Drawing Circles with the NS32CG16; NS32CG16 Note 1

National Semiconductor
Application Note 523
Dave Rand
May 1988

1.0 INTRODUCTION

The NS32CG16 is a 32-bit CMOS, graphics oriented processor. It is software compatible with other Series 32000® CPUs, with new instructions for high-speed graphics. The NS32CG16 is designed specifically for page-oriented printing technologies such as laser, LCS, LED, Ion-Deposition, and Ink Jet.

In this applications note, a method for high-speed circle generation will be described, using an optimized version of Bresenham's circle algorithm.

2.0 DESCRIPTION

A circle can be described by the center coordinates (xc, yc), the radius (r), and the width (w). With the Pythagorean theorem, pixels along the path described by the equation:

$$(x - xc)^2 + (y - yc)^2 = r^2$$

can be set for a width of w perpendicular to the tangent of the arc.

This, however, involves substantial computation for each point on the line. Even taking advantage of the symmetry of circles, a large number of instructions must be executed to calculate the path.

Bresenham's circle algorithm works by determining which of two pixels are nearer the actual circle at each step. Then, using symmetry, eight points on the circle's path can be determined. Applying the width (w) to each of these eight points yields a displayed (or imaged) circle. For the actual derivation of Bresenham's algorithm, see *Reference 1*, and *Reference 2*. This derivation was done by J. Michener.

Bresenham's algorithm can be implemented in the following manner:

1. Select the first position for display as

   $$(x_1, y_1) = (0,r)$$

2. Calculate the first parameter as

   $$p_1 = 3 - 2r$$

   If $p_1 < 0$, the next position is $(x_1 + 1, y_1)$. Otherwise, the next position is $(x_1 + 1, y_1 - 1)$.

3. Continue to increment the x coordinate by unit steps, and calculate each succeeding parameter p from the preceding one. If for the previous parameter we found that $p_i < 0$ then

   $$p_{i+1} = p_i + 4x_i + 6$$

   Otherwise (for $p_i \geq 0$),

   $$p_{i+1} = p_i + 4(x_i - y_i) + 10$$

   Then, if $p_{i+1} < 0$ the next point selected is $(x_i + 2, y_{i+1})$. Otherwise, the next point is $(x_i + 2, y_{i+1} - 1)$. The y coordinate is $y_{i+1} = y_i$, if $p_i < 0$ or $y_{i+1} = y_i - 1$, if $p_i \geq 0$.

4. Repeat the procedures in step 3 until the x and y coordinates are equal.

3.0 IMPLEMENTATION

With the path of the circle described, the pixels along the path can be set using the basic symmetry of the circle. Following is an example of Bresenham's circle algorithm in the C language, based on Michener's derivation.

```
circle(xc,yc,radius,width)
register unsigned int xc,yc,radius,width;
{
    register int y, x, p;
    x = 0;
    y = radius;
    p = 3 - 2 * radius;
    while (x < y) {
        setgrp(xc,yc,x,y,width);
        if (p < 0)
            p += 4 * x + 6;
        else {
            p += 4 * (x - y) + 10;
            y--;
        }
        x++;
    }
    if (y == x)
        setgrp(xc,yc,x,y,width);
}
```

TL/EE/9664–1

```
setgrp(xc,yc,x,y,width)
register int xc,yc,x,y,width;
{
    if ((y - x) <= (width / 2) {
        hset(xc + y, yc + x,width);
        hset(xc - y, yc + x,width);
        hset(xc + y, yc - x,width);
        hset(xc - y, yc - x,width);
        vset(xc + x, yc + y,width);
        vset(xc - x, yc + y,width);
        vset(xc + x, yc - y,width);
        vset(xc - x, yc - y,width);
    }
    vset(xc + y, yc + x,width);
    vset(xc - y, yc + x,width);
    vset(xc + y, yc - x,width);
    vset(xc - y, yc - x,width);
    hset(xc + x, yc + y,width);
    hset(xc - x, yc + y,width);
    hset(xc + x, yc - y,width);
    hset(xc - x, yc - y,width);
}
```

TL/EE/9664–2

The *setgrp* routine in the previous example uses symmetry to set eight points of the circle. *Setgrp* has a special case to handle the boundaries of the eight sections. When the distance between the boundaries is less than half the width of the circle, both vertical and horizontal lines are imaged for each section. The *vset* routine sets *width* pixels vertically in the image, centered around the second argument. The *hset* routine sets *width* pixels horizontally, centered around the first argument. Since these cases are so well defined, the NS32CG16 instructions *SBITPS* and *SBITS* are used for these routines.

The NS32CG16 implementation is very much like the C version, but is optimized for speed. Note the use of the *ADDR* instruction to do the two p_i computations, each in one line of 32000 assembly code.

RRD-B30M105/Printed in U. S. A.

```
        .data
xwarp:  equ     2544            #bits of xwarp to get to next scan
        .comm   _page,4
hlfwdth:double  0
        .text
#
#Bresenham's circle algorithm, as expressed in "Computer Graphics" by
#Donald Hearn and M. Pauline Baker (1986, Prentice-Hall,
#ISBN 0-13-165382-2)
#
#       Inputs:
#               r0 = x coodinate of centre of circle
#               r1 = y coodinate of centre of circle
#               r2 = width (in pixels)
#               r3 = radius (in pixels)
#
#       Outputs:
```

TL/EE/9664–3

```
#               no registers altered
#               circle drawn in ram
#
#Notes:
#       This routine uses two special case line drawing routines:
#               a horizontal case (called HLINE)
#               a vertical case (called VLINE)
#       A general purpose line drawing algorithm could be used, however
#       the new 32CG16 instructions are much faster.
#       If the line is to have a width of > 25 pixels, the BIGSET algorithm
#       must be added to the HLINE routine. No other changes are required.
#
circle: save    [r4,r5,r6,r7]   #save our working registers
        movd    r2,r7           #get current width
        lshd    $-1,r7          #divide by two
        movd    r7,hlfwdth      #and store it away
        movqd   0,r4            #x1 = 0
        movd    r3,r5           #y1 = radius
        movqd   3,r6            #p = 3 - (radius * 2)
        subd    r3,r6
        subd    r3,r6
        br      cirtest
        .align  4
cirlp:  bsr     setgrp          #set a group of points
        cmpqd   0,r6            #is P less than zero?
        blt     pge0            #no, it is not. skip
        addr    6(r6)[r4:d],r6  #p += 4 * x1 + 6
        addqd   1,r4            #x1 ++
cirtest:cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
        br      cirotl

        .align  4
pge0:   movd    r4,r7           #t = x1
        subd    r5,r7           #t = x1 - y1
        addr    10(r6)[r7:d],r6 #p += 4 * (x1 - y1) + 10
        addqd   -1,r5           #y1 --
        addqd   1,r4            #x1 ++
        cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
cirotl: bne     cirout          #if x1 != y1, get out
        bsr     setgrp          #else set last group
cirout: restore [r4,r5,r6,r7]   #restore working registers
        ret     0               #and return




#
#Setgrp sets eight points on a circle, given starting x and y, and the
#current xoffset and y offset.
#
#       Inputs:
#               r0 = centerpoint of circle (x coodinate)
```

TL/EE/9664–4

```
#		r1 = centerpoint of circle (y coodinate)
#		r2 = line width
#		r4 = x offset
#		r5 = y offset
#
#	Ouputs:
#		all registers preserved.
#
	.align	4
setgrp:	movd	r6,tos		#get two temporary values
	movd	r7,tos
	movd	r0,r6		#save old x
	movd	r1,r7		#and y
	movd	r5,r1
	subd	r4,r1		#r1 = (y1 - x1)
	cmpd	r1,hlfwdth	#if the difference is less than
	ble	sg1:w		#half the width, fill in the edges
	movd	r7,r1		#restore y
	addd	r4,r0		#x += x1
	addd	r5,r1		#y += y1
	bsr	vline		#do a vline
	movd	r6,r0		#restore x and y
	movd	r7,r1
	addd	r4,r0		#x += x1
	subd	r5,r1		#y -= y1
	bsr	vline
	movd	r6,r0		#restore x and y
	movd	r7,r1
	subd	r4,r0		#x -= x1
	addd	r5,r1		#y += y1
	bsr	vline
	movd	r6,r0		#restore x and y
	movd	r7,r1
	subd	r4,r0		#x -= x1
	subd	r5,r1		#y -= y1
	bsr	vline

	movd	r6,r0		#restore x and y
	movd	r7,r1
	addd	r5,r0		#x += y1
	addd	r4,r1		#y += x1
	bsr	hline
	movd	r6,r0		#restore x and y
	movd	r7,r1
	addd	r5,r0		#x += y1
	subd	r4,r1		#y -= x1
	bsr	hline
	movd	r6,r0		#restore x and y
	movd	r7,r1
	subd	r5,r0		#x -= y1
	addd	r4,r1		#y += x1
	bsr	hline
	movd	r6,r0		#restore x and y
	movd	r7,r1
	subd	r5,r0		#x -= y1
```

TL/EE/9664–5

```
        subd    r4,r1           #y -= x1
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        movd    tos,r7          #and unstack
        movd    tos,r6
        ret     0

sg1:    movd    r7,r1           #restore y
        addd    r4,r0           #x += x1
        addd    r5,r1           #y += y1
        bsr     hline           #do a hline
        bsr     vline           #and a vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r4,r0           #x += x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        addd    r5,r1           #y += y1
        bsr     hline
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline

        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        subd    r4,r1           #y -= x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
        subd    r4,r1           #y -= x1
        bsr     vline
```

TL/EE/9664–6

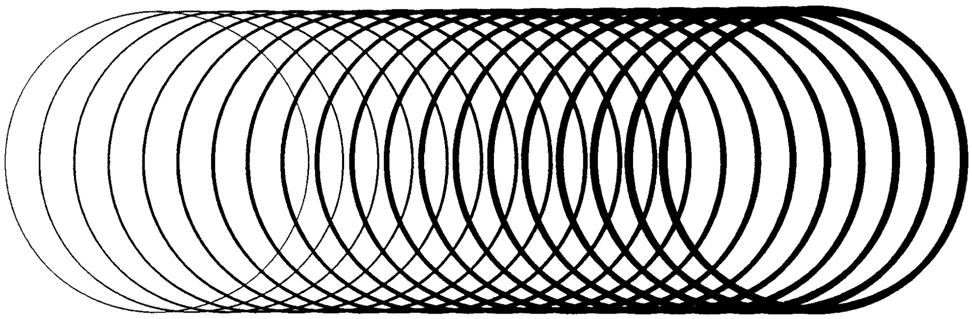
```
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        movd    tos,r7          #and unstack
        movd    tos,r6
        ret     0

#
#A vertical line drawing algorithm, making use of the SBITPS instruction.
#
#       Inputs:
#               r0 = x coodinate of line
#               r1 = centerpoint of y coordinate of line
#               r2 = line length
#
#       Outputs:
#               no registers altered.
#               line drawn in memory.
#
        .align  4
vline:  save    [r0,r1,r2,r3]   #save working registers
        subd    hlfwdth,r1      #y -= half of width to centre vline
        addr    @(xwarp-1),r3   #r3 = xwarp -1
        indexd  r1,r3,r0        #bit off = y * (xwarp) + x
        addqd   1,r3            #move to correct warp value
        movd    _page,r0        #page address in r0
        SBITPS                  #set bit perpendicular string
        restore [r0,r1,r2,r3]   #restore registers
        ret     0

#
#A horizontal line drawing algorithm, using SBITS.
#
#       Inputs:
#               r0 = centerpoint of x coordinate
#               r1 = y coodinate of line
#               r2 = line length
#
        .align  4
hline:  save    [r0,r1,r3]      #save working registers
        subd    hlfwdth,r0      #x -= half of width to centre values
        indexd  r1,(xwarp - 1),r0 # bit off = (y * xwarp) + x
        movd    _page,r0        #page address in r0
        addr    stab,r3         #address of sbits table
        SBITS
        restore [r0,r1,r3]
        ret     0
```

TL/EE/9664–7

Figure 1 shows this algorithm 'at work'. 20 circles of radius 350 pixels, and widths of 1 to 20 pixels are shown. A full listing of this test program is shown in *Figure 2*.

4.0 TIMING

The execution speed of this algorithm is dependent on the radius of the circle, and the circle's width. The test program supplied executes in 2.92 seconds on a NS32016 at 10 MHz with no wait states. The execution time on the NS32CG16 at 15 MHz with no wait states is 1.54 seconds. By using macros for the VLINE and HLINE routines, instead of subroutine calls, the time can be further reduced to 1.39 seconds.



TL/EE/9664–8

FIGURE 1

```
        .data
        .set    xwarp,2544      #bits of xwarp to get to next scan
        .comm   _page,4
hlfwdth:.double 0
        .text
#
# Test is a C - callable function that creates Figure 1.
#
        .globl  _test
_test:  save    [r3,r4,r5,r6,r7]
        addr    @400,r0         #start at x=400
        addr    @400,r1         #         y=400
        movqd   1,r2            #width = 1
        addr    @350,r3         #radius = 350
        addr    @20,r7          #we want to do 20 circles
lp:     bsr     circle          #do a circle
        addr    80(r0),r0       #x += 80
        addqd   1,r2            #width += 1
        acbd    -1,r7,lp        #loop for all 20 circles
        restore [r3,r4,r5,r6,r7]
        ret     0               #and return

#
#Bresenham's circle algorithm, as expressed in "Computer Graphics" by
#Donald Hearn and M. Pauline Baker (1986, Prentice-Hall,
#ISBN 0-13-165382-2)
#
#       Inputs:
#               r0 = x coodinate of centre of circle
#               r1 = y coodinate of centre of circle
#               r2 = width (in pixels)
#               r3 = radius (in pixels)
#
#       Outputs:
#               no registers altered
#               circle drawn in ram
#
#Notes:
#       This routine uses two special case line drawing routines:
#               a horizontal case (called HLINE)
#               a vertical case (called VLINE)
#       A general purpose line drawing algorithm could be used, however
#       the new 32CG16 instructions are much faster.
#       If the line is to have a width of > 25 pixels, the BIGSET algorithm
#       must be added to the HLINE routine. No other changes are required.
#
circle: save    [r4,r5,r6,r7]   #save our working registers
        movd    r2,r7           #get current width
```

TL/EE/9664–9

FIGURE 2

```
        lshd    $-1,r7          #divide by two
        movd    r7,hlfwdth      #and store it away
        movqd   0,r4            #x1 = 0
        movd    r3,r5           #y1 = radius
        movqd   3,r6            #p = 3 - (radius * 2)
        subd    r3,r6
        subd    r3,r6
        br      cirtest
        .align  4
cirlp:  bsr     setgrp          #set a group of points
        cmpqd   0,r6            #is P less than zero?
        blt     pge0            #no, it is not. skip
        addr    6(r6)[r4:d],r6  #p += 4 * x1 + 6
        addqd   1,r4            #x1 ++
cirtest:cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
        br      cirout

        .align  4
pge0:   movd    r4,r7           #t = x1
        subd    r5,r7           #t = x1 - y1
        addr    10(r6)[r7:d],r6 #p += 4 * (x1 - y1) + 10
        addqd   -1,r5           #y1 --
        addqd   1,r4            #x1 ++
        cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
cirout: restore [r4,r5,r6,r7]   #restore working registers
        ret     0               #and return


#
#Setgrp sets eight points on a circle, given starting x and y, and the
#current xoffset and y offset.
#
#       Inputs:
#               r0 = centerpoint of circle (x coodinate)
#               r1 = centerpoint of circle (y coodinate)
#               r2 = line width
#               r4 = x offset
#               r5 = y offset
#
#       Ouputs:
#               all registers preserved.
#
        .align  4
setgrp: movd    r6,tos          #get two temporary values
        movd    r7,tos
        movd    r0,r6           #save old x
```

TL/EE/9664–10

FIGURE 2 (Continued)

```
movd    r1,r7        #and y
movd    r5,r1
subd    r4,r1        #r1 = (y1 - x1)
cmpd    r1,hlfwdth   #if the difference is less than
ble     sg1          #half the width, fill in the edges
movd    r7,r1        #restore y
addd    r4,r0        #x += x1
addd    r5,r1        #y += y1
bsr     vline        #do a vline
movd    r6,r0        #restore x and y
movd    r7,r1
addd    r4,r0        #x += x1
subd    r5,r1        #y -= y1
bsr     vline
movd    r6,r0        #restore x and y
movd    r7,r1
subd    r4,r0        #x -= x1
addd    r5,r1        #y += y1
bsr     vline
movd    r6,r0        #restore x and y
movd    r7,r1
subd    r4,r0        #x -= x1
subd    r5,r1        #y -= y1
bsr     vline

movd    r6,r0        #restore x and y
movd    r7,r1
addd    r5,r0        #x += y1
addd    r4,r1        #y += x1
bsr     hline
movd    r6,r0        #restore x and y
movd    r7,r1
addd    r5,r0        #x += y1
subd    r4,r1        #y -= x1
bsr     hline
movd    r6,r0        #restore x and y
movd    r7,r1
subd    r5,r0        #x -= y1
addd    r4,r1        #y += x1
bsr     hline
movd    r6,r0        #restore x and y
movd    r7,r1
subd    r5,r0        #x -= y1
subd    r4,r1        #y -= x1
bsr     hline
movd    r6,r0        #restore x and y
movd    r7,r1
movd    tos,r7       #and unstack
```

TL/EE/9664–11

FIGURE 2 (Continued)

```
        movd    tos,r6
        ret     0

sg1:    movd    r7,r1           #restore y
        addd    r4,r0           #x += x1
        addd    r5,r1           #y += y1
        bsr     hline           #do a hline
        bsr     vline           #and a vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r4,r0           #x += x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        addd    r5,r1           #y += y1
        bsr     hline
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline

        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        subd    r4,r1           #y -= x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
```

TL/EE/9664–12

FIGURE 2 (Continued)

```
        subd    r4,r1           #y -= x1
        bsr     vline
        bsr     hline
        movd    r6,r∅           #restore x and y
        movd    r7,r1
        movd    tos,r7          #and unstack
        movd    tos,r6
        ret     ∅

#
#A vertical line drawing algorithm, making use of the SBITPS instruction.
#
#       Inputs:
#               r∅ = x coodinate of line
#               r1 = centerpoint of y coordinate of line
#               r2 = line length
#
#       Outputs:
#               no registers altered.
#               line drawn in memory.
#
        .align  4
vline:  save    [r∅,r1,r2,r3]   #save working registers
        subd    hlfwdth,r1      #y -= half of width to centre vline
        addr    @(xwarp-1),r3   #r3 = xwarp -1
        indexd  r1,r3,r∅        #bit off = y * (xwarp) + x
        addqd   1,r3            #move to correct warp value
        movd    _page,r∅        #page address in r∅
#       SBITPS                  #set bit perpendicular string

# - Start of SBITPS emulation code
        .align  4
sblp:   sbitd   r1,∅(r∅)        #set required bit
        addd    r3,r1           #add the bit warp
        acbd    -1,r2,sblp      #loop for the rll
# - End of SBITPS emulation code
        restore [r∅,r1,r2,r3]   #restore registers
        ret     ∅

#
#A horizontal line drawing algorithm, using SBITS.
#
#       Inputs:
#               r∅ = centerpoint of x coordinate
#               r1 = y coodinate of line
#               r2 = line length
#
        .align  4
```

TL/EE/9664–13

FIGURE 2 (Continued)

```
hline:  save    [r0,r1,r3]      #save working registers
        subd    hlfwdth,r0      #x -= half of width to centre values
        indexd  r1,$(xwarp - 1),r0 # bit off = (y * xwarp) + x
        movd    _page,r0        #page address in r0
#       addr    stab,r3         #address of sbits table
#       SBITS

# - start of SBITS emulation code
        movqd   7,r3
        andd    r1,r3

        addd    r3,r3   #* 2
        addd    r3,r3   #* 4
        addd    r3,r3   #* 8
        addd    r3,r3   #* 16
        addd    r3,r3   #* 32
        addd    r2,r3
        ashd    $-3,r1
        ord     stab[r3:d],0(r0)[r1:b]
# - end of SBITS emulation code
        restore [r0,r1,r3]
        ret     0


        .data
stab:   .double h'00000000,h'00000001,h'00000003,h'00000007
        .double h'0000000f,h'0000001f,h'0000003f,h'0000007f
        .double h'000000ff,h'000001ff,h'000003ff,h'000007ff
        .double h'00000fff,h'00001fff,h'00003fff,h'00007fff
        .double h'0000ffff,h'0001ffff,h'0003ffff,h'0007ffff
        .double h'000fffff,h'001fffff,h'003fffff,h'007fffff
        .double h'00ffffff,h'01ffffff,h'03ffffff,h'07ffffff
        .double h'0fffffff,h'1fffffff,h'3fffffff,h'7fffffff
        .double h'00000000,h'00000002,h'00000006,h'0000000e
        .double h'0000001e,h'0000003e,h'0000007e,h'000000fe
        .double h'000001fe,h'000003fe,h'000007fe,h'00000ffe
        .double h'00001ffe,h'00003ffe,h'00007ffe,h'0000fffe
        .double h'0001fffe,h'0003fffe,h'0007fffe,h'000ffffe
        .double h'001ffffe,h'003ffffe,h'007ffffe,h'00fffffe
        .double h'01fffffe,h'03fffffe,h'07fffffe,h'0ffffffe
        .double h'1ffffffe,h'3ffffffe,h'7ffffffe,h'fffffffe
        .double h'00000000,h'00000004,h'0000000c,h'0000001c
        .double h'0000003c,h'0000007c,h'000000fc,h'000001fc
        .double h'000003fc,h'000007fc,h'00000ffc,h'00001ffc
        .double h'00003ffc,h'00007ffc,h'0000fffc,h'0001fffc
        .double h'0003fffc,h'0007fffc,h'000ffffc,h'001ffffc
        .double h'003ffffc,h'007ffffc,h'00fffffc,h'01fffffc
        .double h'03fffffc,h'07fffffc,h'0ffffffc,h'1ffffffc
```

TL/EE/9664–14

FIGURE 2 (Continued)

```
.double h'3ffffffc,h'7ffffffc,h'fffffffc,h'fffffffc
.double h'00000000,h'00000008,h'00000018,h'00000038
.double h'00000078,h'000000f8,h'000001f8,h'000003f8
.double h'000007f8,h'00000ff8,h'00001ff8,h'00003ff8
.double h'00007ff8,h'0000fff8,h'0001fff8,h'0003fff8
.double h'0007fff8,h'000ffff8,h'001ffff8,h'003ffff8
.double h'007ffff8,h'00fffff8,h'01fffff8,h'03fffff8
.double h'07fffff8,h'0ffffff8,h'1ffffff8,h'3ffffff8
.double h'7ffffff8,h'fffffff8,h'fffffff8,h'fffffff8
.double h'00000000,h'00000010,h'00000030,h'00000070
.double h'000000f0,h'000001f0,h'000003f0,h'000007f0
.double h'00000ff0,h'00001ff0,h'00003ff0,h'00007ff0
.double h'0000fff0,h'0001fff0,h'0003fff0,h'0007fff0
.double h'000ffff0,h'001ffff0,h'003ffff0,h'007ffff0
.double h'00fffff0,h'01fffff0,h'03fffff0,h'07fffff0
.double h'0ffffff0,h'1ffffff0,h'3ffffff0,h'7ffffff0
.double h'fffffff0,h'fffffff0,h'fffffff0,h'fffffff0
.double h'00000000,h'00000020,h'00000060,h'000000e0
.double h'000001e0,h'000003e0,h'000007e0,h'00000fe0
.double h'00001fe0,h'00003fe0,h'00007fe0,h'0000ffe0
.double h'0001ffe0,h'0003ffe0,h'0007ffe0,h'000fffe0
.double h'001fffe0,h'003fffe0,h'007fffe0,h'00ffffe0
.double h'01ffffe0,h'03ffffe0,h'07ffffe0,h'0fffffe0
.double h'1fffffe0,h'3fffffe0,h'7fffffe0,h'ffffffe0
.double h'ffffffe0,h'ffffffe0,h'ffffffe0,h'ffffffe0
.double h'00000000,h'00000040,h'000000c0,h'000001c0
.double h'000003c0,h'000007c0,h'00000fc0,h'00001fc0
.double h'00003fc0,h'00007fc0,h'0000ffc0,h'0001ffc0
.double h'0003ffc0,h'0007ffc0,h'000fffc0,h'001fffc0
.double h'003fffc0,h'007fffc0,h'00ffffc0,h'01ffffc0
.double h'03ffffc0,h'07ffffc0,h'0fffffc0,h'1fffffc0
.double h'3fffffc0,h'7fffffc0,h'ffffffc0,h'ffffffc0
.double h'ffffffc0,h'ffffffc0,h'ffffffc0,h'ffffffc0
.double h'00000000,h'00000080,h'00000180,h'00000380
.double h'00000780,h'00000f80,h'00001f80,h'00003f80
.double h'00007f80,h'0000ff80,h'0001ff80,h'0003ff80
.double h'0007ff80,h'000fff80,h'001fff80,h'003fff80
.double h'007fff80,h'00ffff80,h'01ffff80,h'03ffff80
.double h'07ffff80,h'0fffff80,h'1fffff80,h'3fffff80
.double h'7fffff80,h'ffffff80,h'ffffff80,h'ffffff80
.double h'ffffff80,h'ffffff80,h'ffffff80,h'ffffffff
```

TL/EE/9664–15

FIGURE 2 (Continued)

5.0 CONCLUSIONS

The NS32CG16 provides several instructions that increase the speed of imaging common graphic items such as circles, lines, and ellipses. The NS32CG16's high code density, and fast execution, make it ideal for intensive graphics processing.

This algorithm does, however, show an apparent 'thinning' on the 45° boundaries, when the width of the circle is greater than five pixels. An alternate algorithm will be presented in a future applications note. This algorithm is optimized for speed.

6.0 REFERENCES

1. Hearn, Donald and M. Pauline Baker, (1986). *Computer Graphics*, Englewood Cliffs, N.J., Prentice-Hall, 65–69.

2. Foley, James D. and Van Dam, Andries, (1982). *Fundamentals of Interactive Computer Graphics*, Reading, Massachusetts, Addison-Wesley, 441–446.

Lit. # 100523

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