Application Note

AN2621/D
Rev. 0, 11/2003

MPC8220i PF300 Image Coprocessor Operation

Andy Masia
32-bit Embedded Controller Division

The MPC8220i contains an integrated Image Coprocessor called PF300 that can be programmed to accelerate many common color and monochrome image-processing tasks. The PF300 functions can be invoked from page description language interpreters, by firmware that implements copy pipelines for MFP (AIO) devices or digital copiers, or from other image processing applications.

This application note provides an overview of each of the image processing algorithms performed by the PF300. It is addressed to algorithm developers, color and imaging scientists, and printer controller applications developers. More detail can be found in Chapter 32 of the *MPC8220i Microcontroller Preliminary Reference Manual* (MPC8220IRM/D, rev. 1.19).

In order to simplify the explanation in this application note, the following two assumptions have been made: (1) data input to the pipeline is RGB raster data that is stored color and pixel interleaved in contiguous memory, 8 bits per pixel per component, and (2) the color space transformation is from RGB to CMYK. This is a very common mode of operation but the PF300 Image Coprocessor is highly configurable and can be programmed to process data that is not contiguous in memory and has other than three color components for input and other than four components for output as assumed in this discussion.

The image processing functions implemented in the PF300 are: image resampling (geometric transformation), color space conversion, and halftone screening. These functions can be configured into a data processing pipeline that operates at the speed of the 120-MHz MPC8220i memory bus. This speed is equivalent to 214 monochrome or 53 color pages per minute at 600 dpi if the data processing pipeline runs without interruption. Due to the depth of the pipeline, the PF300 is most suited to processing blocks of raster data. The PF300 can be configured to process monochrome or color data, and each of the elements of the pipeline can be enabled or disabled separately.

A data processing flow chart of the PF300 Image Coprocessor is shown in Figure 1.

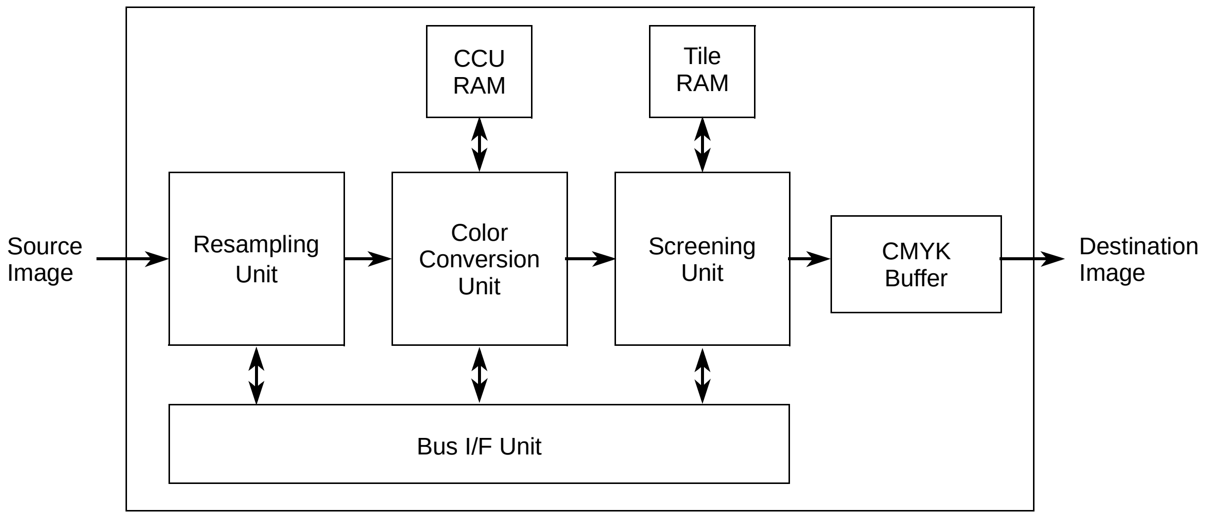


Figure 1. PF300 Image Coprocessor Data Flow

The PF300 is controlled by registers and makes use of both dedicated and system memory in its operation.

1 The Resampling Unit

1.1 Algorithm Description

The Resampling Unit (RU) operates on the locations (addresses) of pixels in the two-dimensional source and destination reference coordinate systems. The RU performs a linear transformation of output (destination) pixel coordinates to compute the required coordinates of the nearest neighbor input (source) pixel. In general, this allows mapping of a rectangular source image to a parallelogram destination image. The destination image can be rotated to an arbitrary angle and shifted by an arbitrary number of pixels with respect to the source image coordinate system. The RU also generates a “shadow” image that is a 1-bit per destination pixel mask indicating whether or not the corresponding destination pixel is valid (that is, whether the associated source pixel was inside or outside of the source image). Most applications will substitute white space or the current background color for destination pixel data that is not valid.

Algebraically, the coordinate transform can be expressed by two simultaneous equations as shown below.

$$\mathrm{Xdest} = a \times \mathrm{Xsrc} + b \times \mathrm{Ysrc} + tx$$
$$\mathrm{Ydest} = c \times \mathrm{Xsrc} + d \times \mathrm{Ysrc} + ty$$

Or, in matrix notation,

$$[\mathrm{Xdest} \quad \mathrm{Ydest} \quad 1] = [\mathrm{Xsrc} \quad \mathrm{Ysrc} \quad 1]\begin{bmatrix} a & c & 0 \\ b & d & 0 \\ tx & ty & 1 \end{bmatrix}$$

In these equations, Xdest and Ydest are the pixel coordinates of the destination image, and Xsrc and Ysrc are the corresponding coordinates in the source image. a, b, tx, c, d, and ty are constants. tx and ty represent translation (shift) in the x and y directions, respectively.

1.2 Computational Details

The RU functions by stepping through the destination pixel array and, with each step, incrementing a pair of high-precision registers that contain the coordinates of the destination pixel transformed back to the coordinate system of the source image. Each time the destination pixel location is incremented by one pixel, the location of the new source pixel is calculated by adding the contents of two registers to the contents of the X dimension source register and the contents of two other registers to the contents of the Y dimension source register. The values of the four summing registers represent the unit vector in the destination image transformed back to the source image space. The X and Y dimension source registers are initialized with the coordinates of the upper left corner of the destination image (1,1) transformed back into the source image coordinate system. Thirty two bits each of integer and fraction (64 bits in total) represent the X and Y dimension source image coordinate registers and each of the four registers that are used to increment them to assure sufficient range and precision. The initial contents of Xsrc and Ysrc and the values of the four summing registers can be found by inverting the system of equation 1.

Once the required source pixel coordinates are found, they are translated into the linear address space of the input pixel array so that the pixel values can be fetched from source memory and loaded into the next stage of the pipeline, the Color Conversion Unit (CCU). The PF300 asserts itself as memory bus master in order to read the required data from SDRAM. Any resulting source image coordinate that falls outside of the source image is marked with a 0 in the “shadow” image mask indicating an invalid pixel value in the output buffer.

A data flow chart of the RU is shown in Appendix A, “Resampling Unit Flow Chart,” and a C language source code listing of the RU implementation in the MPC8220i is supplied in Appendix B, “Resampling Unit C Code.”

2 Color Conversion Unit (CCU)

2.1 Algorithm Description

The Color Conversion Unit (CCU) performs a lookup and piecewise interpolation in a sparse three-dimensional lookup table. A sparse lookup table is one in which only a subset of the possible inputs are represented by data. Outputs for intermediate values are calculated by interpolation between the points that are supplied. The CCU computes the interpolation in dedicated hardware.

Even though the table can be addressed with as many as 16,777,215 (2^{24}) different color combinations, each individual dimension only has 256 possible addresses. The CCU takes advantage of this fact by precomputing many intermediate results and pointers that can then be accessed for use in real time by simple lookups into several much smaller (256 element) tables. The use of this block allows the three-dimensional lookup with a table that is much smaller than the data size used to address it. Up to four tables can be loaded and interpolated simultaneously, each addressed by the same data triplet. This allows the interpolation of all components for four colorant printing systems with a single pass of the color data through the pipeline. Examples of color space conversions that can be performed using this method include the following:

- CIE Lab –> CMYK
- CIE Lab –> RGB
- RGB –> CMYK
- RGB –> CIE Lab
- RGB –> K
- CMYK –> K

Both tri-linear and tri-nonlinear interpolations can be implemented. The algorithm allows non-uniform grid point spacing (that is, the table indices must be located on a three-dimensional grid, but the grid can be either irregular or regular, and the table need not be of equal size in each of the three address dimensions). This allows selection of grid points to coincide precisely with certain important colors including the 16 “Neugebauer” primaries: white paper, C, M, Y, K, CM, CY, CK, MY, MK, YK, CMY, CMK, CYK, MYK, and CMYK. It is important to locate at least some of these colors on table grid points so that (a) they are interpolated with zero interpolation error, and (b) the color gamut of the printer is fully utilized by assuring that the “tent poles” of the interpolating table are located at the extremes of the interpolation space. The use of irregular grids also allows for finer table granularity in regions of color space where the color conversion function has the most curvature. This reduces possible interpolation error while maintaining a relatively small table size.

Following the three-dimensional lookup and interpolation each output channel is further processed by its unique one-dimensional look up table to affect tonal corrections prior to piping the converted data to the next processing block, the Screening Unit (SU).

2.2 Computational Details

Each of the output colors is represented by a sparse three-dimensional (meaning that they are addressed in three-dimensions) color lookup table that can be addressed by 8-bit values in each of the three independent variables that constitute its “input” space. The color lookup tables are stored in system RAM. Not all of the 256^3 possible inputs to the table are represented in the table, however, which is why it is referred to as a “sparse” table. Instead, each of the color tables consists of 8-bit colorant output values for each of M table entries. The table entries are indexed in each of the three input dimensions by indices (I_1, I_2, and I_3), which have values (1, 2, … N_i). The total number of entries in the table is M ($= N_1 \times N_2 \times N_3$). The function of the CCU is to receive a triplet of 8-bit RGB values, to find the eight table entries that surround it, to get the 8-bit color values of each of the surrounding table entries, and to compute the interpolated result for the RGB triplet.

Dedicated lookup tables are supplied to contain the indices and the remainders for each possible input data value in each of the three input dimensions. To effect linear interpolations the tables of indices are programmed with the index number corresponding to the next lower input (address) value that has a table entry, and the tables of remainders are programmed with the distances between the input (address) value and the next lower input (address) value that has a table entry. The remainders are expressed as the fractional distances between the input (address) value of the next lower address with a table entry and the one next higher with a table entry encoded as 8-bit values. To facilitate the calculations, the remainder table is programmed with the bit-wise complement of the computed remainder (contents = 255 – remainder). An example of typical remainder and index tables are shown in Table 1.

Table 1. Example of CCU Index and Remainder Tables for Linear Interpolation

Address	Index Table Contents	Remainder	Remainder Table Contents	Comments
0	1	0	255	Start of range of addresses associated with first index
1	1	85	170	One-third distance through span
2	1	170	85	Two-thirds of distance through span
3	2	0	255	Start of range of addresses associated with second table entry
4	2	28	227	One-ninth distance through span

Table 1. Example of CCU Index and Remainder Tables for Linear Interpolation (continued)

Address	Index Table Contents	Remainder	Remainder Table Contents	Comments
5	2	57	198	Two-ninths distance through span
6	2	85	170	Three-ninths distance through span
7	2	113	142	Four-ninths distance through span
8	2	142	113	Five-ninths distance through span
9	2	170	85	Six-ninths distance through span
10	2	198	57	Seven-ninths distance through span
11	2	227	28	End of range of addresses associated with second table entry
12	3	0	255	Start of range of addresses associated with third table entry
13	3	16	239	One-sixteenth distance through span
14	3	32	223	Two-sixteenths distance through span
15	3	48	207	Three-sixteenths distance through span
16	3	64	191	Four-sixteenths distance through span
17	3	80	175	Five-sixteenths distance through span
18	3	96	159	Six-sixteenths distance through span
19	3	112	143	Seven-sixteenths distance through span
20	3	128	128	Eight-sixteenths distance through span
21	3	143	112	Nine-sixteenths distance through span
22	3	159	96	Ten-sixteenths distance through span
23	3	175	80	Eleven-sixteenths distance through span
24	3	191	64	Twelve-sixteenths distance through span
25	3	207	48	Thirteen-sixteenths distance through span
26	3	223	32	Fourteen-sixteenths distance through span
27	3	239	16	Fifteen-sixteenths distance through span
28	4	0	255	Start of range of addresses associated with fourth table entry
.	.	.	.	
.	.	.	.	
.	.	.	.	
240	10	0	255	Start of range of addresses of tenth table entry
241	10	23	232	
242	10	46	209	
243	10	70	185	
244	10	93	162	

Table 1. Example of CCU Index and Remainder Tables for Linear Interpolation (continued)

Address	Index Table Contents	Remainder	Remainder Table Contents	Comments
245	10	116	139	
246	10	139	116	
247	10	162	93	
248	10	185	70	
249	10	209	46	
250	10	232	23	End of range of addresses of tenth table entry
251	11	0	255	Start of range of addresses of eleventh table entry
252	11	64	191	
253	11	128	128	
254	11	191	64	End of range of addresses of eleventh table entry
255	12	0	255	Last table entry

Table 1 shows the contents of typical index and remainder tables for one of the three input dimensions. In this example, the table has 12 indices, or table entries ($N_i = 12$, for this dimension). The number of indices is programmable within the range of 2 to 256. The first table entry spans three input addresses, so the remainder table for addresses in this span are programmed with 255–0×(255/3), 255–1×(255/3), and 255–2×(255/3). The second table entry spans nine input addresses, so the remainders table is programmed with 255–0×(255/9), 255–1×(255/9), 255–2×(255/9), 255–3×(255/9), … 255–7×(255/9), and 255–8×(255/9). The index and remainder table entries for the rest of the addresses are programmed in a similar manner. Each of the three input channels has its own set of index and remainder tables, so there is no requirement for the color table to be indexed the same way in each dimension.

The three-dimensional color lookup table itself is programmed with the desired output values for the input addresses that constitute the first of each span (that is, those with zero remainders, 255 as programmed values). They are aligned with the indices.

In use, a triplet of 8-bit color values for each pixel is input to the block. Each individual 8-bit color value is used to address its separate index and remainder tables. This results in the coordinates of a three-dimensional color lookup table grid point (the three indices) and a three-dimension vector representing the fractional distance to the next table grid point in each dimension (the reminders). Since the three indices cannot be interpreted directly as an address to the color lookup table memory, they must first be combined to form the addresses of the eight color lookup table grids that surround the point being interpolated. This is accomplished by using the indices to address the CCU offset lookup table memory and summing the result to form a physical address of the location in system memory containing the output color data for the required grid point. The CCU offset lookup table must be programmed with the proper decoding algorithm according to the following equations:

Input color channel 1: Offset Value = BaseAdr + $(I–1) \times 4$

Input color channel 2: Offset Value = $N_1 \times (I–1) \times 4$

Input color channel 3: Offset Value = $N_1 \times N_2 * (I–1) \times 4$

In each of these equations, the BaseAdr is the starting address of the sparse color lookup table in system memory. I is the index number for the table. N_1 and N_2 are the number of table indices in each of the first two index dimensions, respectively. The additional factor of 4 is required if the normal three-color input (RGB) to four-color output (CMYK) conversion is being implemented.

The sparse output color lookup table data is stored sequentially in system memory starting at the BaseAdr. The first 32 bits of the color lookup table contains 1 byte for each of the output colorant values (in this case C, M, Y, and K) for the first table grid point. The second 32 bits of the colorant lookup table contains the output colorant values for the second grid point (indexed in the first input dimension) and so forth.

Once the indices are converted to colorant lookup table addresses, the output color data are fetched from the color lookup table. The vector of remainders is then used to perform the interpolation within the eight surrounding colorant values to produce an 8-bit colorant value for each output channel for the current pixel. The interpolation is computed in parallel for each of the output channels.

The last stage of the CCU consists of individual one-dimensional lookup tables for each of the interpolated output channels. The one-dimensional lookup tables are loaded into a dedicated PF300 memory block referred to as the CCU output lookup table, which must not be confused with the sparse three-dimensional colorant lookup table located in system memory and discussed in the preceding paragraph.

3 The Screening Unit

3.1 Algorithm Description

The Screening Unit (SU) of the PF300 implements a standard threshold array algorithm (See Adobe Systems Incorporated, *PostScript Language Reference Manual*, *Second Edition*, Section 6.4.5 pages 316-317) for halftone screening. A two-dimensional threshold array, also referred to as a tile, memory is loaded with 8-bit threshold values. Image pixel data is read into the Screening Unit in pixel-by-pixel, line-by-line (raster) order. With each input pixel the tile memory is addressed modulo (the tile X dimension size), the threshold value at that address is accessed, and a comparison is made with the current input pixel. If the input pixel value is less than the corresponding threshold array value then the binary output for that pixel is set to 1, otherwise it is set to zero. The binary output pixel is then stored in the output buffer, the next input pixel is input to the block, and the process repeats. At the end of each raster line the Y threshold array address is incremented and evaluated modulo (the tile Y dimension size) and the entire process repeats.

The result is a binary image that represents the input gray scale image through area, instead of intensity, modulation. The threshold array is applied as if it were replicated in a step and repeat fashion across the entire image.

3.2 Computational Details

The SU in the PF300 Image Coprocessor contains programmable registers for each of the (up to) four screen tiles to control the following:

- X dimension tile size
- Y dimension tile size
- Start address of the tile in threshold array memory
- X dimension starting pixel phase
- Y dimension starting pixel phase

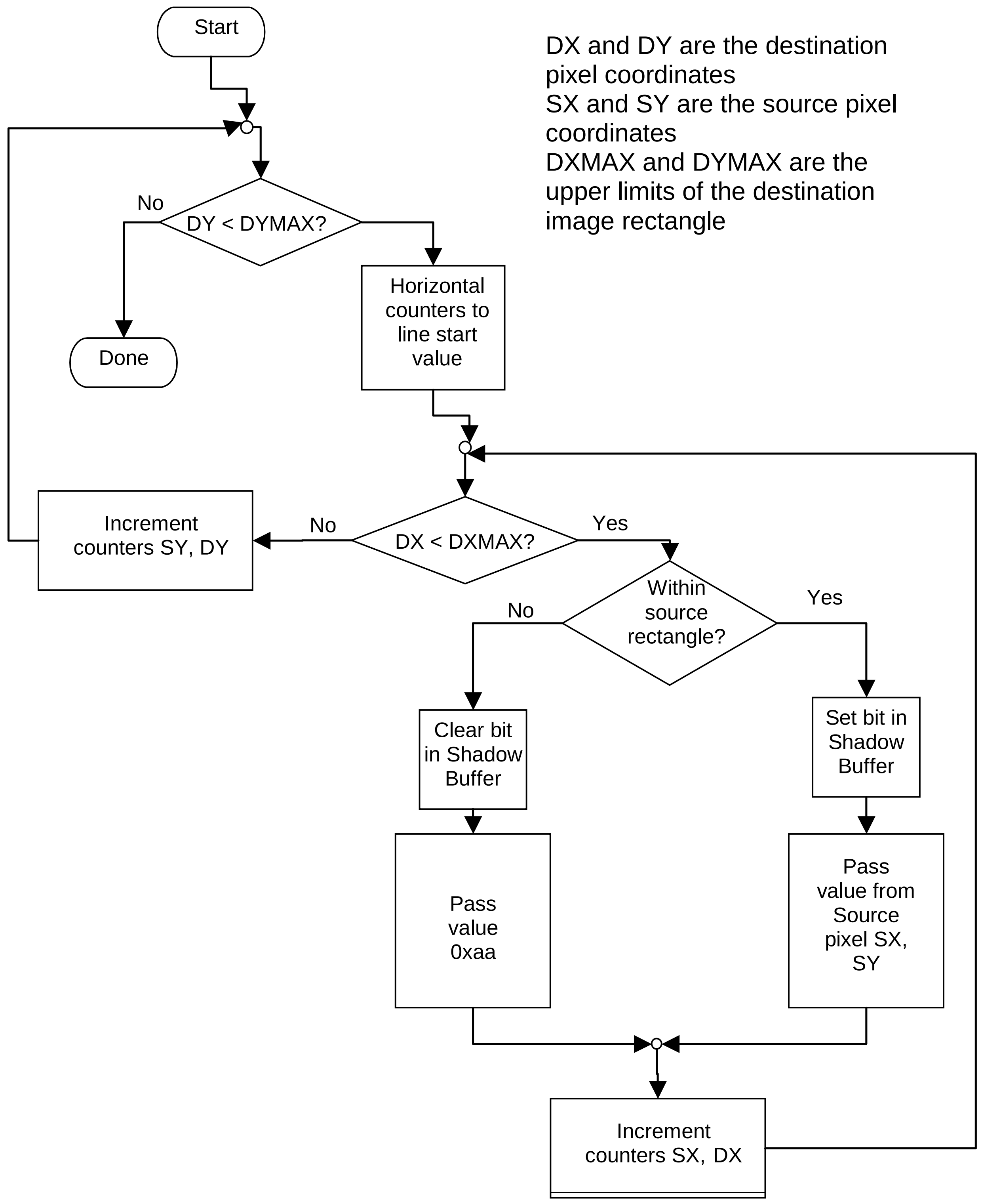
The first three registers allow the application that initializes the screening function to define a two-dimensional pixel array user space in which to work. The data held in these registers allow the Screening Unit to map the user space to the processor's linear address space. The last two registers allow the starting phase of each threshold array tile to be aligned to other threshold array tiles or to the beginning of the image. More detail about the location of the screener block control registers and the threshold array value memory can be found in Chapter 27 of the *MPC8220i Reference Manual*. A flow chart of the algorithm is presented in Appendix C, "Screening Unit Flow Chart," and the C code representation of the halftone screening algorithm as implemented in the PF300 is shown in Appendix D, "Screening Unit C Code."

The maximum tile size is limited by the amount of memory dedicated to this function. The implementation in the MPC8220i is limited to 4 Kbytes. Applications developers have significant flexibility to configure tiles of arbitrary dimensions that fit within this memory.

Each of up to four tiles can be configured with arbitrary X and Y dimensions with only two restrictions: each tile must be at least two scan lines high (Y dimension), and all of the tiles together must not exceed the 4 Kbytes of available memory. There are no further restrictions. A separate, dedicated, memory is provided for the tiles, which is mapped into the general purpose SDRAM address space so that the core processor can read and write tile data.

If the target engine prints only a single color plane at a time (including monochrome printers) and it can afford the overhead of reloading the screen tiles between processing of each color plane, then all 4 Kbytes can be used for the single tile. In some applications, particularly when "FM" screens are employed, the same tile can be used for each of the four-color planes so long as they are addressed out of phase from one another. In this case the entire 4 Kbytes of memory can be dedicated to the single threshold array tile, and each color plane can address it simultaneously in proper phase through appropriate initialization of the X and Y dimension starting pixel phase registers.

Appendix A
Resampling Unit Flow Chart


Start
No
DY < DYMAX?
Done
Horizontal counters to line start value
Increment counters SY, DY
No
DX < DXMAX?
Yes
Within source rectangle?
No
Yes
Clear bit in Shadow Buffer
Set bit in Shadow Buffer
Pass value 0xaa
Pass value from Source pixel SX, SY
Increment counters SX, DX

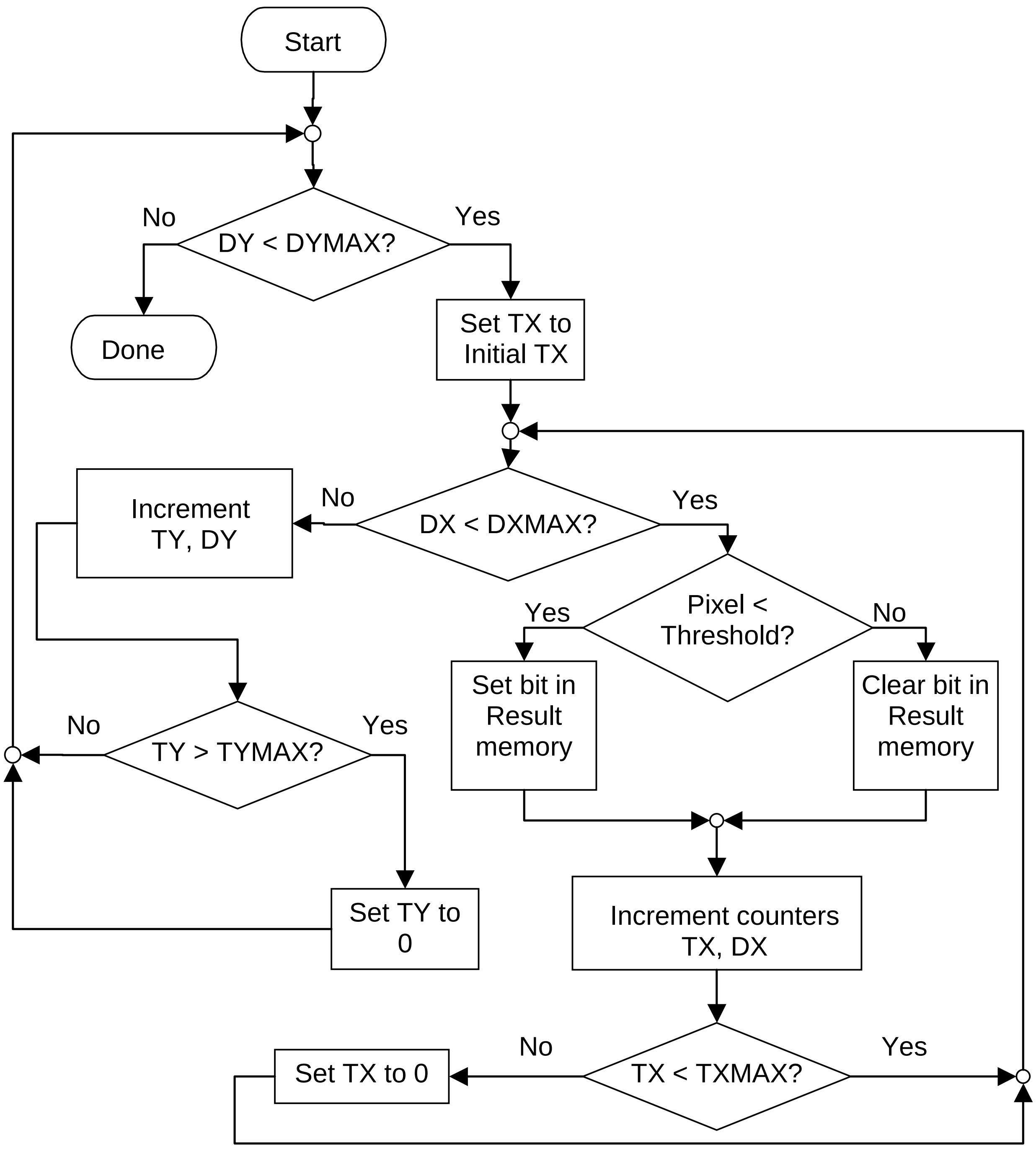
DX and DY are the destination pixel coordinates
SX and SY are the source pixel coordinates
DXMAX and DYMAX are the upper limits of the destination image rectangle

Appendix B
Resampling Unit C Code

```
Data types:
  Uint33  // unsigned 33-bit integer.
  Uint32  // unsigned 32-bit integer.
  Sint32  // signed 32-bit integer.
  Uint8   // unsigned 8-bit integer.
  Uint1   // Single bit value

  Adding a "*" to any of the data types indicates it is the address of
  a value of the specified type.

  Adding a "[]" to any of the data types indicates it is an array with
  allocated storage of the specified type.  The number between the
  "[]" indicates how many elements of storage of the specified type
  are allocated.

PDLA initial condition register definitions:

  Sint32 sxmini;     // Source x integer position at top left of dest
  Uint32 sxminf;     // Source x fraction position at top left of dest
  Sint32 symini;     // Source y integer position at top left of dest
  Uint32 syminf;     // Source y fraction position at top left of dest
  Sint32 dsxi;       // delta x integer in source per dest pixel delta x
  Uint32 dsxf;       // delta x fraction in source per dest pixel delta x
  Sint32 dsyi;       // delta y integer in source per dest pixel delta x
  Uint32 dsyf;       // delta y fraction in source per dest pixel delta x
  Sint32 dvsxi;      // delta x integer in source per dest pixel delta y
  Uint32 dvsxf;      // delta x fraction in source per dest pixel delta y
  Sint32 dvsyi;      // delta y integer in source per dest pixel delta y
  Uint32 dvsyf;      // delta y fraction in source per dest pixel delta y
  Uint32 nsrc;       // Highest source line index, Should be Uint8!
  Uint32 w;          // Source width (in pixels)
  Uint8 *src_p[256]; // Addresses of the left most pixel of source scan lines
  Uint32 dx;         // Number of pixels to step destination in x
  Uint32 dy;         // Number of pixels to step destination in y
  Uint8 *shadow;     // Address to store result for shadow buffer
  Uint8 *dest;       // Address to store result


Temporary registers:
  Uint32 x;      // Current horizontal position in destination
  Uint32 y;      // Current vertical position in destination
  Sint32 sxi;    // Current integer portion of position in source
  Uint33 sxf;    // Current fractional portion of position in source
  Sint32 syi;    // Current integer portion of position in source
  Uint33 syf;    // Current fractional portion of position in source
  Uint1 *shad_p; // Pointer to bit strings for shadow result

Resample loops:
```

```
// Loop over the number of lines in the destination
for (y = 0; y <= dy; y++)
  {
    // Reset the current position to beginning of next destination line
    sxi = sxmini;
    sxf = sxminf;
    syi = symini;
    syf = syminf;

   // Reset shad_p to current shadow position
   shad_p = (Uint1 *)shadow;

    // Loop over the width of the destination
    for (x = 0; x <= dx; x++)
      {
        // Check to see if location is within the source rectangle
        if ((sxi >= 0) && (sxi < w) &&
            (syi >= 0) && (syi < nsrc))
          {
             // Location is inside, set the correspoding bit in shadow
             shad_p[x] = 1;

             // Assign the value of the source pixel to the destination
             dest[x] = src_p[syi][sxi];
           }
        else
           {
             // Location is outside, clear corresponding bit in shadow
             shad_p[x] = 0;

             // Assign a constant value to the destination
             dest[x] = 170;
            }

        // Advance the source indices by the horizontal components
        sxi = sxi + dsxi;
        sxf = sxf + dsxf;
        syi = syi + dsyi;
        syf = syf + dsyf;

        // Add "carry" bit to integer locations if necessary
        if (sxf >= (1 << 32))
          {
             sxi = sxi + 1;
             sxf = sxf - (1 << 32);
           }
        if (syf >= (1 << 32))
           {
             syi = syi + 1;
             syf = syf - (1 << 32);
           }
      } // This completes the horizontal loop
    // Advance the source indices by the vertical components
    sxmini = sxmini + dvsxi;
    sxf = sxminf + dvsxf;
    symini = symini + dvsyi;
    syf = syminf + dvsyf;

    // Add the "carry bit to the integer locations if necessary
    if (sxf >= (1 << 32))
      {
         sxmini = sxmini + 1;
         sxf = sxf - (1 << 32);
       }
    if (syf >= (1 << 32))
       {
         symini = symini + 1;
         syf = syf - (1 << 32);
       }

    // Store the corrected for "carry" values
    sxminf = sxf;
    syminf = syf;

    // Advance the result addresses
    shadow = shadow + ((dx + 1) / 8);
    dest = dest + (dx + 1);
 } // This completes the vertical loop
```

Appendix C
Screening Unit Flow Chart



Appendix D
Screening Unit C Code

```
Data types:
  Uint32  // unsigned 32-bit integer.
  Sint32  // signed 32-bit integer.
  Uint8   // unsigned 8-bit integer.
  Uint1   // Single bit value

  Adding a "*" to any of the data types indicates it is the address of
  a value of the specified type.

  Adding a "*" to a reference to a data type means use the value
  pointed to by the variable for the operation.

PF300 initial condition register definitions:
  Uint8 *tilep;      // Pointer to top left pixel of tile
  Uint32 tileo;      // Offset in bytes to left most pixel on initial line
  Uint32 tx;         // Number of pixels in horizontal dimension of tile
  Uint32 ty;         // Number of pixels in vertical dimension of tile
  Uint32 txo;        // Initial horizontal position within tile (phase)
  Uint32 tyo;        // Initial horizontal position within tile (phase)
  Uint32 dx;         // Number of pixels to step horizontally in destination
  Uint32 dy;         // Number of pixels to step vertically in destination
  Uint1 *dest;       // Address to store result


Temporary registers:
  Uint32 txc;    // Current horizontal position in tile
  Uint32 tyc;    // Current vertical position in tile
  Uint8 *tp;     // Pointer to current value in tile
  Uint8 pixel;   // Value of the current input pixel
  Uint32 x;      // Current horizontal position in destination
  Uint32 y;      // Current vertical position in destination

Screening loops:

  // Initialize the current vertical location in the tile
  tyc = tyo;

  // Loop over the number of lines in the destination
  for (y = 0; y <= dy; y++)
    {
      // Reset the current horizontal location in the tile to inital pos.
      txc = txo;

      // Calculate the address of the initial position in tile.
      // tileo is incremented/wrapped at the bottom of the vertical loop.
      tp = tilep + tileo + txo;

      // Loop over the number of pixels in a line of the destination
      for (x = 0; x <= dx; x++)
```

```
         {
           // Get the next pixel from the pipeline.
           pixel = getNextPixel();

           // Compare the threshold to the pixel value
           if (pixel < *tp)
             *dest = 1;
           else
             *dest = 0;

           // Increment the destination
           dest = dest + 1;

           // Increment the horizontal position in tile
           txc = txc + 1;
           tp = tp + 1;

           // Check to see if tile needs to wrap horizontally
           if (txc >= tx)
              {
                // Reset current horizontal position in tile
                txc = 0;

                // Reset tile pointer to beginning of line.
                tp = tp - tx;
              }
           } // This closes horizontal loop

        // Increment the vertical location in the tile
        tyc = tyc + 1;
        tileo = tileo + tx;


// Check to see if tile needs to wrap vertically.
      if (tyc >= ty)
         {
           // Reset to top most row of tile
           tyc = 0;
           tileo = 0;
         }
     } // This closes vertical loop
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

AN2621/D, Rev. 0, 11/2003