Line Drawing with the NS32CG16; NS32CG16 Graphics Note 5

National Semiconductor
Application Note 522
Nancy Cossitt
July 1988

1.0 INTRODUCTION

The Bresenham algorithm, as described in the "Series 32000® Graphics Note 5" is a common integer algorithm used in many graphics systems for line drawing. However, special instructions of the NS32CG16 processor allow it to take advantage of another faster integer algorithm. This application note describes the algorithm and shows an implementation on the NS32CG16 processor using the SBITS (Set BIT String) and SBITPS (Set BIT Perpendicular String) instructions. Timing for the DRAW_LINE algorithm is given in Tables A, B and C of the Timing Appendix. The timing from the original Bresenham iterative method using the NS32CG16 is given in Table D.

The bit map memory conventions followed in this note are the same as those given in the NS32CG16 Reference Manual and Datasheet, and all lines drawn are monochrome. Series 32000 Graphics Note 5, AN-524, is recommended reading.

2.0 DESCRIPTION

All rasterized lines are formed by sequences of line "slices" which are separated by a unit shift diagonal to the direction of these slices. For example, the line shown in *Figure 1* is composed of 7 slices, each slice separated by a unit diagonal shift in the positive direction. Notice that the slices of the line vary in length. The algorithm presented in this note determines the length of each slice, given the slope and the endpoints of the line.

Depending on the slope of the line, these slices will extend along the horizontal axis, the vertical axis or the diagonal axis with respect to the image plane (i.e., a printed page or CRT screen). If the data memory is aligned with the image plane so that a positive one unit horizontal (x-axis) move in the image plane corresponds to a one bit move within a byte in the data memory, and so that a positive one unit vertical (y-axis) move in the image plane corresponds to a positive one "warp" (warp = the number pixels along the major axis of the bit map) move within the data memory, then the SBITS and SBITPS instructions can be used to quickly set bits within data memory to form the line slices on the image plane, as explained in section 3.1. For long horizontal lines, the MOVMP (MOVe Multiple Pattern) instruction is more efficient than SBITS. This instruction is discussed in section 3.1 and in the NS32CG16 Reference Manual.

2.1 Derivation of the Bresenham SLICE Algorithm

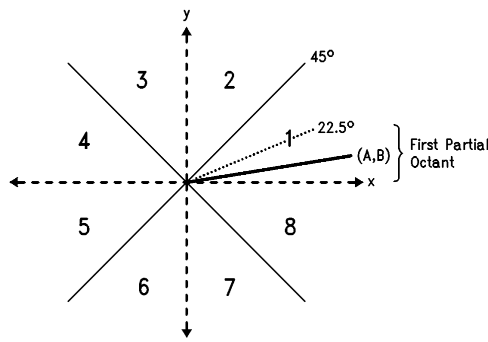
For the moment, consider only those lines in the X-Y coordinate system starting at the origin (0,0), finishing at an integer end point (x,y) and lying in the first partial octant, as in *Figure 2*. (The analysis will be extended for all lines in section 2.2.) The equation for one such line ending at (A,B) is:

$$y = mx,$$

where

$$m = B/A$$

is the slope of the line. Note that because the line lies in the first partial octant, $A > 2B \geq 1$.



TL/EE/9663–2

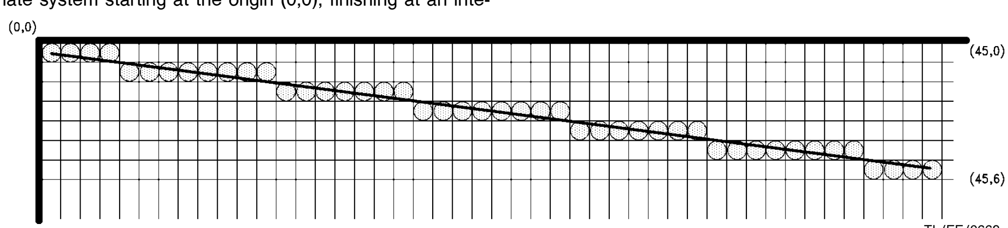
FIGURE 2

Each pixel plotted can be thought of as a unit square area on a Real plane *(Figure 3).* Assume each pixel square is situated so that the center of the square is the integer address of the pixel, and each pixel address is one unit away from its neighbor. Then let A_i represent the X-coordinate of the pixel, as shown in *Figure 3*. The value of Y at A_i is:

$$y = (B/A)A_i$$

where y is Real.

Since the address of each pixel plotted must have corresponding integer coordinates, the closest integer to y is either the upper bound of y or the lower bound. (Recall that upper and lower bounds refer to the smallest integer greater than or equal to y and the largest integer less than or equal to y respectively.) The original Bresenham algorithm was based on this concept, and had a decision variable within the main loop of the algorithm to decide whether the next y_{i+1} was the previous y_i (lower bound) or $y_i + 1$ (upper bound). For the SLICE algorithm, we are only concerned with when the value changes to $y_i + 1$, and the length of the previous slice up to that point.

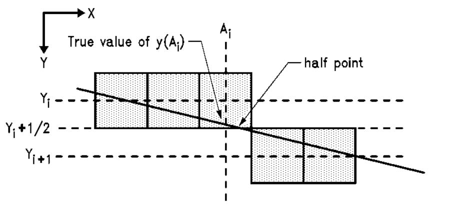


TL/EE/9663–1

The line from (0,0) to (45,6) is a first octant line with run lengths 3-7-6-7-6-7-3. Notice that a pixel is plotted before the run begins so that the actual number of pixels plotted is equivalent to the run length +1.

FIGURE 1

Series 32000® is a registered trademark of National Semiconductor Corporation.



TL/EE/9663–3

Y is incremented when the location of the half point is beyond A_i, or when the true value of Y at A_{i+1} is greater than $Y_i + ½$.

FIGURE 3

In order for y_i to be incremented along the Y-axis, the true value of real y at $A_i + 1$ must be greater than or equal to the halfway point between y_i and $y_i + 1$ *(Figure 3)*. If we let i increment along the Y-axis, then this half point occurs when:

$$y = 1/2 + y_i$$

Or, because $y_i = i$ when incrementing along the Y-axis,

$$y = (1 + 2i)/2.$$

The real value of x at this point is:

$$x = A(1 + 2i)/2B$$

using $x = (1/m)y$. The lower bound of this value of x represents the x-coordinate of the pixel square containing the half point.

Letting A_i and A_{i+1} be two integer values of x where the real value of y is greater than or equal to the half point value $y_i + 1/2$ *(Figure 4)*, then the run length extends from $(A_i + 1, i + 1)$ to $(A_{i+1}, i + 1)$. The run length can then be calculated as:

$$H_{i+1} = A_{i+1} - A_i - 1$$

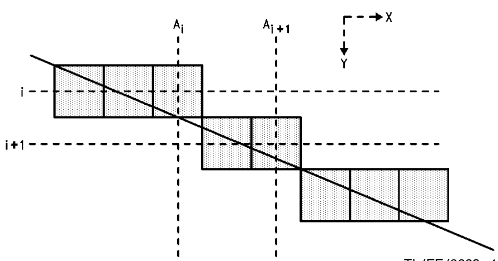
for $i = 0,1, \ldots ,(B-2)$. Using the equation for x above, we can now better define A_i as:

$$A_i = (A/2B) + (iA/B).$$

This equation has two real-valued divisions which are not suitable for an integer algorithm. However, the equation can be broken down so that it only involves an integer-valued division and its integer remainder, which is more efficient for processing. To do this we must define some intermediary integer values:

$\mathbf{Q} = \textbf{lower}[A/B]$	{Lower bound of inverted slope}
$\mathbf{R} = {}_{B}\vert^{A}$	{Integer residue of A modulo B}
$\mathbf{M} = \textbf{lower}[A/2B]$	{Can also be defined as Q/2}
$\mathbf{N} = {}_{2B}\vert^{A}$	{Integer residue of A modulo 2B}
$\mathbf{T_i} = {}_{2B}\vert^{(N+2iR)}$	{Integer residue of (N+2iR) modulo 2B}

Note: ${}_{A}|^{B} = B + A * \textbf{lower}[A/B]$.



TL/EE/9663–4

Run length is calculated as $A_{i+1} - A_i - 1$. In this example, the run length is 1.

FIGURE 4

Using the above values we can now define A_i as,

$$A_i = (M + N/2B) + (iQ + iR/B)$$

$$A_i = M + iQ + (N + 2iR)/2B$$

Therefore, substituting A_i and A_{i+1} into the equation for H_{i+1}, the intermediate horizontal lengths are,

$$H_{i+1} = A_{i+1} - A_i - 1$$

$$H_{i+1} = \{M + (i+1)Q + \textbf{lower}[(N + 2(i+1)R)/2B]\} - \{M + iQ + \textbf{lower}[(N + 2iR)/2B]\} - 1$$

$$H_{i+1} = Q + \textbf{lower}[(N + 2iR)/2B + 2R/2B] - \textbf{lower}[(N + 2iR)/2B] - 1$$

$$H_{i+1} = Q - 1 + \textbf{lower}[(T_i + 2R)/2B]$$

Analyzing the term $\textbf{lower}[(T_i + 2R)/2B]$ it is shown that if $T_i + 2R \geq 2B$ then the term becomes 1, otherwise it becomes 0. This is due to the definition of residue and modulo. The term T_i is defined as:

$$(N + 2iR) - 2B(\textbf{lower}[(N + 2iR)/2B]),$$

which means that $0 \leq T_i < 2B$. The same is true for R:

$$R = A - B(\textbf{lower}[A/B]),$$

so that $0 \leq 2R < 2B$. Therefore,

$$0 \leq T_i + 2R < 4B$$

and,

$$0 \leq (T_i + 2R)/2B < 2.$$

The only possible integer values for this term are 0 and 1. The term will equal 0 if $T_i + 2R < 2B$, and it will equal 1 when $T_i + 2R \geq 2B$, and H_{i+1} will equal Q. The decision variable can now be defined as

$$\text{testvar} = T_i + 2R - 2B.$$

If testvar ≥ 0 then the horizontal run length is Q; if testvar < 0 then the run length is $Q-1$.

Looking again at the definition of T_i, a recursive relationship for the testvar can be formed.

$$T_{i+1} = (N + 2R(i+1)) - 2B(\textbf{lower}[(N + 2R(i+1))/2B]$$

$$T_{i+1} = (N + 2iR + 2R) - 2B(\textbf{lower}[(N + 2iR + 2R)/2B]$$

Since, as shown above, $0 < (T_i + 2R)/2B < 2$ then $\textbf{lower}[(T_i + 2R)/2B] \leq 1$. In fact, if $T_i + 2R < 2B$ then $\textbf{lower}[(T_i + 2R)/2B] = 0$, and if $T_i + 2R \geq 2B$ then $\textbf{lower}[(T_i + 2R)/2B] = 1$. Therefore, letting $T_0 = N$,

$$T_{i+1} = T_i + 2R \quad \text{if } (T_i + 2R) < 2B$$

$$T_{i+1} = T_i + 2R - 2B \quad \text{if } (T_i + 2R) \geq 2B.$$

This gives the recursive relationship for testvar:

$$\text{testvar}_{i+1} = \text{testvar}_i + 2R$$

$$H_i = Q - 1$$

if $\text{testvar}_i < 0$. And, if $\text{testvar}_i \geq 0$:

$$\text{testvar}_{i+1} = \text{testvar}_i + 2R - 2B$$

$$H_i = Q.$$

These recursive equations allow the intermediate run lengths to be easily calculated using only a few additions and compare-and-branches.

The initial run length is calculated as follows:

$$H_0 = A_0 = \textbf{lower}[A/2B] = M + \textbf{lower}[N/2B] = M.$$

The final run length is similarly calculated as:

$$H_f = M - 1 \quad \text{if } N = 0 \text{ else } H_f = M.$$

Thus, the SLICE algorithm calculates the horizontal run lengths of a line using various parameters based on the first partial octant abscissa and ordinate of the line. The algorithm is efficient because it need only execute its main loop B times, which is a maximum of A/2, if A is normalized for the first partial octant. Compare this with the original Bresenham algorithm which always executes its main loop A times.

2.2 Extended Analysis for All Other Lines

In section 2.1 the SLICE algorithm was derived for lines starting at the origin and contained within the first octant (B $<$ 2A). The algorithm is easily extended to encompass lines in all octants starting and ending at any integer coordinates within the pre-defined bit map. The only modifications necessary for this extension are those relating to the direction of movement and in defining the coordinates A and B.

In order to extend the algorithm to cover all classes of lines, the key parameters used by the algorithm must be normalized to the first partial octant. Those parameters are the abscissa and ordinate displacements and the movement of the bit pointer along the line. The abscissa and ordinate displacements of the line are normalized to the first octant by calculating:

$$\text{delta x} = x_f - x_s \text{ and delta y} = y_f - y_s$$

which represent the abscissa (delta x) and ordinate (delta y) displacements of the original line. Then, the first octant equivalents of A and B will be:

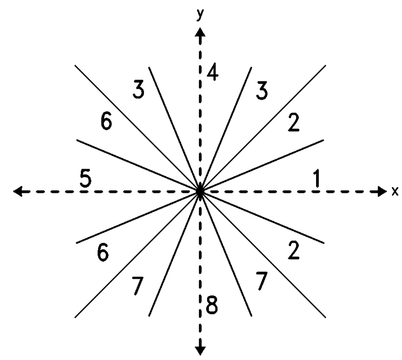
$A = \text{maximum } \{|\text{delta x}|, |\text{delta y}|\}$

$B' = \text{minimum } \{|\text{delta x}|, |\text{delta y}|\}$

$B = \text{minimum } \{B', A - B'\}$

The next step in normalizing the line for the first octant is to assign the correct value to the movement parameters. A line in the first octant and starting at the origin always has horizontal run lengths in the positive direction along the X (major) axis, and has diagonal movement one unit in the positive X direction and one unit in the positive Y (minor) direction. Since the SLICE algorithm calculates the run lengths independent of direction, variables can easily be defined which contain the direction of movement for each slice and each diagonal step within the different octants.

Lines of different angles starting at the origin have slices of different angles. For example, a line of angle between 22.5 degrees and 45 degrees has run lengths that are diagonal, not horizontal, and the direction of the diagonal step is horizontal, not diagonal. Because of this characteristic, it is convenient to break the 8 octants of the X-Y coordinate system into 16 sections, representing all of the partial octants. Then, re-number these partial octants so that they form new octants as in *Figure 5*. These redefined octants represent



TL/EE/9663–5

Redefined octants for SLICE algorithm. Notice that some of the octants are split. The origin is at the center of the drawing. Setting DELX positive on all lines makes opposite octants equivalent in the table below.

FIGURE 5

each of the eight angle classes of lines. For example, the lines in octants 3 and 7 are composed of diagonal (45 degree) slices in either the positive or negative direction, and have diagonal step in the vertical position. Lines in octants 4 and 8 have run length slices in the vertical direction with diagonal steps in the horizontal direction with respect to the X-Y plane.

In conclusion, the SLICE algorithm calculates successive run lengths in the same manner for lines in each octant. The only difference between the octants is the direction of movement of the bit pointer after each successive run length is calculated. The run lengths and diagonal steps for each octant are given in Table I. *Figure 5* shows the octants used by the SLICE algorithm.

3.0 IMPLEMENTATION OF SLICE USING SBITS, SBITPS AND MOVMP

The NS32CG16 features several powerful graphics instructions. The SLICE algorithm described by this application note is implemented with three of these instructions: SBITS, SBITPS and MOVMP. The SBITS instruction allows a horizontal string of bits to be set, while the SBITPS instruction can set vertical or diagonal strings of bits. The MOVMP instruction, not detailed in this application note, can be used to set long strings of bits faster than SBITS when the length is more than 200 bits in the horizontal direction. The BIGSET.S routine given in the appendix uses this instruction in conjunction with SBITS for long lines. These are very useful instructions for the SLICE run length algorithm, as will be shown in section 3.2.

TABLE I

OCTANT	DELA	DELB	DIAGONAL MOVE	RUN LENGTH
1 & 5	DELX	|DELY|	1 + (±WARP)	+HORZ
2 & 6	DELX	DELA-|DELY|	+ 1	±DIAG
3 & 7	|DELY|	DELA-DELX	±WARP	±DIAG
4 & 8	|DELY|	DELX	+ 1	±WARP

If DELX $<$ 0 then the starting and ending coordinates are swapped. This simplifies initialization.

3.1 SBITS and SBITPS Tutorial

SBITS:

SBITS (Set BIT String) sets a string of bits along the horizontal axis of a pre-defined bit map. The instruction sets a string of up to 25 bits in a single execution using four arguments pre-stored in registers R0 through R3.

R0 = (32 bits) Base address of bit-string destination.

R1 = (32 bits, signed) Starting bit-offset from R0.

R2 = (32 bits, unsigned) Run length of the line segment.

R3 = (32 bits) Address of the string look-up table.

The value of the bit offset is used to calculate the bit number within the byte, assuming that the first bit is bit 0 and the last bit is bit 7. A maximum of 7 for the starting bit number added to a maximum of 25 for the run length requires a total of 32 bits. SBITS calculates the destination address of the first byte of the 32-bit double word to contain the string of set bits by the following:

Destination Byte = Base Address + Offset DIV 8.

Then, the starting bit number within the destination byte is:

Starting Bit = Offset MOD 8.

SBITS instruction then calculates the address for the 32-bit double word within the string look-up table (found in the NS32CG16 manual) which will be OR'ed with the 32-bit double word whose starting byte address is Destination Byte, as calculated above. The table is stored as eight contiguous sections, each containing 32 32-bit double words. Each of the eight sections corresponds to a different value of Starting Bit (Offset MOD 8), which has a possible range of 0 through 7. The 32 double words in each section correspond to each value of the run length (up to 25) added to the starting bit offset.

example:

Register Contents

before	after
R0 = 1000	R0 = 1000
R1 = 235	R1 = 235
R2 = 16	R2 = 16
R3 = $stab	R3 = $stab

Destination Address = 1000 + (235 DIV 8) = 1029
Starting Bit = 235 MOD 8 = 3
Table Address = $stab + 4*(16 + (32*3)) = $stab + 448 bytes
32-bit Mask = 0x0007FFF8

This mask value is OR'ed with the 32-bit double word starting at byte address 1029 decimal. Notice that the mask 0x0007FFF8 leaves the first 3 bits and the last 13 bits alone. Thus, a string of 16 bits is set starting at bit number 3 at address 1029 decimal. The contents of the registers are unaffected by the execution of the SBITS instruction.

Since the SBITS instruction can set up to 25 bits in one execution, the run length in R2 can be compared to 25, and a special subroutine executed if it exceeds 25 bits. The subroutine will set the first 25 bits, then subtract 25 from the run length, and compare this to 25 again. This process is repeated until the run length is less than 25, in which case the remaining bits are set and the subroutine returns. The DRAW__LINE algorithm implemented in this application note uses this method for strings of bits to be set less than 200. For horizontal lines greater than 200 pixels in length, the BIGSET routine is more efficient, as described below.

BIGSET:

The utility program BIGSET.S is used to draw longer lines, more than 200 pixels in length, more efficiently than SBITS. BIGSET.S, which is given in the appendix, uses the MOVMP instruction (MOVe Multiple Pattern) to set long strings of bits. Since MOVMP operates on double-word aligned addresses most efficiently, the string is broken up into a starting string within the first byte, a series of bytes to be set, and an ending string which is the leftover bits to be set within the final byte. The starting and ending strings of bits, if any, are set using the SBITS table with an OR instruction.

SBITPS:

SBITPS (Set BIT Perpendicular String) handles both vertical lines and diagonal lines. This instruction also requires four arguments pre-stored in R0 through R3. R0, R1 and R2 are the Base Address, Starting Bit Offset and Run Length respectively, as for SBITS. R3, however, contains the destination warp.

Note: The Destination warp is the number of bits along the horizontal length of the bit map, or the number of bits between scan lines. It is also referred to as the "pitch" of the bit map. Thus, a vertical one-unit move in the positive direction would require adding the value of the warp to the bit pointer. A diagonal or 45 degree line is drawn when the warp is incremented or decremented by one.

The run length is a 32 bit unsigned magnitude.

example:

(Assume that the bit map is a 904 x 904 pixel grid.)

Register Contents

before	after
R0 = 1000	R0 = 1000
R1 = 235	R1 = 235 + (150*904) = 135,835
R2 = 150	R2 = 0
R3 = +904	R3 = +904

Destination Address = 1029
Starting Bit Number = 3
Run Length = 150
Warp = +904

As in the example for SBITS, the Destination Address is 1029, with Starting Bit Number = 3. Since the warp in this example is +904 and the bit map is 904 x 904 bits, the line is vertical, has a length of 150 pixels and starts at bit number 3 within the byte whose address is 1029 decimal. Unlike the SBITS instruction, the SBITPS alters registers R1 and R2 during execution. R1 is set to the position of the last bit set plus the warp. However, this is convenient for drawing the next slice since R1 has been automatically updated to its proper horizontal position for setting the next bit. The bit offset in R1 need only be incremented by +1 or −1 to point to the exact position of the next bit to be set.

Diagonal lines are drawn when the value contained in R3 is an increment of the bit map's warp.

example:
(Assume that the bit map is a 904 x 904 pixel grid.)

Register Contents	
before	**after**
R0 = 1000	R0 = 1000
R1 = 235	R1 = 235 + (150*905) = 135,985
R2 = 150	R2 = 0
R3 = +905	R3 = +905

This example draws a diagonal line with positive slope starting at bit position 3 in byte 1029. Notice that the new value of R1 = 135,985 is exactly 150 pixels offset from the value of R1 in the vertical line drawn in the previous example. Adding +1 to the warp in this example caused the bit position to move not only in the positive vertical direction, but also in the positive horizontal direction, forming a diagonal line.

3.2 Implementation of DRAW_LINE and SLICE on the NS32CG16

Both a C version of the DRAW_LINE algorithm and an NS32CG16 assembly version are given in the appendix. The C program was implemented on SYS32/20 which uses the NS32032 processor. An emulation package developed by the Electronic Imaging Group at National was used to emulate the SBITS and SBITPS instructions in C, and also the MOVMP instruction used for lines longer than 200 pixels. The emulation routines, which cover all NS32CG16 instructions not available on other Series 32000 processors, are available as both C functions and Series 32000 assembly subroutines.

The DRAW_LINE program was first written in C using the emulation functions. Once this version was tested and functional, it was translated into Series 32000 code and further optimized for speed. The assembly version uses the Series 32000 assembly subroutines which emulate the SBITS and SBITPS instructions. NS32CG16 executable code was developed by replacing the emulation subroutine calls with the actual NS32CG16 instruction. The functional and optimized code was finally executed on the NS32CG16 processor with the aid of the DBG16 debugger for downloading the code to an NS32CG16 evaluation board. Timing for lines of various slopes is given in the Timing Appendix.

Most of the optimization efforts are concentrated in the main loop of the SLICE algorithm. Since the use of SBITS or SBITPS for the run length depends on the slope of the line, the code is unrolled for the different octants. This minimizes branching within the main loop, and cuts down on overall execution time. Also, the DRAW_LINE takes advantage of the NS32CG16's ability to draw fast horizontal, vertical and diagonal lines by separating these lines out from the actual Bresenham SLICE algorithm. Therefore, time is not wasted for trivial lines on executing the initialization sections and main loop sections of the SLICE algorithm.

Branching within the initialization section is also minimized by unrolling the code for each octant. Recall from section 2.2 that in order to extend the algorithm over all octants, the abscissa and ordinate displacements must be normalized to the first octant and the run length directions must be modified to preserve the slope of the line. Partitioning the program into "octant" modules makes the initialization for each octant less cluttered with compare-and-branches. Table I shows that each octant has a unique value for DELA and DELB (the normalized abscissa and ordinate displacements). Note that at the beginning of the programs, DELX or $x_f - x_s$ is checked for sign, and if negative, the absolute value function is performed and the starting and ending points are exchanged. This is done because each octant module of the SLICE algorithm only cares about the sign of DELY with respect to coordinate (x_s, y_s). DELX is only important when initializing DELA or DELB, and in this case, only the absolute value is needed.

4.0 SYSTEM SET-UP

NS32CG16 Evaluation Board:

—NS32CG16 with a 30 MHz Clock
—256KB Static RAM Memory (No Wait States)
—2 Serial ports
—MONCG16 Monitor

Host System:

—SYS32/20 running Unix System V
—DBG16 Debugger

Software for Benchmarking:

—START.C	Starts timer and calls DRIVER.
—DRIVER.C	Feeds vectors to DRAW_LINE.
—DRAW_LINE.S	Line drawing routine which includes SLICE.
—BIGSET.S	Uses MOVMPi to set longer lines. Called by DRAW_LINE if length > 200.

4.1 Timing

Timing Assumptions:

1. No wait states are used in the memory.
2. No screen refresh is performed.
3. The overhead referred to as the "driver" overhead is the time it takes to create the endpoints for each vector. This is application dependent, and is not included in the Vector/Sec and Pixel/Sec times.
4. The overhead referred to as the "line drawing" overhead is the time it takes to set up the registers for the actual line drawing routine. This overhead comes from the DRAW_LINE program only and is included in all times.
5. Raw data given in the Timing Appendix for the SBITS, SBITPS and MOVMP is the peak performance for these instructions. These times do not include line drawing overhead or driver overhead.

The timing for this line-drawing application was done so as to give meaningful results for a real graphics application and to allow the reader to calculate additional times if desired. The routines are not optimized for any particular application. All line drawing overhead, such as set-up and branching, is included in the given times for Timing Table A, B and C. The 23 μs driver overhead of the calling routines is not included in the given times for vectors per second and pixels per second. Calculation of these values was done by subtracting the 23 μs out of the average time per vector so that the given times are only for the processing of the vectors. They do not include the overhead of DRIVER.C and START.C (refer to these programs in the appendix).

In addition, the DRAW_LINE algorithm is timed for several test vectors at various strategic points in the code so that the reader may verify set-up times or calculate other relevant times. The program DRAW_LINE.S in the appendix contains markers (e.g., T1, T2 . . .) for each point at which a particular time was taken. The program was run using a driver program (DRIVER.C in the appendix) which consists of several loops which pass test vectors to the DRAW_LINE routine. A "return" instruction was placed at the time marker so that the execution time was only measured up to that marker. These times are given in the Timing Appendix Table E and include total execution time up to each of the markers.

A millisecond interrupt timer on the NS32CG16 evaluation board was used to time the execution. For each execution, the DRIVER program executed its inner loop over 100 times, and sometimes over 1000 times, so that an accurate reading was obtained from the millisecond timer. The final times were divided by this loop count to obtain a "benchmark" time. This benchmark time was divided by the total number of lines drawn to obtain an average time per vector. The overhead of START.C and DRIVER.C in calling the DRAW_LINE.S routine was not counted in the average time per vector or the average time per pixel calculation. Table E of the Timing Appendix gives the timing for each of the markers and the conditions under which these times were taken.

5.0 CONCLUSION

The timing for the DRAW_LINE algorithm is a good indication of the performance of the NS32CG16 in a real application, something which the datasheet specifications can't always show. The timing clearly shows that the NS32CG16 is well-suited for line-drawing applications. Using the SBITS, SBITPS and the MOVMPi instructions, fast line-drawing is achieved for lines of all slopes and lengths. The NS32CG16 is an ideal processor for taking advantage of the much faster SLICE algorithm.

The SLICE algorithm, which calculates run lengths of line segments to form a complete rasterized line, is much faster than its Bresenham predecessor which calculates the line pixel by pixel. The SLICE algorithm always executes the main loop at least twice as fast as the original Bresenham algorithm, which executes its main loop exactly max{|delx|,|dely|} times for each line.

REFERENCES

J.E. Bresenham, IBM, Research Triangle Park, USA. "Run Length Slice Algorithm for Incremental Lines", **Fundamental Algorithms for Computer Graphics,** Springer-Verlag Berlin Heidelberg 1985.

N.M. Cossitt, National Semiconductor, "Bresenham's Line Algorithm Using the SBIT Instruction", **Series 32000 Graphics Note 5, AN-524**, 1988.

National Semiconductor, **NS32CG16 Supplement to the Series 32000 Programmer's Reference Manual**, 1988.

Bresenham's SLICE Algorithm:

1. INITIALIZE PARAMETERS, MAKE NECESSARY ROTATIONS
2. OUTPUT INITIAL RUN LENGTH (H_O) IN PROPER OCTANT DIRECTION
   MOVE DIAGONALLY IN APPROPRIATE DIRECTION TO START OF NEXT RUN LENGTH
3. OUTPUT INTERMEDIATE RUN LENGTHS
   COUNT = COUNT − 1
   IF COUNT ≤ 0 GOTO 4.
   IF TESTVAR < 0 H = Q − 1 AND TESTVAR = TESTVAR + 2*R
   ELSE H = Q AND TESTVAR = TESTVAR + 2*R − 2*DELB
   OUTPUT RUN LENGTH OF LENGTH H IN PROPER DIRECTION
   MOVE DIAGONALLY IN PROPER DIRECTION
   GOTO 3.
4. OUTPUT FINAL RUN LENGTH OF LENGTH H_F
5. END

INITIALIZED PARAMETERS

DELA = MAXIMUM OF {|DELX|,|DELY|}

DELB = MINIMUM OF {|DELA|,DELA-MINIMUM{|DELX|,|DELY|}}

Q = LOWER[DELA/DELB]

R = DELA − DELB*Q

M = LOWER[Q/2]

N = R (IF Q EVEN)
N = R + DELB (IF Q ODD)

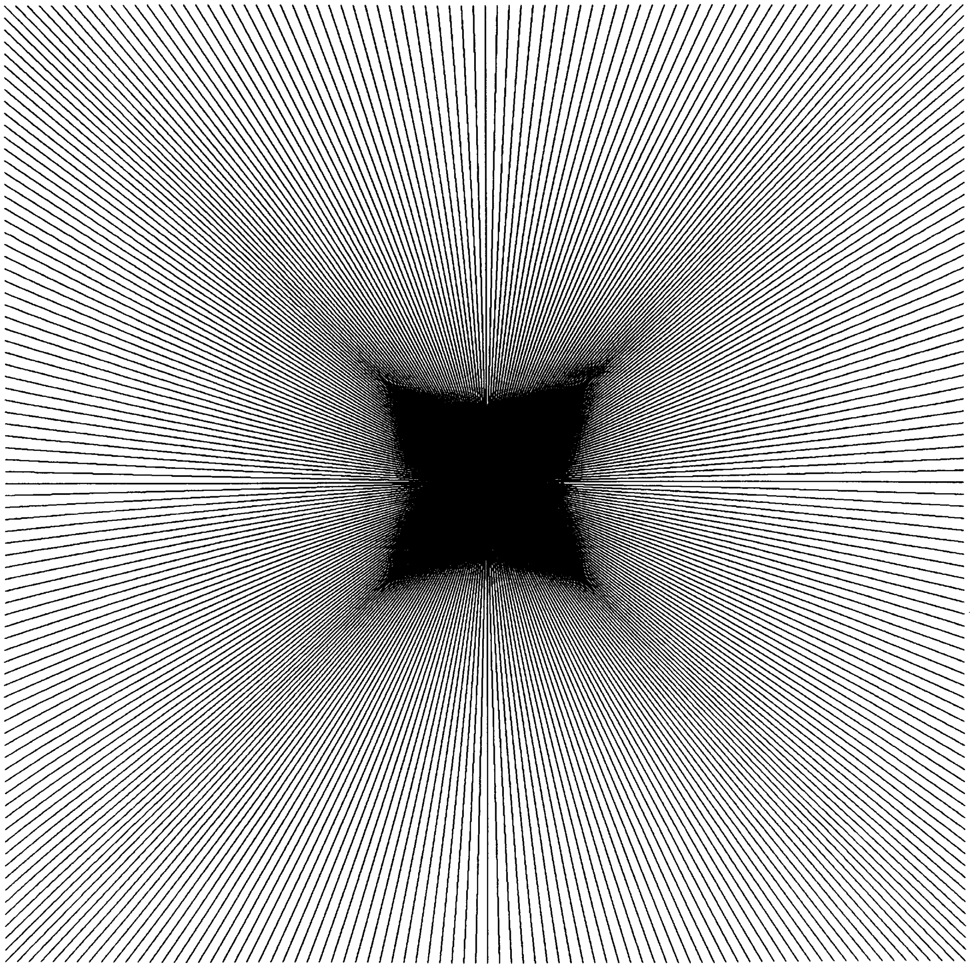
H_O = M (IF DELY ≥ 0 OR N <> 0)
H_O = M − 1 (IF DELY < 0 AND N = 0)

H_F = M (IF DELY < 0 OR N <> 0)
H_F = M − 1 (IF DELY ≥ 0 AND N = 0)

COUNT = DELB

$TESTVAR_0$ = N + 2*R − 2*DELB (IF DELY ≥ 0)
$TESTVAR_0$ = N + 2*R − 2*DELB − 1 (IF DELY < 0)

Graphics Image (2000 x 2000 Pixels), 300 DPI



TL/EE/9663–6

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 DRAW_LINE.S routine. There are a total of 160 lines. The total time for drawing this Star-Burst is 1.0s on 15 MHz NS32CG16.

TIMING APPENDIX

A. PEAK RAW PERFORMANCE AT 15 MHz

Function	Rate*
Horizontal Line (SBITS)	9 MBits/s
Horizontal Line (MOVMP)	60 MBits/s
Vertical Line (SBITPS)	440 kBits/s

*Raw performance does not include any register set-up, branching or other software set-up overhead.

B. TRIVIAL LINES (Using 1k x 1k Bit Map Grid)

	Pixels/Line	Vectors/Sec	Pixels/Sec	Comments**
Horizontal:	1000	13,361	13,361,838	Uses BIGSET.S with MOVMP.
	100	24,136	2,413,593	Uses SBITS only.
	10	45,687	456,870	Uses SBITS only.
Vertical and Diagonal:	1000	424	424,000	Uses SBITPS.
	100	3,975	397,460	
	10	24,491	244,910	

**Pixels/Sec and Vectors/Sec are measured from start of DRAW_LINE.S only. The 23.128 μs driver overhead was not included in these measurements.

C. ALL LINES (Using the "Star-Burst" Benchmark and the SLICE Algorithm)

Pix/Vector	Vectors/Sec	Pixels/Sec	Total Time*	Comments**
1000	318	318,165	0.8s	250 Lines in Star-Burst
100	2,811	281,118	0.019s	50 Lines in Star-Burst
10	14,549	145,490	0.001s	10 Lines in Star-Burst
Avg. Set-up Time Per Line (Measured from Start of DRAW_LINE Only): 37 μs				

D. ALL LINES (Using Original BRESENHAM Iterative Method with SBIT and the Star-Burst Benchmark)

Pix/Vector	Vectors/Sec	Pixels/Sec	Total Time*	Comments**
1000	163	162,746	1.5s	250 Lines in Star-Burst
100	1,568	158,332	0.033s	50 Lines in Star-Burst
10	11,547	127,021	0.001s	10 Lines in Star-Burst
Avg. Set-up Time Per Line (Measured for Line Drawing Routine Only): 30 μs				

The Bresenham program used for the above table can be found in the Series 32000® Graphics Application Note 5.

*Total time is measured from start of execution to finish. It includes all line drawing pre-processing, set-up and branching, and it includes all driver overhead of DRIVER.C and START.C. This time is a good indication of the pages per minute for the complete Star-Burst benchmark. Vectors/Sec and Pixels/Sec are measured from start of DRAW_LINE.S only. The 23.712 μs overhead was not included in these measurements.

**Star-Burst benchmark draws an equal number of lines in each octant. DRIVER.C creates vectors that form the Star-Burst image, passing these vectors to DRAW_LINE.S as they are created. The bit map image can then be downloaded to a printer for a hard copy, as in *Figure 6*.

TIMING APPENDIX TABLE E

Measurement Point	Measured Time/Vector*	Test Vector Used	Octant of Test Vector (Refer to *Figure 5*) And Length of Vector	Comments
T1	23.128 μs	Any Non-Calculated	Any Octant, Any Length	Overhead of entry into DRAW__LINE when not calculating endpoints of line. Application dependent.
	23.712	STAR-BURST	All Octants, 1000 Pixels	Overhead of entry into DRAW__LINE when calculating the STAR-BURST vectors. Application dependent.
T2	40.056	(0,0,0,999)	Vertical, 1000 Pixels/Vector	Average overhead per vertical line to start of line draw instruction (SBITPS).
T3	41.780	(0,999,0,0)	Vertical, 1000 Pixels/Vector	Average overhead per vertical line with negative slope to start of line draw instruction.
T4	40.884	(0,0,999,0)	Horizontal, 1000 Pix/Vect	Average overhead per horizontal line to start of line draw instruction. (SBITS and BIGSET).
	43.912	(999,0,0,0)	Same	Same as above with negative delta × value.
T5	44.532	(0,0,999,999)	Diagonal, 1000 Pix/Vect	Average overhead per diagonal line to start of line draw instruction (SBITPS).
T6	45.356	(0,999,999,0)	Same	Same as above for diagonal line with negative delta × value.
T7	71.164	(0,0,999,10)	Octant 1 1000 Pix/Vect	Average overhead per line to first run length slice of the SLICE algorithm for octant 1.
T8	87.476 75.572 75.568	(0,0,999,10) (0,0,99,10) (0,0,9,2)	Octant 1 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T9	100.348μs 88.444 88.436	(0,0,999,10) (0,0,99,10) (0,0,9,2)	Octant 1 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.
T10	71.856	(0,0,9,8)	Octant 2 10 Pix/Vect	Average overhead per line to first run length. Not dependent on vector length.
T11	79.632 80.040 84.180	(0,0,999,800) (0,0,99,80) (0,0,9,8)	Octant 2 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T12	89.060 89.476 105.376	(0,0,999,800) (0,0,99,80) (0,0,9,8)	Octant 2 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.
T13	73.024	(500,0,700,999)	Octant 3 1000 Pix/Vect	Average overhead per line to first run length. Not dependent on the vector length.
T14	80.736 80.872 80.116	(500,0,700,999) (50,0,70,99) (5,0,7,9)	Octant 3 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T15	89.888 90.020 89.268	(500,0,700,999) (50,0,70,99) (5,0,7,9)	Octant 3 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.
T16	73.712	(10,0,990,999)	Octant 4 1000 Pix/Vect	Average overhead per line to first run length. Not dependent on the vector length.
T17	137.532 81.148 78.256	(10,0,999,999) (10,0,90,99) (2,0,8,9)	Octant 4 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T18	147.236 90.856 87.956	(10,0,999,999) (10,0,90,99) (2,0,8,9)	Octant 4 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.

*Each time was measured from start of benchmark execution to the Tx marker in the DRAW__LINE.S program. Thus, the overhead of the calling routine to the DRAW__LINE routine is T1 = 23.712 μs for the STAR-BURST benchmark. All programs used for timing are included in the Appendix. All times given above are for a 1k x 1k bit map.

```
/* This program draws a line in a defined bit map using Bresenham's */
/* SLICE algorithm. */

#include<stdio.h>
#define xbytes  25Ø
#define warp    2ØØØ
#define maxy    1999
unsigned char   bit_map[xbytes*maxy];
extern unsigned char    sbitstab[];

draw_line(xs,ys,xt,yt)

int     xs,ys,xt,yt;

{
        int     bit,i,j,delx,dely,dela,delb,
                hf,h,hØ,testvar,q,r,m,
                n,count,xinc,yinc;

        delx=xt-xs;
        dely=yt-ys;

        if (xt-xs<Ø){
                xs=xt;
                ys=yt;
                delx=abs(delx);
                dely= -dely;
        }
        bit=xs+ys*warp;
        if (delx==Ø){
                if (dely>=Ø){
                        sbitps(bit_map,bit,dely,warp);
                        return;
                }
                else{
                        sbitps(bit_map,bit,abs(dely),-warp);
                        return;
                }
        }
        if (dely==Ø){
                sbits(bit_map,bit,delx,sbitstab);
                return;
        }
        if (abs(delx)==abs(dely)){
                if(delx*dely>=Ø){
                        sbitps(bit_map,bit,abs(dely),warp+1);
                        return;
                }
                else {
                        sbitps(bit_map,bit,delx,-warp+1);
                        return;
                }
        }
if (abs(delx)>abs(dely)){

        if (abs(dely)<(delx-abs(dely)))
        {
                dela=delx;
                delb=abs(dely);
                xinc=1;
                if (dely>=Ø)
                        yinc=warp;
                else
                        yinc= -warp;

                q=dela/delb;
```

TL/EE/9663–7

```
        r=dela-delb*q;
        m=q/2;
        if (q-2*(q/2)==Ø)
                n=r;
        else
                n=r+delb;

        if ((dely>=Ø)||(n!=Ø))
                hØ=m;
        else
                hØ=m-1;

        if ((dely<Ø)||(n!=Ø))
                hf=m;
        else
                hf=m-1;

        count=delb;

        if(dely>=Ø)
                testvar=n+2*r-2*delb;
        else
                testvar=n+2*r-2*delb-1;
        sbits(bit_map,bit,hØ+1,sbitstab);
        bit=bit+hØ+yinc+xinc;

        for(i=count-1;i>Ø;i--) {
                if (testvar<Ø){
                        h=q-1;
                        testvar+=2*r;
                }
                else {
                        h=q;
                        testvar+=2*r-2*delb;
                }
                sbits(bit_map,bit,h+1,sbitstab);
                bit=bit+h+yinc+xinc;

        }
        sbits(bit_map,bit,hf,sbitstab);
        return;
}
else{
        dela=abs(delx);
        delb=dela-abs(dely);
        xinc=1;
        if(dely>=Ø)
                yinc=warp;
        else
                yinc= -warp;
        q=dela/delb;
        r=dela-delb*q;
        m=q/2;
        if (q-2*(q/2)==Ø)
                n=r;
        else
                n=r+delb;
        if ((dely>=Ø)||(n!=Ø))
                hØ=m;
        else
                hØ=m-1;

        if ((dely<Ø)||(n!=Ø))
                hf=m;
        else
                hf=m-1;
```

TL/EE/9663–8

```
                count=delb;

                if(dely>=Ø)
                        testvar=n+2*r-2*delb;
                else
                        testvar=n+2*r-2*delb-1;
                sbitps(bit_map,bit,hØ+1,yinc+1);
                bit=bit+hØ+hØ*yinc+1;

                for(i=count-1;i>Ø;i--) {
                        if (testvar<Ø){
                                h=q-1;
                                testvar+=2*r;
                        }
                        else {
                                h=q;
                                testvar+=2*r-2*delb;
                        }
                        sbitps(bit_map,bit,h+1,yinc+1);
                        bit=bit+h+yinc*h+1;
                }
                sbitps(bit_map,bit,hf+1,yinc+1);
                return;
        }
}
else{
        if (abs(delx)<(abs(dely)-abs(delx))){
                dela=abs(dely);
                delb=abs(delx);
                yinc=1;
                if(dely>Ø)
                        xinc=warp;
                else
                        xinc= -warp;

                q=dela/delb;
                r=dela-delb*q;
                m=q/2;
                if (q-2*(q/2)==Ø)
                        n=r;
                else
                        n=r+delb;
                if ((dely>=Ø)||(n!=Ø))
                        hØ=m;
                else
                        hØ=m-1;

                if ((dely<Ø)||(n!=Ø))
                        hf=m;
                else
                        hf=m-1;
                count=delb;

                if(dely>=Ø)
                        testvar=n+2*r-2*delb;
                else
                        testvar=n+2*r-2*delb-1;
                sbitps(bit_map,bit,hØ+1,xinc);
                bit=bit+yinc+(1+hØ)*xinc;
                for(i=count-1;i>Ø;i--) {

                        if (testvar<Ø){
                                h=q-1;
                                testvar+=2*r;
                        }
                        else {
```

TL/EE/9663–9

```
                    h=q;
                    testvar+=2*r-2*delb;
                }
                sbitps(bit_map,bit,h+1,xinc);
                bit=bit+yinc+xinc*(1+h);
            }
            sbitps(bit_map,bit,hf+1,xinc);
            return;
        }
        else{
            dela=abs(dely);
            delb=dela-abs(delx);
            yinc=1;
            if(dely>Ø)
                xinc=warp;
            else
                xinc= -warp;

            q=dela/delb;
            r=dela-delb*q;
            m=q/2;
            if (q-2*(q/2)==Ø)
                n=r;
            else
                n=r+delb;
            if ((dely>=Ø)||(n!=Ø))
                hØ=m;
            else
                hØ=m-1;

            if ((dely<Ø)||(n!=Ø))
                hf=m;
            else
                hf=m-1;
            count=delb;

            if(dely>=Ø)
                testvar=n+2*r-2*delb;
            else
                testvar=n+2*r-2*delb-1;
            sbitps(bit_map,bit,hØ+1,xinc+1);
            bit=bit+hØ+(1+hØ)*xinc;
            for(i=count-1;i>Ø;i--) {

                if (testvar<Ø){
                    h=q-1;
                    testvar+=2*r;
                }
                else {
                    h=q;
                    testvar+=2*r-2*delb;
                }
                sbitps(bit_map,bit,h+1,xinc+1);
                bit=bit+h+xinc*(1+h);
            }
            sbitps(bit_map,bit,hf,xinc+1);
            return;
        }
}
}
```

TL/EE/9663–10

```
#       National Semiconductor Corporation.
#       CTP version 2.4    -- draw_line.s --     Tue Nov 17 13:2Ø:24 1987
#       compilation options: -O -S -KC332 -KFØ81 -KB4
#
.file   "draw_line.s"
        .comm  _bit_map,49975Ø
        .set    WARP,2ØØØ
        .globl  _draw_line
        .globl  _sbitstab
        .align 4
_draw_line:
        enter    [r3,r4,r5,r6,r7],12
# T1
        movd     16(fp),r4      # xf
        movd     8(fp),r5       # xs
        subd     r5,r4          # delx
        movd     2Ø(fp),r6      # yf
        movd     12(fp),r7      # ys
        subd     r7,r6          # dely
        cmpqd    $(Ø),r4        # Ø>delx
        ble      .VERT
        movd     16(fp),r5      # xf=new xs
        movd     2Ø(fp),r7      # yf=new ys
        absd     r4,r4          # delx=|delx|
        negd     r6,r6          # dely=(-dely)
.VERT:
        movd     r7,r1          # ys
        muld     $WARP,r1       # ys*warp
        addd     r5,r1          # bit=ys*WARP+xs
        cmpqd    $(Ø),r4        # delx=Ø?
        bne      .HORZ
        cmpqd    $(Ø),r6        # dely>Ø?
        bgt      .VNEG          # if no then warp is neg
        addr     _bit_map,rØ    # set registers for sbitps
        movd     r6,r2          # r2=dely=length of line
        movd     $WARP,r3       # r3=warp
# T2
        sbitps                  # draw line
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.VNEG:
        addr     _bit_map,rØ    # set reg's for sbitps
        movd     r6,r2          # r2=(-dely)
        absd     r2,r2          # r2=dely=length of line
        movd     $(-WARP),r3    # r3=warp
# T3
        sbitps                  # draw line
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.HORZ:
        cmpqd    $(Ø),r6        # dely=Ø?
        bne      .DIAG
        addr     _bit_map,rØ    # set reg's for sbits
        movd     r4,r2          # r4=delx=length
        addr     _sbitstab,r3   # table pointer
# T4
        sbits                   # try sbits
        bfc      ok             # if not more than 25, skip it
        cmpd    $2ØØ,r2
        blt     bigs1
        addr    25,r2
        .align  4
alp1:   sbits
        addd    r2,r1
```

TL/EE/9663–11

```
        subd    r2,r4
        cmpd    r2,r4
        blt     alp1
        .align  4
        movd    r4,r2
        sbits
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
bigs1:  bsr      bigset
ok:     exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.DIAG:
        absd     r6,r5            # r5=|dely|
        cmpd     r5,r4            # |dely|=delx?
        bne      .SLOPELT1
        cmpqd    $(Ø),r6          # dely>Ø?
        bgt      .DNEG
        addr     _bit_map,rØ      # set reg's for sbitps
        movd     r4,r2            #  r2=delx=length
        movd     $WARP + 1,r3     # r3=warp+1 for diag
# T5
        sbitps                    # draw line
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.DNEG:
        addr     _bit_map,rØ      # set reg's for sbitps
        movd     r4,r2            # r2=delx=lenght
        movd     $-WARP + 1,r3    # r3=warp-1 for neg slope
# T6
        sbitps                    # draw line
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.SLOPELT1:                        # slope less than 1
        cmpd     r5,r4            # |dely|>delx?
        bgt      .SLOPEGT1
        movd     r4,r2            # r2=delx
        subd     r5,r2            # delx-|dely|
        cmpd     r5,r2            # |dely|>delx-|dely|?
        bgt      .OCTANT2         # if no, start octant1 else octant2
        cmpqd    $(Ø),r6          # dely>Ø?
        bgt      .NEGWARP
        addr     WARP,-4(fp)      # pos slope then warp=positive
        br       .INIT1
        .align 4
.NEGWARP:
        addr     -WARP,-4(fp)     # warp=negative for neg slope
.INIT1:                           # calculate parameters
        movd     r4,r3            # delx=dela  |dely|=delb
        quow     r5,r3            # dela/delb=q
        movd     r3,rØ            # calc m
        ashd     $-1,rØ           # m=q/2
        movd     r3,r2            # calc r
        mulw     r5,r2            # delb*q
        subd     r2,r4            # r=dela-delb*q
        movd     r4,r2            # set r2 = r
        tbitb    $Ø,r3            # is r3 odd?
        bfc      .INIT2           # yes, n = r
        addd     r5,r2            # n=r+delb
        .align 4
.INIT2:
        movd     r2,r7            # pop n
        movd     r3,tos           # push q on stack
        movd     rØ,r2            # r2=m=hØ
```

TL/EE/9663–12

```
        movd     rØ,-8(fp)       # mem=m=hpartb
        cmpqd    $(Ø),r7         # n=Ø?
        bne      .INIT3
        cmpqd    $(Ø),r6         # dely>Ø?
        blt      .INIT4
        addqd    $-1,r2          # hØ=m-1
        br       .INIT3
.INIT4:
        subd     $1,-8(fp)       # hpartb=m-1
.INIT3:
        addqd    $1,r2           # takes care of dashes
        addr     _bit_map,rØ     # set reg's for sbits
        addr     _sbitstab,r3    # hØ=r2 bit=r1
# T7
        sbits
        bfc      .2DONE          # set bits if less than 25
        cmpd     $2ØØ,r2
        blt      BIGSET1
        movd     r5,tos
        movd     r2,r5
        movd     $25,r2
.2DRAW25:
        subd     r2,r5
        sbits
        addd     r2,r1
        cmpd     r2,r5
        blt      .2DRAW25
        movd     r5,r2
        movd     tos,r5
        sbits
        br       .2DONE
BIGSET1:
        bsr      bigset
.2DONE:
# T8
        addd     r2,r1           # bit=bit+hØ+1
        addd     -4(fp),r1       # bit=bit+hØ+1+warp
        addd     r4,r4           # 2*r
        movd     r5,r3           # save delb
        addd     r5,r5           # delb*2
        addd     r4,r7           # n=n+2*r
        subd     r5,r7           # testvar=n+2*r+delb*2
        cmpqd    $(Ø),r6         # dely>Ø
        blt      .INIT5
        addqd    $-1,r7          # testvar-1
.INIT5:
        movd     tos,r2          # r2=q=h=run length
        addqd    $1,r2           # smoothes out line
        movd     r3,tos          # push delb=count
        addr     _sbitstab,r3    # set reg's for sbits
        addr     _bit_map,rØ
        movd     -4(fp),r6       # warp
        addqd    $-1,tos         # count=count-1
        cmpqd    $Ø,Ø(sp)        # count=Ø?
        bge      .LASTRUN
.MAINLOOP:                       # Bresenham slice algorithm
# T9
        cmpqd    $(Ø),r7         # testvar>Ø?
        ble      .CASE2
        addqd    $-1,r2          # h=q-1
        addd     r4,r7           # testvar=testvar+2*r
        sbits
        bfc      .3DRAWLAST      # set bits if less than 25
        cmpd     $2ØØ,r2
        blt      BIGSET3
        movd     r2,tos
```

TL/EE/9663–13

```
        movd    r5,tos
        movd    r2,r5
        movd    $25,r2
.3DRAW25:
        subd    r2,r5
        sbits
        addd    r2,r1
        cmpd    r2,r5
        blt     .3DRAW25
        movd    r5,r2
        sbits
        addd    r2,r1
        movd    tos,r5
        movd    tos,r2
        br      .3DONE
BIGSET3:
        bsr     bigset
.3DRAWLAST:
        addd     r2,r1          # update bit
.3DONE:
        addd     r6,r1          # bit=bit+warp+h+1
        addd     $1,r2          # exit h
        addqd    $(-1),tos      # count=count-1
        cmpqd    $(Ø),Ø(sp)     # count=Ø?
        blt      .MAINLOOP
        .align 4
.LASTRUN:
        cmpqd    $(Ø),tos       # pop stack
        movd     -8(fp),r2      # hpartb=last run length
        sbits
        bfc      .4DONE         # set bits if less than 25
        cmpd    $2ØØ,r2
        blt     BIGSET4
        movd    r2,tos
        movd    r5,tos
        movd    r2,r5
        movd    $25,r2
.4DRAW25:
        subd    r2,r5
        sbits
        addd    r2,r1
        cmpd    r2,r5
        blt     .4DRAW25
        movd    r5,r2
        sbits
        addd    r2,r1
        movd    tos,r5
        movd    tos,r2
        br      .4DONE
BIGSET4:
        bsr     bigset
.4DONE:
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.CASE2:
        addd     r4,r7          # testvar=testvar+2*r
        subd     r5,r7          # testvar=testvar+2*r-2*delb
        sbits
        bfc      .5DRAWLAST     # SET BITS IF LESS THAN 25
        cmpd    $2ØØ,r2
        blt     BIGSET5
        movd    r2,tos
        movd    r5,tos
        movd    r2,r5
        movd    $25,r2
```

TL/EE/9663–14

```
.5DRAW25:
        subd    r2,r5
        sbits
        addd    r2,r1
        cmpd    r2,r5
        blt     .5DRAW25
        movd    r5,r2
        sbits
        addd    r2,r1
        movd    tos,r5
        movd    tos,r2
        br      .5DONE
BIGSET5:
        bsr     bigset
.5DRAWLAST:
        addd     r2,r1          # update bit
.5DONE:
        addd     r6,r1          # bit=bit+warp+h+1
        addqd    $(-1),tos      # update count
        cmpqd    $(Ø),Ø(sp)     # count=Ø?
        blt      .MAINLOOP
        cmpqd    $(Ø),tos       # pop stack
        movd     -8(fp),r2      # hpartb=last run length
        sbits
        bfc      .6DONE         # set bits if less than 25
        bsr     bigset
.6DONE:
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.OCTANT2:                       # draw line in octant 2
        cmpqd    $(Ø),r6        # dely>Ø?
        bgt      .2NEGWARP
        addr     WARP,-4(fp)    # pos slope then warp=positive
        br       .2INIT1
.2NEGWARP:
        addr     -WARP,-4(fp)   # warp=negative for neg slope
.2INIT1:                        # calculate parameters
        movd     r4,r3          # dela=delx
        movd     r2,r5          # delb=delx-|dely|
        quow     r5,r3          # dela/delb=q
        movd     r3,rØ          # calc m
        ashd     $-1,rØ         # m=q/2
        movd     r3,r2          # calc r
        mulw     r5,r2          # delb*q
        subd     r2,r4          # r=dela-delb*q
        movd     r4,r2          # push r on stack
        tbitb    $Ø,r3
        bfc      .2INIT2        # then n=r
        addd     r5,r2          # n=r+delb
        .align 4
.2INIT2:
        movd     r2,r7          # pop n
        movd     r3,tos         # push q on stack
        movd     rØ,r2          # r2=m=hØ
        addqd    $1,r2          # set one extra bit for smoothness
        movd     rØ,-8(fp)      # mem=m=hpartb
        cmpqd    $(Ø),r7        # n=Ø?
        bne      .2INIT3
        cmpqd    $(Ø),r6        # dely>Ø?
        blt      .2INIT4
        subd     $1,r2          # hØ=m-1
        br       .2INIT3
.2INIT4:
        subd     $1,-8(fp)      # hpartb=m-1
.2INIT3:
```

TL/EE/9663–15

```
        addr      _bit_map,rØ     # set reg's for sbits
        movd      -4(fp),r3       # warp=r3 hØ=r2 bit=r1
        addqd     $1,r3           # octant 2 needs diag runs
# T1Ø
        sbitps                    # draw first run length
# T11
        addqd     $1,r1           # update bit in x direction
        subd      r3,r1           # sbitps adds extra warp
        addd      r4,r4           # 2*r
        movd      tos,r2          # q=h=next run length
        addqd     $1,r2           # set extra bit for smoothness
        movd      r5,tos          # push delb=count
        addd      r5,r5           # delb*2
        addd      r4,r7           # n=n+2*r
        subd      r5,r7           # testvar=n+2*r+delb*2
        cmpqd     $(Ø),r6         # dely>Ø
        blt       .2INIT5
        subd      $1,r7           # testvar-1
.2INIT5:
        subd      $1,tos          # count=count-1
        cmpqd     $Ø,Ø(sp)        # count=Ø?
        bge       .2LASTRUN
.2MAINLOOP:                       # Bresenham slice algorithm
# T12
        cmpqd     $(Ø),r7         # testvar>Ø?
        ble       .2CASE2
        subd      $1,r2           # h=q-1
        addd      r4,r7           # testvar=testvar+2*r
        movd      r2,tos          # preserve h
        sbitps                    # draw diag line of length h
        movd      tos,r2          # renew h
        addqd     $1,r1           # update bit in x direction
        subd      r3,r1           # sbitps adds one warp extra
        addd      $1,r2           # exit h to q
        subd      $1,tos          # count=count-1
        cmpqd     $Ø,Ø(sp)        # count=Ø?
        blt       .2MAINLOOP
        .align 4
.2LASTRUN:
        cmpqd     $(Ø),tos        # pop stack
        movd      -8(fp),r2       # hpartb=last run length
        sbitps                    # all other reg's set up
        exit      [r3,r4,r5,r6,r7]
        ret       $(Ø)
        .align 4
.2CASE2:
        addd      r4,r7           # testvar=testvar+2*r
        subd      r5,r7           # testvar=testvar+2*r-2*delb
        movd      r2,tos          # preserve h
        sbitps                    # draw line of length h=q
        movd      tos,r2          # renew h
        addqd     $1,r1           # update bit in x direction
        subd      r3,r1           # sbitps adds one warp extra
        subd      $(1),tos        # update count
        cmpqd     $Ø,Ø(sp)        # count=Ø?
        blt       .2MAINLOOP
        cmpqd     $(Ø),tos        # pop stack
        movd      -8(fp),r2       # hpartb=last run length
        sbitps                    # all other reg's set up
        exit      [r3,r4,r5,r6,r7]
        ret       $(Ø)
        .align 4
.SLOPEGT1:                        # coordinates are rotated for these lines
        movd      r5,r2           # r2=|dely|
        subd      r4,r2           # |dely|-delx
        cmpd      r4,r2           # delx>|dely|-delx?
```

TL/EE/9663–16

```
        bgt      .2OCTANT2       # if no, start octant1 else octant2
        cmpqd    $(Ø),r6         # dely>Ø?
        bgt      .3NEGWARP
        addr     WARP,-4(fp)     # pos slope then warp=positive
        br       .3INIT1
.3NEGWARP:
        addr     -WARP,-4(fp)    # warp=negative for neg slope
.3INIT1:                         # calculate rotated parameters
        movd     r5,r3           # dela=|dely|
        movd     r4,r5           # delb=delx
        movd     r3,r4           # dela in r4
        quow     r5,r3           # dela/delb=q
        movd     r3,rØ           # calc m
        ashd     $-1,rØ          # m=q/2
        movd     r3,r2           # calc r
        mulw     r5,r2           # delb*q
        subd     r2,r4           # r=dela-delb*q
        movd     r4,r2           # push r on stack
        tbitb    $Ø,r3
        bfc      .3INIT2         # then n=r
        addd     r5,r2           # n=r+delb
        .align 4
.3INIT2:
        movd     r2,r7           # pop n
        movd     r3,tos          # push q on stack
        movd     rØ,r2           # r2=m=hØ
        addqd    $1,r2           # set one extra bit for smoothness
        movd     rØ,-8(fp)       # mem=m=hpartb
        cmpqd    $(Ø),r7         # n=Ø?
        bne      .3INIT3
        cmpqd    $(Ø),r6         # dely>Ø?
        blt      .3INIT4
        subd     $1,r2           # hØ=m-1
        br       .3INIT3
.3INIT4:
        subd    $1,-8(fp)        # hpartb=m-1
.3INIT3:
        addr     _bit_map,rØ     # set reg's for sbits
        movd     -4(fp),r3       # warp=r3 hØ=r2 bit=r1
# T13
        sbitps                   # draw first run length
# T14
        addqd    $1,r1           # update bit in x direction
        addd     r4,r4           # 2*r
        movd     tos,r2          # q=h=next run length
        addqd    $1,r2           # set extra bit for smoothness
        movd     r5,tos          # push delb=count
        addd     r5,r5           # delb*2
        addd     r4,r7           # n=n+2*r
        subd     r5,r7           # testvar=n+2*r+delb*2
        cmpqd    $(Ø),r6         # dely>Ø
        blt      .3INIT5
        subd     $1,r7           # testvar-1
.3INIT5:
        subd     $1,tos          # count=count-1
        cmpqd    $Ø,Ø(sp)        # count=Ø?
        bge      .3LASTRUN
.3MAINLOOP:                      # Bresenham slice algorithm
# T15
        cmpqd    $(Ø),r7         # testvar>Ø?
        ble      .3CASE2
        subd     $1,r2           # h=q-1
        addd     r4,r7           # testvar=testvar+2*r
        movd     r2,tos          # preserve h
        sbitps                   # draw vert line of length h
        movd     tos,r2          # renew h
```

TL/EE/9663–17

```
        addqd    $1,r1            # update bit in x direction
        addd     $1,r2            # exit h to q
        subd     $1,tos           # count=count-1
        cmpqd    $Ø,Ø(sp)         # count=Ø?
        blt      .3MAINLOOP
        .align 4
.3LASTRUN:
        cmpqd    $(Ø),tos         # pop stack
        movd     -8(fp),r2        # hpartb=last run length
        sbitps                    # all other reg's set up
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.3CASE2:
        addd     r4,r7            # testvar=testvar+2*r
        subd     r5,r7            # testvar=testvar+2*r-2*delb
        movd     r2,tos           # preserve h
        sbitps                    # draw line of length h=q
        movd     tos,r2           # renew h
        addqd    $1,r1            # update bit in x direction
        subd     $(1),tos         # update count
        cmpqd    $Ø,Ø(sp)         # count=Ø?
        blt      .3MAINLOOP
        cmpqd    $(Ø),tos         # pop stack
        movd     -8(fp),r2        # hpartb=last run length
        sbitps                    # all other reg's set up
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.2OCTANT2:                        # draw line in octant 2
        cmpqd    $(Ø),r6          # dely>Ø?
        bgt      .4NEGWARP
        addr     WARP,-4(fp)      # pos slope then warp=positive
        br       .4INIT1
.4NEGWARP:
        addr     -WARP,-4(fp)     # warp=negative for neg slope
.4INIT1:                          # calculate parameters
        movd     r5,r3            # dela=delx
        movd     r5,r4            # dela into r4
        movd     r2,r5            # delb=delx-|dely|
        quow     r5,r3            # dela/delb=q
        movd     r3,rØ            # calc m
        ashd     $(-1),rØ         # m=q/2
        movd     r3,r2            # calc r
        mulw     r5,r2            # delb*q
        subd     r2,r4            # r=dela-delb*q
        movd     r4,r2            # push r on stack
        tbitb    $Ø,r3
        bfc      .4INIT2          # then n=r
        addd     r5,r2            # n=r+delb
        .align 4
.4INIT2:
        movd     r2,r7            # pop n
        movd     r3,tos           # push q on stack
        movd     rØ,r2            # r2=m=hØ
        addqd    $1,r2            # set one extra bit for smoothness
        movd     rØ,-8(fp)        # mem=m=hpartb
        cmpqd    $(Ø),r7          # n=Ø?
        bne      .4INIT3
        cmpqd    $(Ø),r6          # dely>Ø?
        blt      .4INIT4
        subd     $1,r2            # hØ=m-1
        br       .4INIT3
.4INIT4:
        subd     $1,-8(fp)        # hpartb=m-1
.4INIT3:
```

TL/EE/9663–18

```
        addr      _bit_map,rØ     # set reg's for sbits
        movd      -4(fp),r3       # warp=r3 hØ=r2 bit=r1
        addqd     $1,r3           # octant 2 needs diag runs
# T16
        sbitps                    # draw first run length
# T17
        subd      $1,r1           # update bit
        addd      r4,r4           # 2*r
        movd      tos,r2          # q=h=next run length
        addqd     $1,r2           # set extra bit for smoothness
        movd      r5,tos          # push delb=count
        addd      r5,r5           # delb*2
        addd      r4,r7           # n=n+2*r
        subd      r5,r7           # testvar=n+2*r+delb*2
        cmpqd     $(Ø),r6         # dely>Ø
        blt       .4INIT5
        subd      $1,r7           # testvar-1
.4INIT5:
        subd      $1,tos          # count=count-1
        cmpqd     $Ø,Ø(sp)        # count=Ø?
        bge       .4LASTRUN
.4MAINLOOP:                       # Bresenham slice algorithm
# T18
        cmpqd     $(Ø),r7         # testvar>Ø?
        ble       .4CASE2
        subd      $1,r2           # h=q-1
        addd      r4,r7           # testvar=testvar+2*r
        movd      r2,tos          # preserve h
        sbitps                    # draw diag line of length h
        movd      tos,r2          # renew h
        subd      $1,r1           # sbitps adds one warp extra
        addd      $1,r2           # exit h to q
        subd      $1,tos          # count=count-1
        cmpqd     $Ø,Ø(sp)        # count=Ø?
        blt       .4MAINLOOP
        .align 4
.4LASTRUN:
        cmpqd     $(Ø),tos        # pop stack
        movd      -8(fp),r2       # hpartb=last run length
        addqd     $1,r2
        sbitps                    # all other reg's set up
        exit      [r3,r4,r5,r6,r7]
        ret       $(Ø)
        .align 4
.4CASE2:
        addd      r4,r7           # testvar=testvar+2*r
        subd      r5,r7           # testvar=testvar+2*r-2*delb
        movd      r2,tos          # preserve h
        sbitps                    # draw line of length h=q
        movd      tos,r2          # renew h
        subd      $1,r1           # sbitps adds one warp extra
        subd      $(1),tos        # update count
        cmpqd     $Ø,Ø(sp)        # count=Ø?
        blt       .4MAINLOOP
        .align 4
        cmpqd     $(Ø),tos        # pop stack
        movd      -8(fp),r2       # hpartb=last run length
        addqd     $1,r2
        sbitps                    # all other reg's set up
        exit      [r3,r4,r5,r6,r7]
        ret       $(Ø)
```

TL/EE/9663–19

```
# BIGSET.S uses MOVMP and the OR instructions to set long horizontal lines
#
        .globl  bigset
bigset: save    [rØ,r1,r2,r3,r4,r5,r6]  #save registers we will affect
        movd    r1,r4                   #get current bit offset
        ashd    $-3,r4                  #divide by eight to get byte offset
        addd    r4,rØ                   #add in base. rØ is new base pointer
        andd    $7,r1                   #mask off msb's of bit pointer to
                                        #get bit = bit offset mod 8

 #Now we have true base address and bit offset within base. Now we will move
 #to double word alignment. This speeds up the MOVMPD for long bit sequences.

        movqd   3,r4                    #place mask in r4
        andd    rØ,r4                   #get low two bits of address
        xorb    $3,r4                   #and get bytes left to alignment
        addqd   1,r4                    #rem += 1 (for the byte we are on)
        ashd    $3,r4                   #rem *= 8 to get bits to alignment
        subd    r1,r4                   #subtract current bit offset
        cmpd    r4,r2                   #is this more than number of bits left
        bge     shrt                    #it is, do it the short way
        cmpd    $32,r4                  #if we are already double aligned, go
                                        #do the MOVMPD
        beq     mvm
        movd    r1,r5                   #calculate index into table
        lshd    $5,r5                   #index = 32 * bit offset
        addd    r4,r5                   #index += run length
        ord     r3[r5:d],Ø(rØ)          #or in required bits
        bicb    $3,rØ                   #clear last two bits, and
        addqd   4,rØ                    #bump to next double
        subd    r4,r2                   #zap sp'd bits off
mvm:    movd    r2,r4                   #save run length for a minute
        movd    r3,r5                   #and save pointer to table
        ashd    $-5,r2                  #r1 = r1 / 32 = number of doubles
        movd    1Ø2Ø(r3),r3             #get source pattern from table
        movqd   4,r1                    #increment is r1
        movmpd                          # yes, use instruction
        andd    $Øx1f,r4                #mask off all but last 32 bits
        ord     r5[r4:d],Ø(rØ)          #insert the last few bits
        restore [rØ,r1,r2,r3,r4,r5,r6]  #restore saved registers
        ret     $Ø

        .align  4
shrt:   cmpb    $32,r2                  #check to see if it is exactly
        beq     shrt1                   #32 bits. If it is, branch.
        movd    r1,r4                   #calculate index into table
        lshd    $5,r4                   #index = 32 * bit offset
        addd    r2,r4                   #index += run length
        ord     r3[r4:d],Ø(rØ)          #or in required bits
        restore [rØ,r1,r2,r3,r4,r5,r6]  #restore saved registers
        ret     $Ø
        .align  4
shrt1:  movd    1Ø2Ø(r3),Ø(rØ)          #copy last entry of table
        restore [rØ,r1,r2,r3,r4,r5,r6]  #(all 32 bits) and restore
        ret     $Ø
```

TL/EE/9663–20

```
/* 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)
                        draw_line(Ø,i,maxx,maxy-i);
                for (i=Ø;i<=maxx;i+=25)
                        draw_line(i,maxy,maxx-i,Ø);
        }

}
```

TL/EE/9663–21

```
/* 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/9663–22

Lit. # 100522

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

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