd     r0,r4                     # add c1 to p
        addd     r5,r1                     # inc bit by x_inc only
        sbitd    r1,@_bit_map              # plot bit
        cmpd     r3,r1                     # end only if bit=last_bit
        bne      .LL6
        exit     [r3,r4,r5,r6,r7]
        ret      $(0)
        .align 4
.LL11:                                     # p>0 then inc in y dir
        addd     r2,r4                     # add c2 to p
        addd     r7,r1                     # add y_inc to bit
        addd     r5,r1                     # add x_inc to bit
        sbitd    r1,@_bit_map              # plot bit
        cmpd     r1,r3                     # end only when bit=last_bit
        bne      .LL6
        exit     [r3,r4,r5,r6,r7]
        ret      $(0)
```

TL/EE/9665–14

```
/* This program calculates points on a line using Bresenham's iterative */
/* method.  */

#include<stdio.h>
#define xbytes  25Ø             /* number of bytes along x-axis*/
#define warp    xbytes * 8      /* number of bits along x_axis*/
#define maxy    1999            /* number of lines in y_axis*/
unsigned char   bit_map[xbytes*maxy];   /* array contains bit map*/
static unsigned char    bit_pos[]={1,2,4,8,16,32,64,128};
                                 /* look-up table for setting bit */

line_draw(xs,ys,xf,yf)          /* starting (s) and finishing (f) points */

int     xs,ys,xf,yf;

{
        int     dx,dy,x_inc,y_inc,      /* deltas and increments */
                bit,last_bit,           /* current and last bit positions */
                p,c1,c2;                /* decision variable p and constants */

        dx=xf-xs;
        dy=yf-ys;
        bit=(ys*warp)+xs;               /* initialize bit to first bit pos */
        last_bit=(yf*warp)+xf;          /* calculate last bit on line */

        if (abs(dy) > abs(dx))
        {                               /* abs(slope)>1 must rotate space */
                                        /* see Figure 5 a.,g.,and h. */
                if (dy>Ø)
                        x_inc=warp;     /* x_axis is now original y_axis */
                else
                        x_inc= -warp;
                if (dx>Ø)
                        y_inc=1;        /* y_axis is now original x_axis */
                else
                        y_inc= -1;
                c1=2*abs(dx);           /* calculate Bresenham's constants */
                c2=2*(abs(dx)-abs(dy));
                p=2*abs(dx)-abs(dy);    /* p is decision variable now rotated */
        }
        else {                  /* abs(slope)<1 use original axis */
                if (dy>Ø)
                        y_inc=warp;     /* y_inc is +/-warp number of bits */
                else
                        y_inc= -warp;
                if (dx>Ø)
                        x_inc=1;        /* move forward one bit */
                else
                        x_inc= -1;      /* or backward one bit */
                c1=2*abs(dy);           /* calculate constants and p */
                c2=2*(abs(dy)-abs(dx));
                p=2*abs(dy)-abs(dx);
        }

/* Bresenham's Algorithm */
                do      /* do once for each x increment, i.e. dx times */
                {
                        if (p<Ø)                /* no y movement if p<Ø */
                                p+=c1;
                        else {                  /* move in y dir if p>Ø */
                                p+=c2;
                                bit+=y_inc;
                        }
                        bit+=x_inc;             /* always increment x */

                        /* bit is set by calculating bit MOD 8, which is */
```

TL/EE/9665–15

```
                        /* same as bit & 7, then looking up appropriate  */
                        /* bit in table bit_pos.  This bit pos is then set */
                        /* in byte bit/8 */

                        bit_map[bit/8] |= bit_pos[(bit&7)];
                } while (bit!=last_bit);
}
```

TL/EE/9665–16

```
 /* Program driver.c feeds line vectors to LINE_DRAW.S forming Star-Burst.   */

#include <stdio.h>
#define xbytes  25Ø
#define maxx    1999
#define maxy    1999

unsigned char   bit_map[xbytes*maxy];

main()

{
    int i,count;

/* generate Star-Burst image */

        for (count=1;count<=1ØØØ;test++){

                for (i=Ø;i<=maxy;i+=25)
                        line_draw(Ø,i,maxx,maxy-i);
                for (i=Ø;i<=maxx;i+=25)
                        line_draw(i,maxy,maxx-i,Ø);
        }

}
```

TL/EE/9665–17

```
/* Start timer and call main procedure of DRIVER.C to draw lines */

start() {
    long *timer = (long *) Øx6ØØ;
    *timer = Ø;         /* write a zero to timer location */
                main(Ø,Ø);      /* Show argc as zero, argv ->Ø */
    return(*timer);     /* return, in rØ, the current time */
}
```

TL/EE/9665–18

Lit. # 100524

National Semiconductor Corporation
1111 West Bardin Road
Arlington, TX 76017
Tel: 1(800) 272-9959
Fax: 1(800) 737-7018

National Semiconductor Europe
Fax: (+49) 0-180-530 85 86
Email: cnjwge@tevm2.nsc.com
Deutsch Tel: (+49) 0-180-530 85 85
English Tel: (+49) 0-180-532 78 32
Français Tel: (+49) 0-180-532 93 58
Italiano Tel: (+49) 0-180-534 16 80

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductor Japan Ltd.
Tel: 81-043-299-2309
Fax: 81-043-299-2408