Advanced Development

Classroom of Tomorrow
InterACTIVE Systems
ERDE
Systems Technology
Graphics & Sound
Object Oriented Systems

QuickScan Design Review

10 July 1986

Steve Perlman
ADG Computer Graphics

Agenda

- Project Goals

- System Overview
  - The Display Model
  - The System Architecture
  - The Line Buffer Architecture
  - The Drawing Primitives
  - Estimated Performance

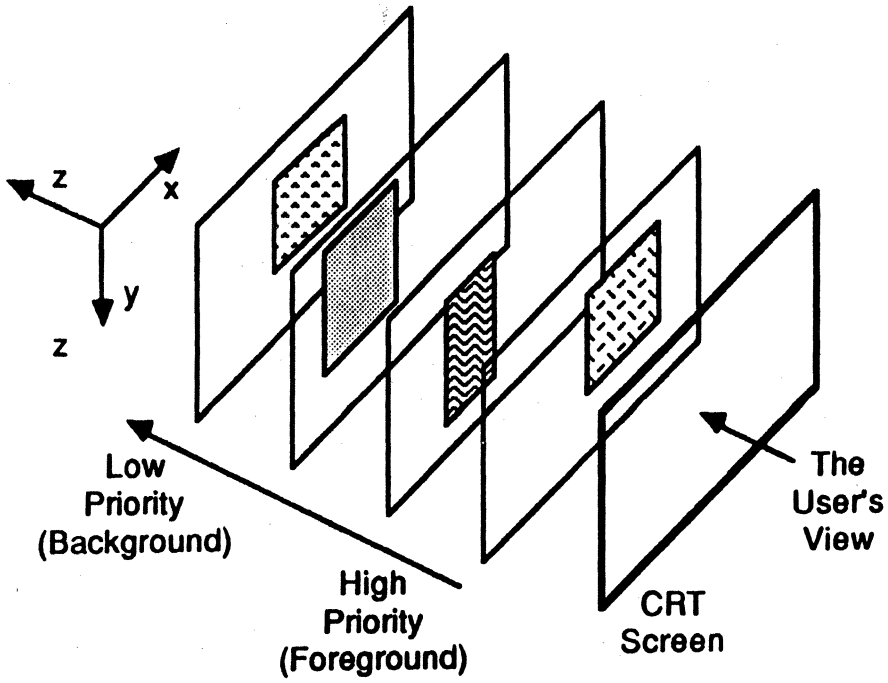
- Implementation Strategies
  - Outside Development
  - Internal Hardware Development
  - Internal Software Development
  - Productization

- Future Directions
  - Splinal Rasterization
  - Smooth Shading
  - Z-Buffering
  - Anti-aliasing

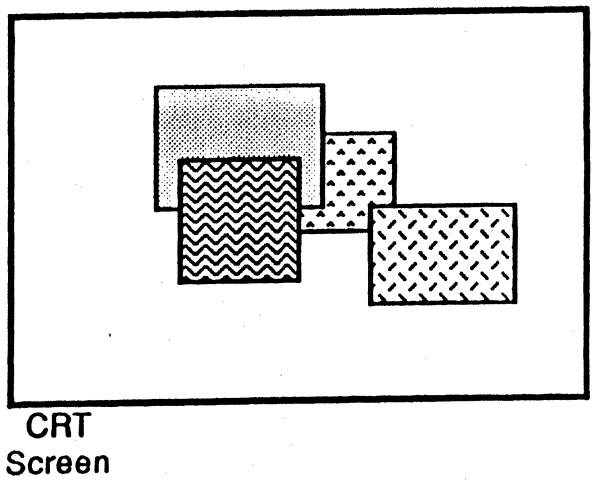
Project Goals

- Develop display subsystem which supports real-time animation of bit-map images, 3-D models, and cartoons.
- Support a 2-1/2D compositing model with as much generality as possible.
- Provide architecture which allows for individual displayed objects to be stored and handled independently.
- Incorporate within the compositing model mechanisms to reduce the spatial complexity of objects in both storage space and drawing speed.
- Keep the display model simple.
- Support color resolution up to 24 bits/pixel.
- Provide easy interface for special-purpose hardware to drive display subsystem.
- Have low-cost version.
- Maintain compatibility with existing and forthcoming Mac software.
- Support QuickDraw primitives wherever possible.
- Provide for future expandability.

The 2-1/2D Model



The Software Model

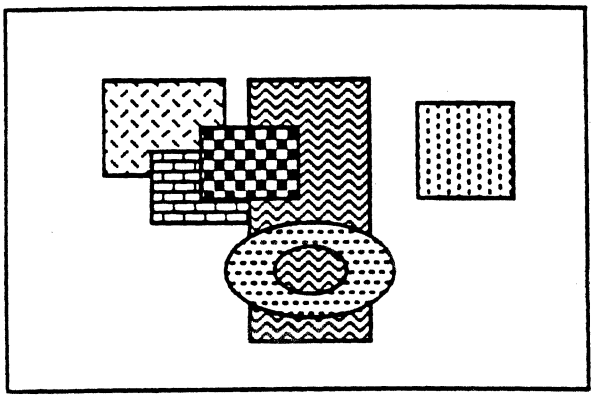


The User's View

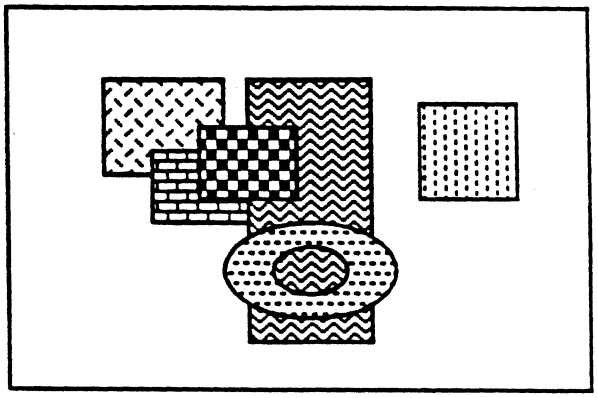
SGP 2/6/86

Frame Buffer Bit Map Organization

The RAM Layout
(bit maps are stored corresponding to their position on the screen)

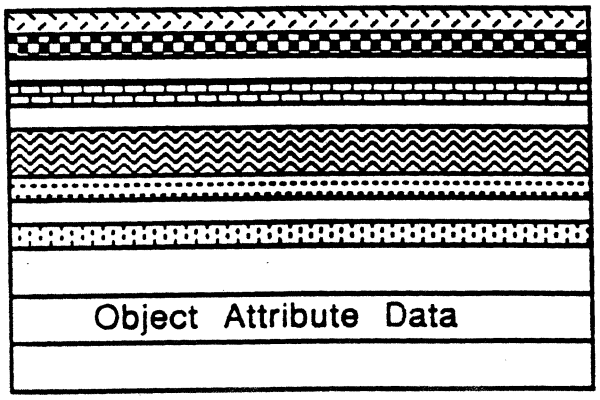


The Resulting Display
(the only possible display for that RAM organization)

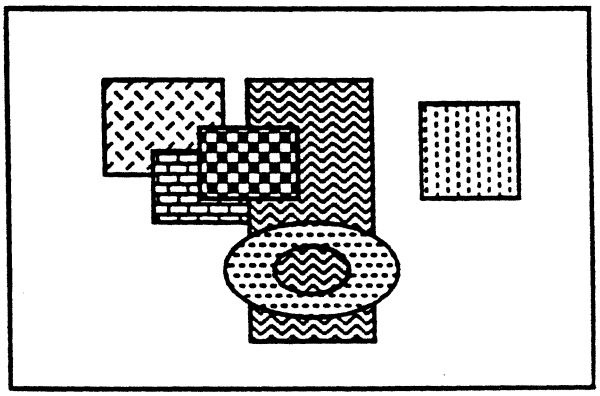


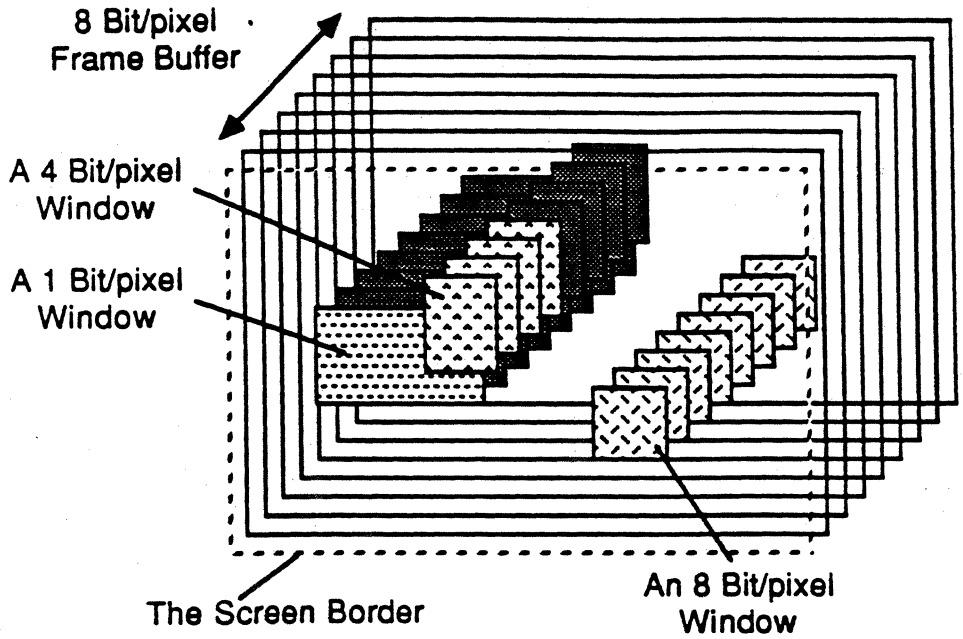
QuickScan Bit Map Organization

One Possible RAM Layout
(base addresses are arbitrary, but each bit map is stored contiguously)

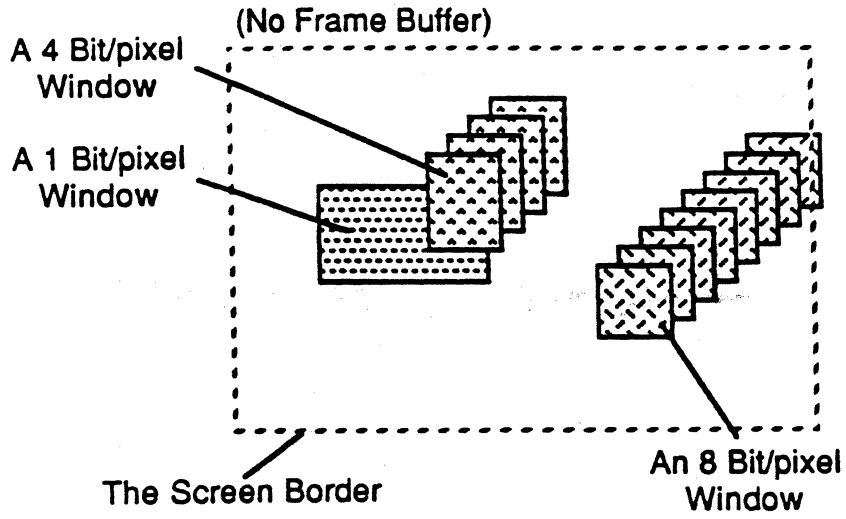


One Possible Resulting Display
(bit map positions are arbitrary)

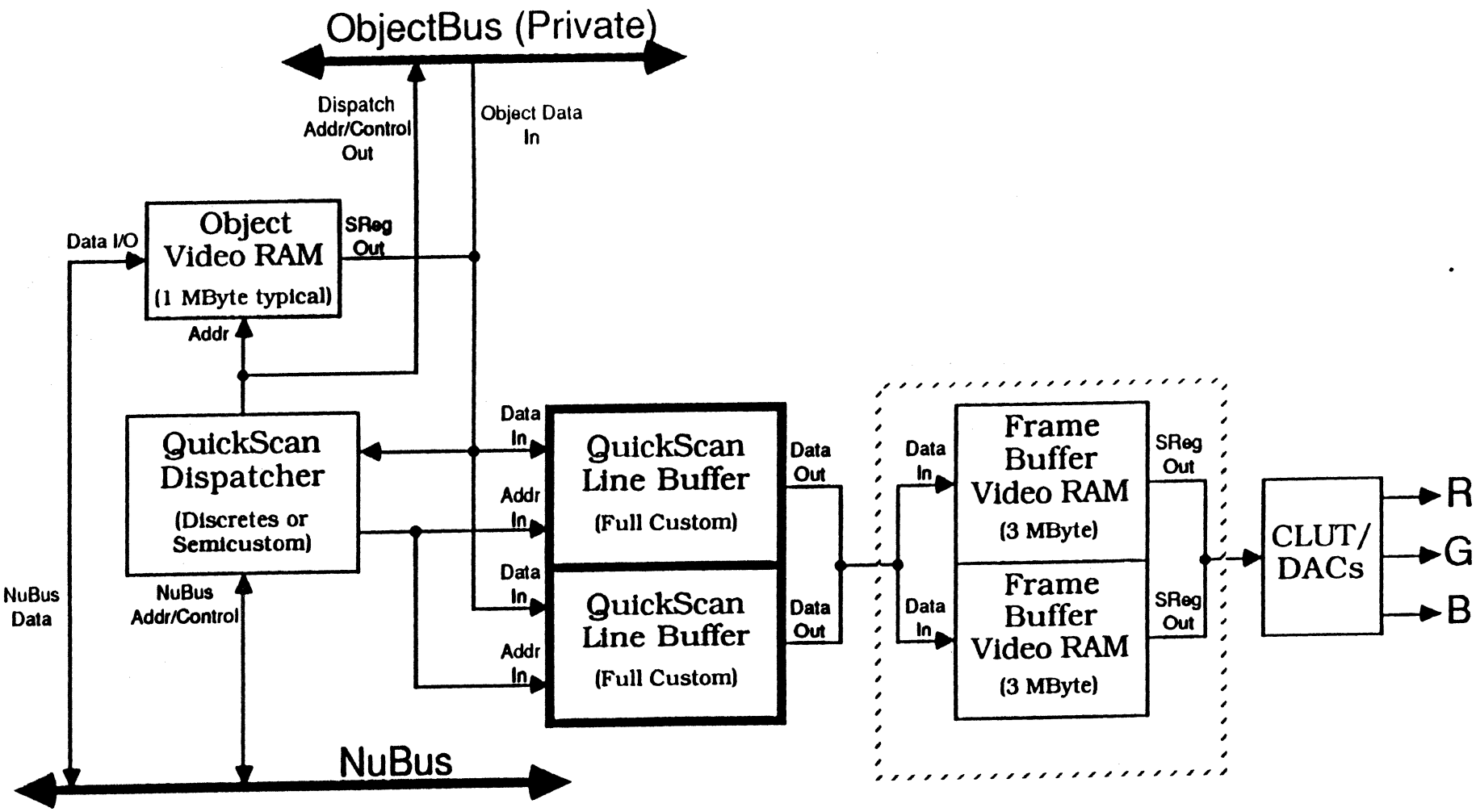




Software Model for Frame Buffer Windows



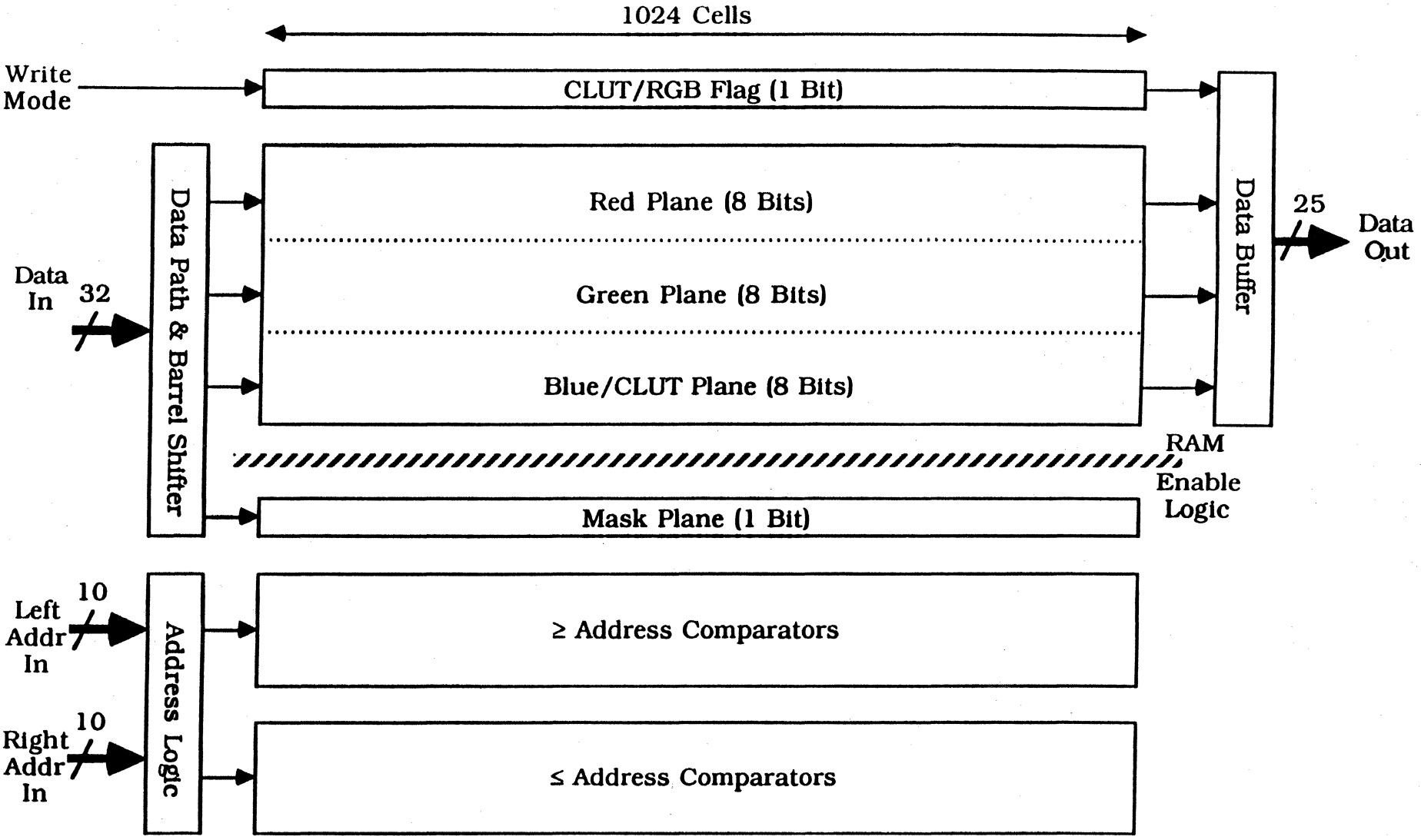
Software Model for QuickScan Windows



QuickScan: System Block Diagram

9 July 1986 *SGP*

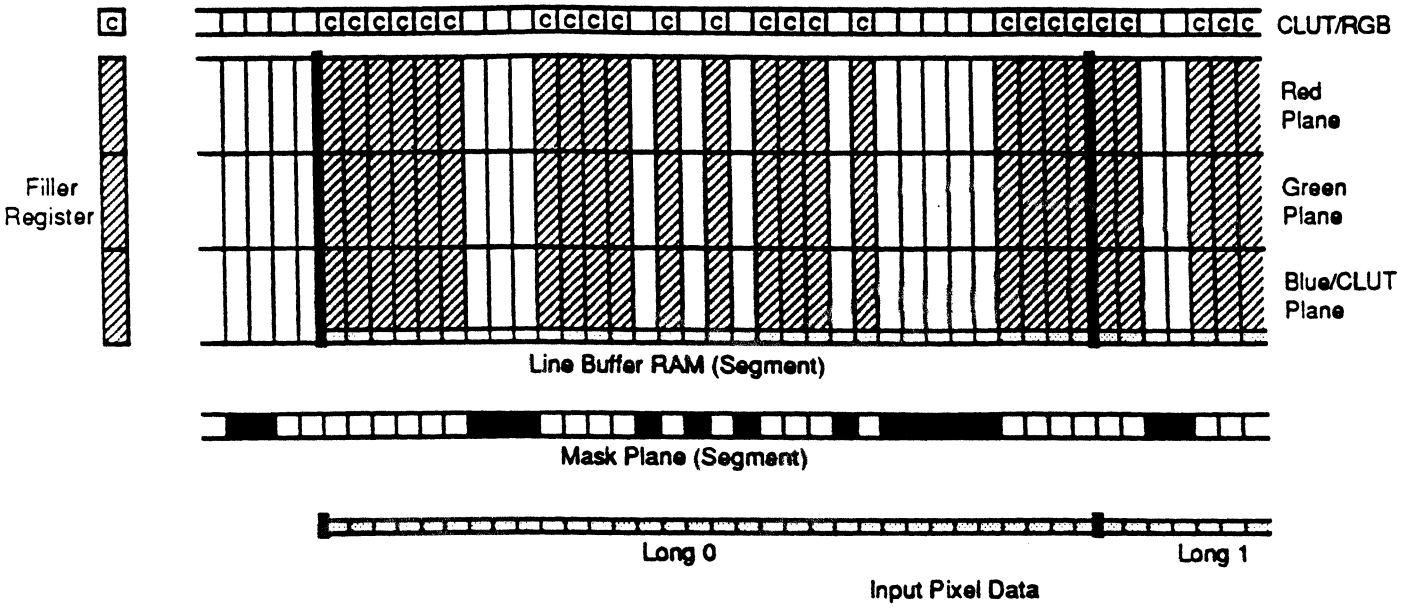
Apple Computer Confidential



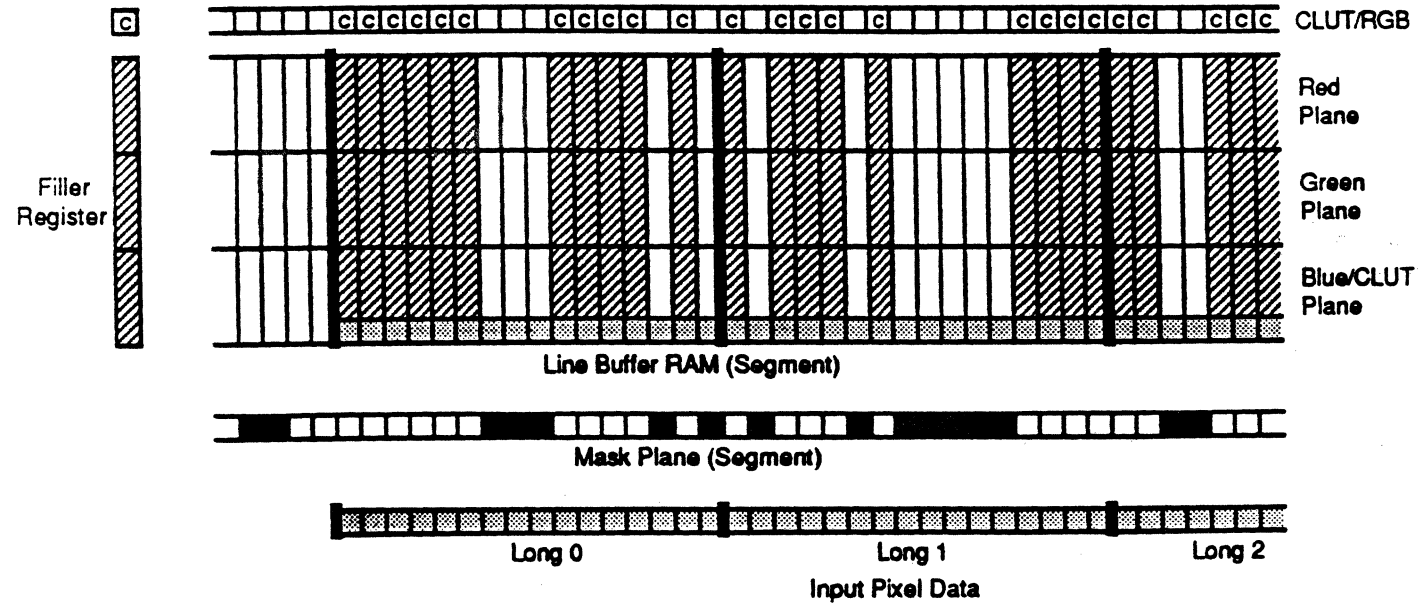
QuickScan: Conceptual Block Diagram

9 July 1986 *SGP*

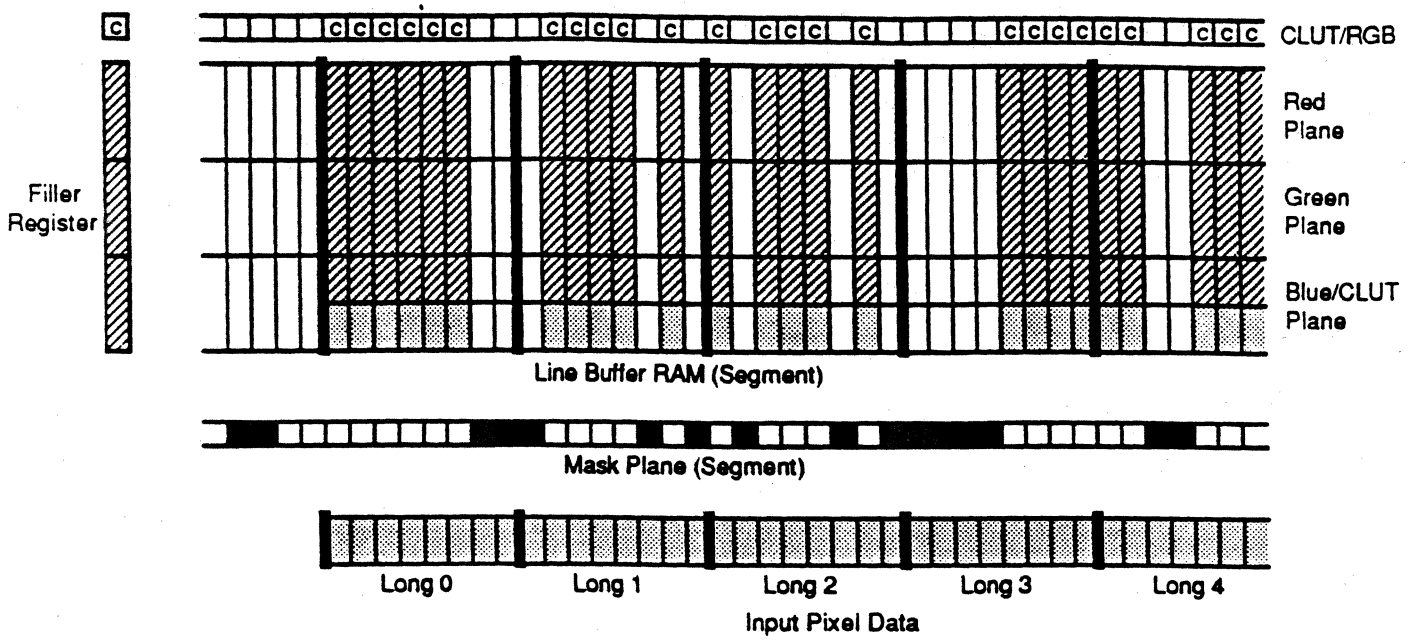
Apple Computer Confidential



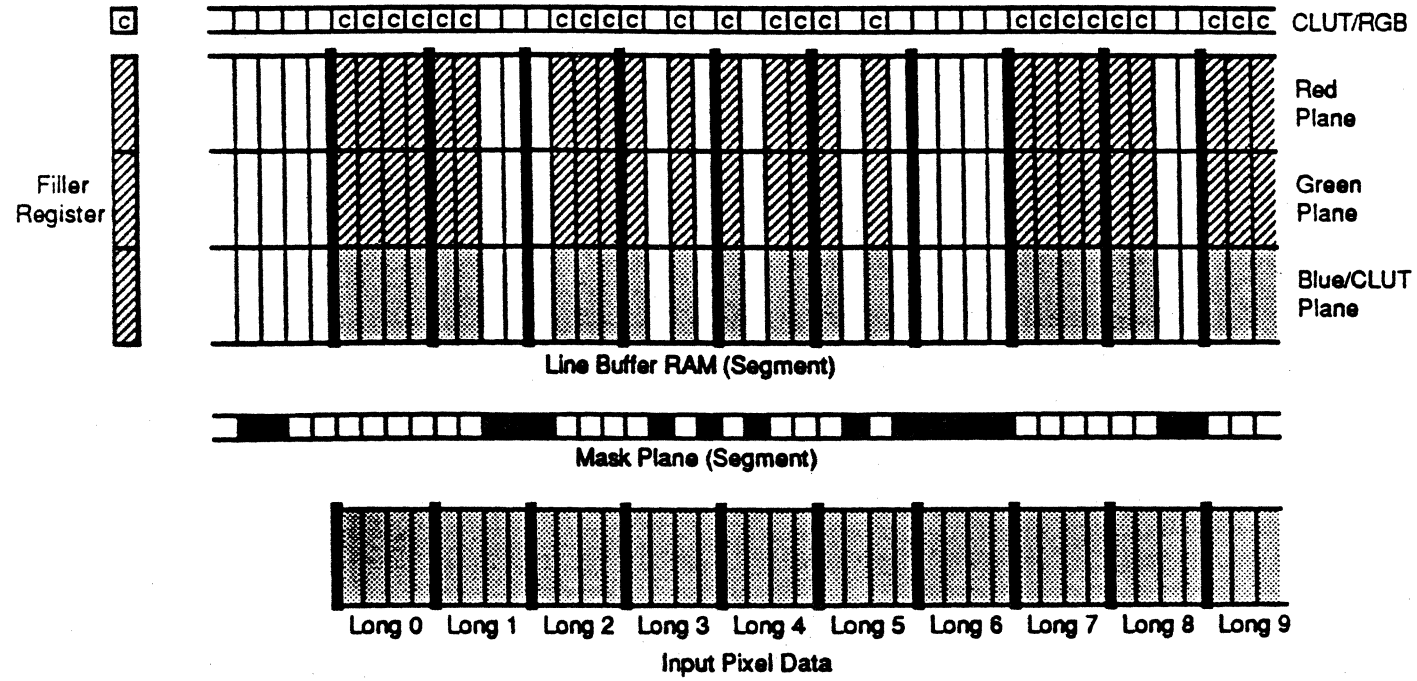
1 Bit/Pixel PixelMap Load



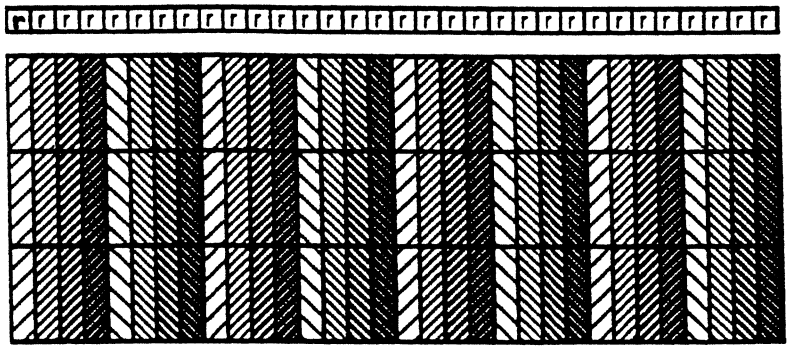
2 Bit/Pixel PixelMap Load



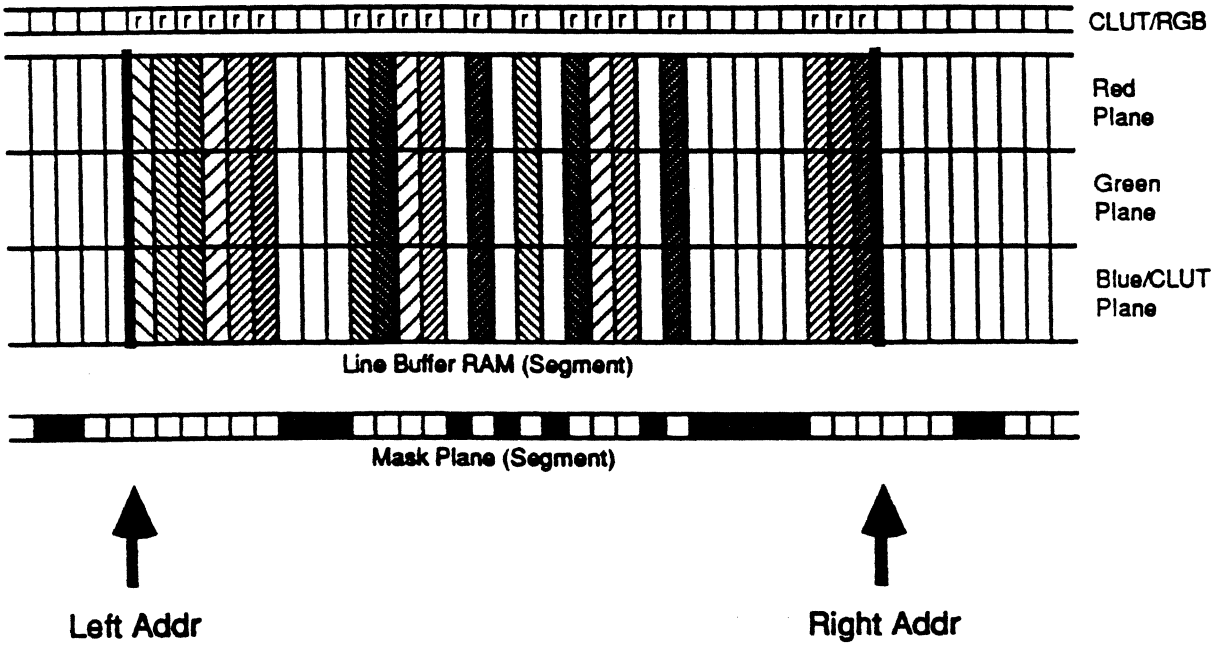
4 Bit/Pixel PixelMap Load



8 Bit/Pixel PixelMap Load



32 Filler Registers

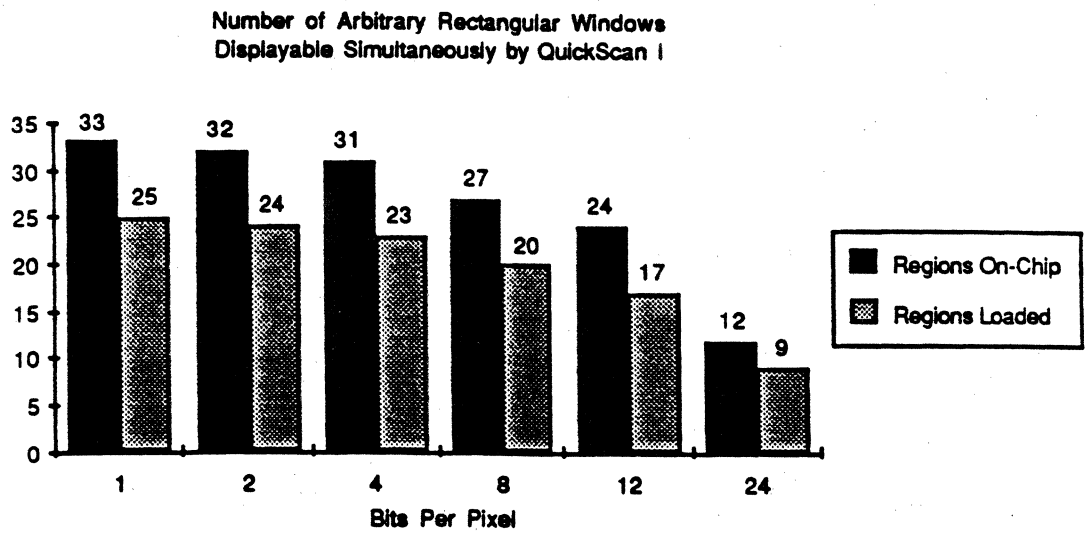


Modulo-8 Run Fill

QuickScan I Estimated Performance

(no frame buffer back-end):

Windows:



Polygons:

About 870 flat-shaded, convex polygons with 640x480 @ 67 Hz refresh.

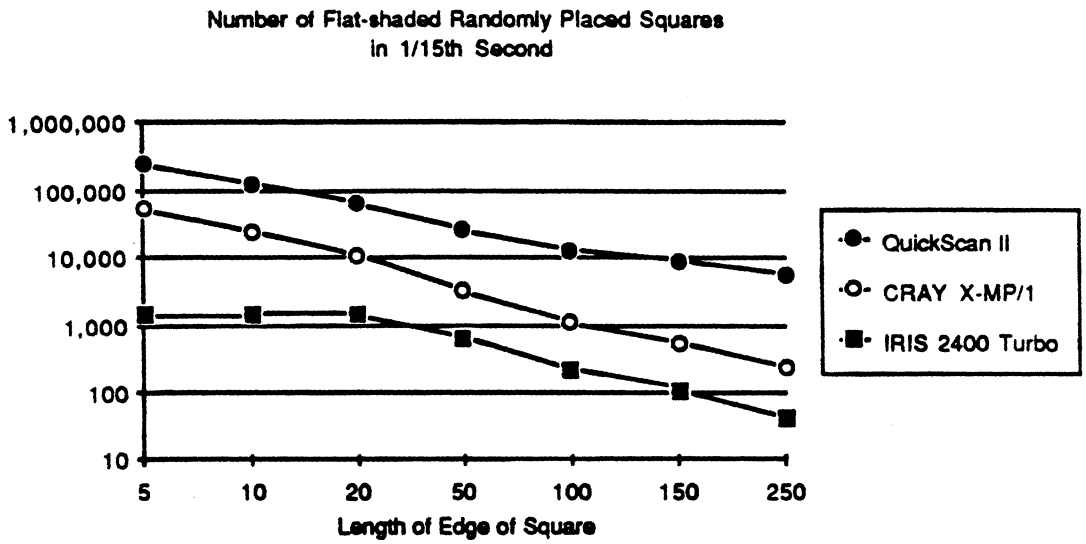
QuickScan II Estimated Performance

(double-buffered frame buffer back-end):

Windows:

Effectively unlimited number in real-time.

Polygons:

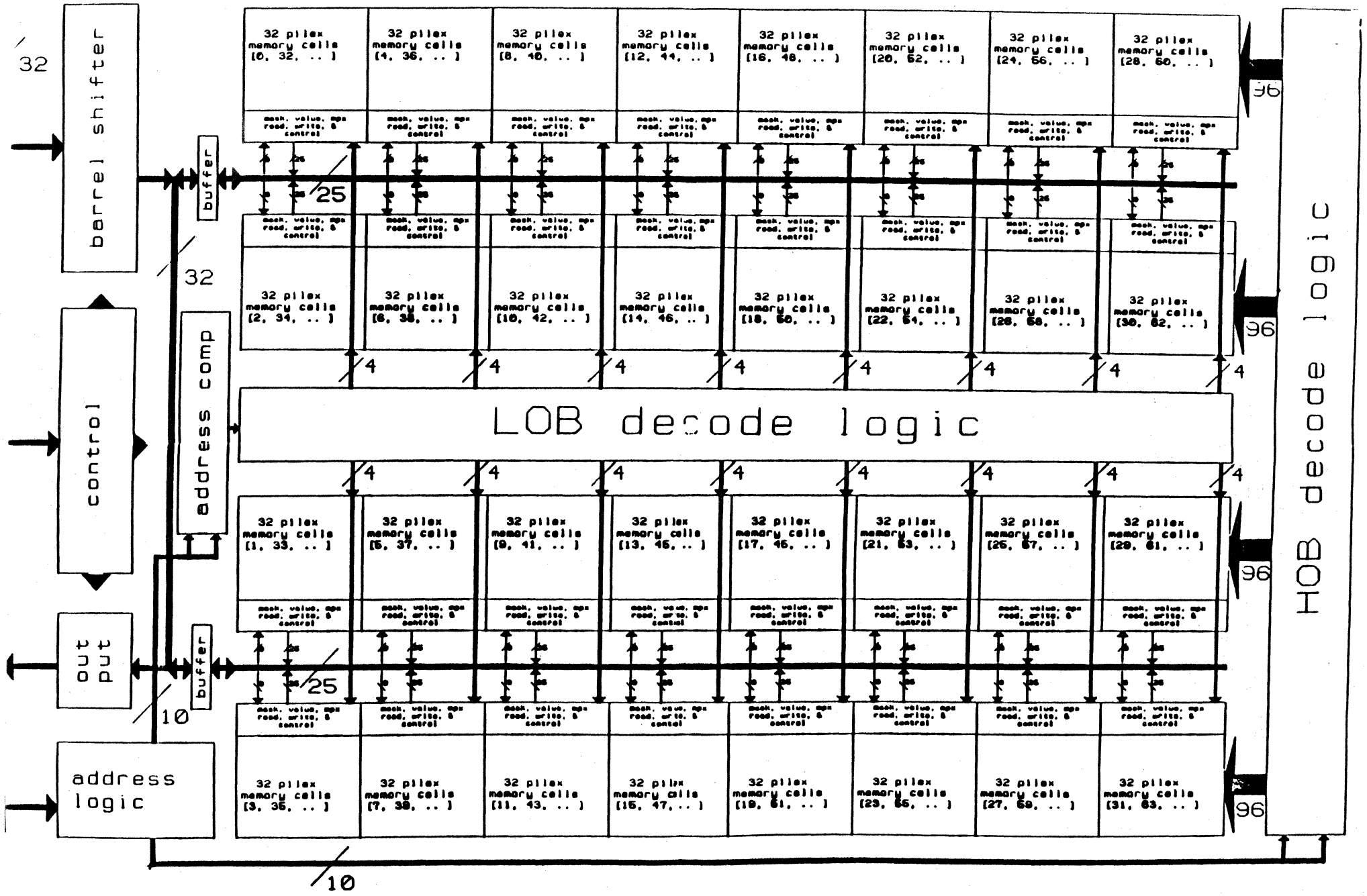


Outside Development

- Silicon Design Labs
- 1.5 micron double-metal CMOS process with dynamic cell characterization, probably Motorola
- Approx. 300 mil per side, <100 pins
- 14 months to packaged prototypes
- 2 chips per system

Internal Hardware Development

- The Dispatcher
- The QuickScan NuBus Card
- ObjectBus
- Polygonal Rasterization
- Ikki to Cray Interface


barrel shifter
32
buffer
32
address comp
control
out put
buffer
10
address logic
10
32 pilex memory cells [0, 32, ..]
32 pilex memory cells [4, 36, ..]
32 pilex memory cells [8, 40, ..]
32 pilex memory cells [12, 44, ..]
32 pilex memory cells [16, 48, ..]
32 pilex memory cells [20, 52, ..]
32 pilex memory cells [24, 56, ..]
32 pilex memory cells [28, 60, ..]
mask, value, ops read, write, & control
25
96
32 pilex memory cells [2, 34, ..]
32 pilex memory cells [6, 38, ..]
32 pilex memory cells [10, 42, ..]
32 pilex memory cells [14, 46, ..]
32 pilex memory cells [18, 50, ..]
32 pilex memory cells [22, 54, ..]
32 pilex memory cells [26, 58, ..]
32 pilex memory cells [30, 62, ..]
4
LOB decode logic
32 pilex memory cells [1, 33, ..]
32 pilex memory cells [5, 37, ..]
32 pilex memory cells [9, 41, ..]
32 pilex memory cells [13, 45, ..]
32 pilex memory cells [17, 49, ..]
32 pilex memory cells [21, 53, ..]
32 pilex memory cells [25, 57, ..]
32 pilex memory cells [29, 61, ..]
32 pilex memory cells [3, 35, ..]
32 pilex memory cells [7, 39, ..]
32 pilex memory cells [11, 43, ..]
32 pilex memory cells [15, 47, ..]
32 pilex memory cells [19, 51, ..]
32 pilex memory cells [23, 55, ..]
32 pilex memory cells [27, 59, ..]
32 pilex memory cells [31, 63, ..]
HOB decode logic

Internal Software Development

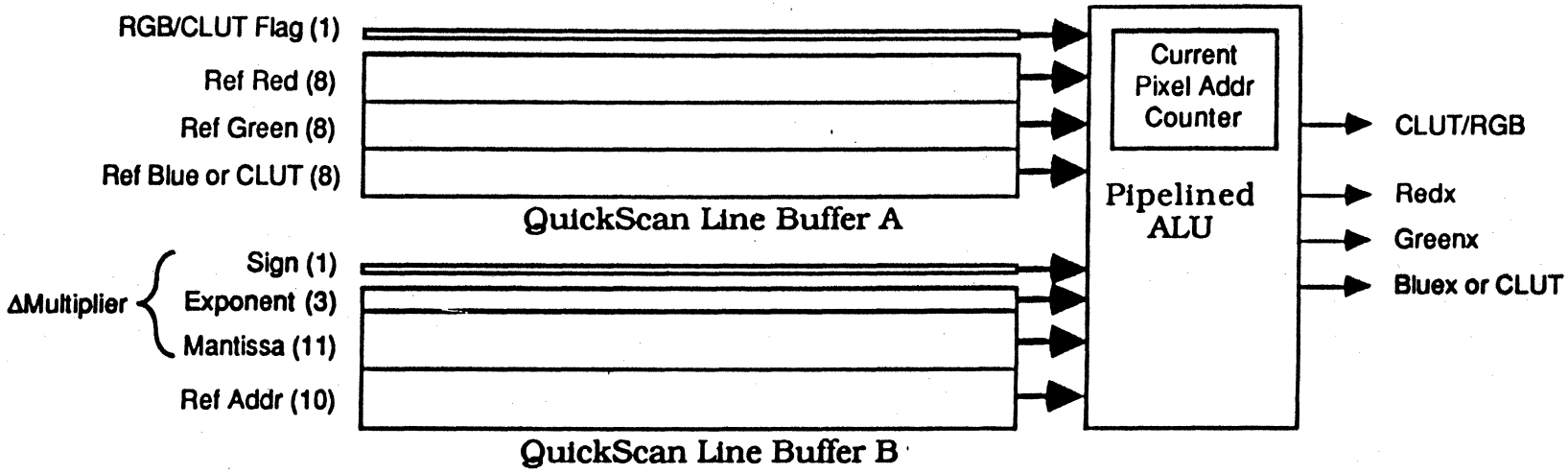
- The Display Model
- Window/Color Manager Extensions
- Object/Animation Manager
- Animation Applications
- QuickScan Simulation

Productization

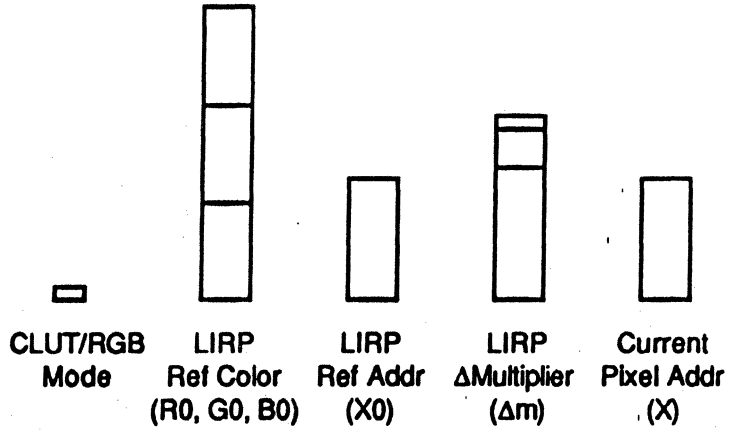
- Rev 0 QuickScan Product
- Rev 1 QuickScan Product
- External Graphics Cards

Future Directions

- Splinal Rasterization
- Smooth Shading
- Z-Buffering
- Anti-aliasing



Block Diagram



Data Available each Pixel time

$$\Delta m = (R1/R0 - 1)/(X1 - X0)$$

Formula Computed prior to Write (X1 is any X)

$$Rx = R0\ ((X - X0) * \Delta m + 1)$$

$$Gx = G0\ ((X - X0) * \Delta m + 1)$$

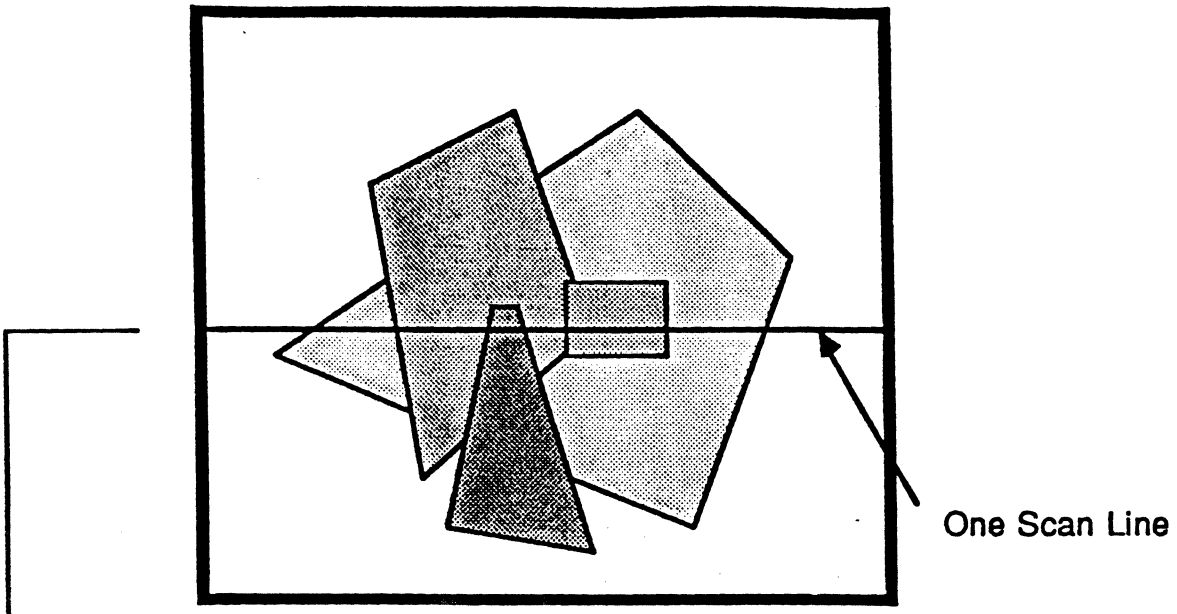
$$Bx = B0\ ((X - X0) * \Delta m + 1)$$

Formulae Computed in ALU

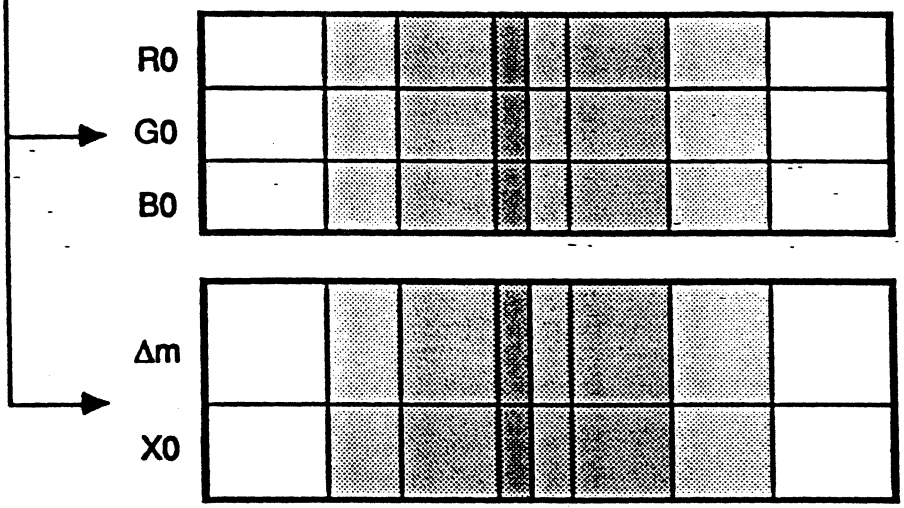
QuickScan LIRP System

Block Diagram

7/14/86 *SGP* Apple Computer Confidential



A Display with 4 Gouraud Shaded Polygons



Two QuickScan Line Buffers Filled with
Data from the Indicated Scan Line
(the result of 4 Write Operations)

QuickScan LIRP Line Buffer

Fill Example

Computed Formulae

$$Rx = R0 ((X - X0) * \Delta m + 1)$$
$$Gx = G0 ((X - X0) * \Delta m + 1)$$
$$Bx = B0 ((X - X0) * \Delta m + 1)$$

Operations	Clock Cycles
Δm(float) (15 bits) -> Δm(fixed) (19 bits)	3 (Shifter) 1 (Negate)
X - X0 -> Xnorm	1 (Add)
Xnorm * Δm -> m	10 (Mult)
m + 1 -> m	1 (Add)
R0 * m -> Rx G0 * m -> Gx B0 * m -> Bx	8 (Mult)
	24 (Total)

QuickScan LIRP ALU Pipeline

Computation Sequence

7/14/86 *SGP* Apple Computer Confidential

Advanc Steph - I dug this up. Thought you'd find it interesting Steve nent

Application Technologies
Classroom of Tomorrow
External R&D in Education
Graphics & Sound
Object-Oriented Systems
Systems Technology

To: Mike Potel
From: Steve Perlman x6248
Date: 27 August 1986
Subject: Parallel Gouraud Shading with QuickScan
cc: Graphics, North, Tesler, Marion, Kay

incorporate early Figs.

Abstract

First Paragraph.

Background

The QuickScan Line Buffer chips which we are currently implementing in VLSI incorporate a parallel write mechanism which allows us to fill with a single color or a repeating pattern any contiguous range of a single line in a single write cycle. By using this parallel write capability repeatedly we can fill large, overlapping areas very rapidly, provided that each area filled has long horizontal stretches of a single color or a repeating pattern (i.e. they are *spatially coherent*). If, however, an area to be filled has differing color values across horizontal stretches, then it is best filled using the sequential write mechanism with a bit map (or at best with a sequencer if the data is not random) producing the color information to be written to the QuickScan Line Buffer.

Although sequential writes allow us full generality of coloring varying horizontal stretches, we achieve that generality at a cost of about 2 orders of magnitude in speed. If there is no pattern at all to the colors being written (e.g. text or a digitized image), then there is nothing much to be done. But, if there is some regular progression to the data, then it is possible to construct a parallel computation structure which determines a unique color value for each pixel in the line as a function of its position (the approach used by Henry Fuchs at UNC for "Pixel Planes").

While a parallel computation approach is quite possible, it unfortunately is very expensive because, not even considering the parallel computation mechanism, there ultimately must be a unique data path for each pixel cell in the line buffer. A common data path, shared by groups of pixel cells, requires far less silicon real

estate, but unfortunately implies that any parallel writes to the pixel cells of a given group must all be written with the same data. We opted for the common data path approach with QuickScan because we simply could not fit its 1024 x 25 bits of 25 MHz RAM onto a reasonable die without such optimization. It would seem that we are destined to support no more than parallel writes of single colors and repeating patterns (i.e. one color to each pixel cell group) with QuickScan's architecture.

Such a limitation is regretable because there is a class of horizontal color progressions which are *extremely* useful in computer graphics: color computations which are a function of horizontal position (i.e. of x), and in particular, first-order functions of x, called "Linear Interpolations" or LIRPs for short. LIRPs generate a linear progression, or "ramp" of color intensity interpolated from a start intensity to a end intensity, which models point light source illumination of a one line of a perfectly matte surface (i.e. there is no specular reflection). This is useful for a number of applications, most notably for applying Gouraud (or Smooth) Shading to 3-D polygons (LIRPs are done both horizontally and vertically for this model).

Although most commercial 3-D systems support Gouraud Shading, none of them that we know of support it in anywhere near real-time. Even a new system, *Renaissance,* from Hewlett-Packard which has special hardware for smooth shading and purports "real-time Gouraud shading", in practice can only handle small, simple objects in real-time. Only Henry Fuch's experimental system at UNC, a rack of boards for even a low-resolution display, can fly about 6000 smooth-shaded polygons in real-time.

Smooth shading adds a substantial degree of realism to polygon modeling, and it is essential that we eventually provide such a capability for Apple 3-D graphics products. It is unfortunate that we have to wait for the next generation of our graphics hardware development in order to provide LIRPs in real-time... or do we?

Ten Bits of Data We All Forgot About

As far as I can tell, it is indeed the case that the current generation QuickScan parallel write mechanism can only fill a horizontal stretch with a single RGB data code or with a repeating pattern of RGB data codes. And, since a particular data code stored in the line buffer will always generate a particular color (e.g. R=255, G=255, B=0 always generates bright yellow), a horizontal stretch of the same data codes results in the same color or pattern being generated across that stretch. This is because when the data is scanned out of the line buffer, pixel by pixel, to be displayed on the monitor or written into a frame buffer, there is no other data except for the RGB data code by which to determine a color to display. . . .but is that really correct? Is that RGB data code the only meaningful data available when the data is read out of the line buffer? Could it be that we forgot about some extra data that's been there all along?

When data is read out in serial order, as is the case when QuickScan's line buffer is scanned-out, it is trivial to have a counter keep track of the number of clocks, for QuickScan the number of pixels, which have passed since the data stream began, for QuickScan the beginning of the horizontal line. This count provides another piece of data, notably a piece of data which is unique for each element of serial data. In the case of QuickScan the counter output is a 10 bit number which identifies the x position of the pixel currently being scanned-out. Coincidentally, we are concerned with computing a function of x! Perhaps there is something here for us to work with.

When one has a function of only x, she has by definition a formula in which x is the single variable, and all of the other elements are constants. Since a horizontal LIRP is a function of only x, it can be computed from only constants and x. QuickScan's parallel write mechanism can fill a horizontal stretch with constant values. Our pixel counter provides us with x. So, in theory, we should be able to apply a LIRP function based on x to the RGB constants as they are scanned-out of QuickScan using the constant data that was written in parallel. This effectively would give us smooth-shaded fills at the same rate that we get single color or repeating pattern fills. But, can it be done in practice?

3. The Mathematics of Horizontal LIRPs in RGB Space[1]

3.1. The Arithmetic

Since we are interpolating linearly from some color C_0 to some color C_1, from some position x_0 to some position x_1, then there must be some expression of the form $C_{\Delta x} = C_0 + m\Delta x$, where C_0 is the color at the start of the LIRP run, m is the unit change in the intensity of the color, and Δx is the unit distance from x_0. m can be derived from any two locations of the LIRP run, x_0 and x_1, by computing the slope, $(C_1 - C_0) / (x_1 - x_0)$. Since $\Delta x = x - x_0$, where x is the current pixel position, clearly C_x can be computed with the constants C_0, x_0, and m, and the variable x.

So, if we want to use the QuickScan parallel write mechanism to write information which can generate a LIRP run, all we need do is write the three constants, C_0, x_0, and m, across the length of a LIRP run in the line buffer, and provide an ALU on the output of QuickScan which computes C_x from these constants and the x provided by a counter.[2] Since we have three colors, R, G, and B, we need to duplicate C_0 and m three times for each component, resulting in R_0,

[1] As they apply to Gouraud shading. This section may not apply LIRPs in RGB space generally (but then again, it might!).

[2] Although it may seem ambitious to do a subtraction, a multiplication, and an add at the video rates that QuickScan outputs data, we can quite practically build a pipelined ALU to accomplish this arithmetic with effectively no more than a single addition per pixel clock cycle (more on this later).

G_0, and B_0 and m_R, m_G, and m_B. One x_0 will suffice if R_0, G_0, and B_0 all come from the same pixel location.[3]

In practice, LIRPs are primarily used to ramp between two intensities of the same hue. If this is the case, then the slope for each of the components is usually different, but the ratios between starting and ending values of each of the components is the same. That is:

$$\frac{R_1}{R_0} = \frac{G_1}{G_0} = \frac{B_1}{B_0} \quad ,$$

even though $m_R \neq m_G \neq m_B$.[4] Maybe we can make use of the coherence between the R, G, and B LIRPs to reduce the amount of constant data that must be stored with each LIRP run.

To do this, we must first determine a way to derive the slope for each component from a common constant, β, since we must have the slope in order to compute a component's value at any given x position. But, the only information that we have at the time of the computation which is component-specific is C_0, the reference color. So, if it is possible to derive the slope of a component from the common constant, β, the formula must involve the component's reference color, C_0. And, since this same formula must result in three separate slopes for the three components, β must include C_0 and C_1 only in a form which is invariant for all three components, i.e., C_1/C_0. As this necessitates an extra division in the formulation of β, there must be an extra multiplication when β is expanded. Hence, it is a good guess that the following formula can be used to compute each component's slope from β and each component's reference color, C_0:

$$m = C_0\beta \, .$$

Now, let's solve for β. If we substitute for m, we get:

$$(C_1 - C_0) / (x_1 - x_0) = C_0\beta \, ,$$

and after a little algebra, this becomes:

$$\beta = \frac{(C_1/C_0) - 1}{(x_1 - x_0)} \quad .$$

---

[3]In fact, no x_0 is needed if we normalize R_0, G_0, and B_0 from a known pixel location (e.g. pixel 0), but if they do not come from a pixel location within the extent of the LIRP, then we cannot guarantee that they will have values representable in 8 bits. (Perhaps we *should* normalize them and then represent their values in a different way. Under study.)

[4]For example, if we double the intensity of the RGB triple (1,2,4) across a stretch of 16 pixels, we get at the end of the LIRP the triple (2,4,8), resulting in slopes of 1/16, 1/8, and 1/4, respectively. The slope of the LIRP for each component is different, but the ratio between the starting and ending values of each of the components, 2:1, 4:2, and 8:4, is the same.

As we had hoped, the formulation of β indeed includes C_0 and C_1 only in a form which is invariant for all three components, C_1/C_0, so when β is multiplied with each component's C_0, it should result in a unique slope for each component.

Now, if we put all of the pieces together, inserting the derived slope into our original LIRP function, we get:

$C_{\Delta x} = C_0 + C_0\beta\Delta x$, or more conveniently,

$C_{\Delta x} = C_0 (\beta(x - x_0) + 1)$.

And, here we have what we were looking for: a relative intensity function of x and the constants C_0, β, and x_0, applicable to all three components. Thus, in order to use QuickScan's parallel write mechanism for filling a LIRP run of the same hue, we need to store only 5 constants: R_0, G_0, B_0, β, and x_0.

3.2. The Representation

The only question remaining is what numeric representation is appropriate for each constant?

Since our color resolution is 8 bits, there is little advantage to storing R_0, G_0, and B_0 as integers of more than 8 bits, provided that x_0 is chosen to be at a location where at least one of the components has its maximum value in the LIRP (i.e. the last pixel at the bright end of the LIRP). The reason for this is that numbers represented in fixed-point (in contrast to those represented in floating-point) can be represented more accurately (i.e. will have more bits of significance) when they are large. If we always multiply the largest color by an (accurate) fraction, the resulting color will never be off by more than 1/2 of the least-significant bit, which is as good as we can hope to do.[5]

x_0 is easy: there are 1024 pixels in a line, so x_0 can be located at exactly one of 1024 locations. We need a 10 bit integer.

β, however, presents a fairly complex numerical analysis. Its range extends from about 1/256K to 1, so if we use fixed point numbers, we need at least 19 bits of significance just to reach the extremes of range. Since we would like the accuracy of the derived color at each pixel to be within 1/2 of the least significant bit, we may need yet another bit of significance because we are going from an 8 bit color representation to effectively a 9 bit color representation. This gives us 20 bits for β's fixed-point representation.

[5]To see an illustration of this, consider the following example: We have a LIRP extending from 5% to 50% maximum intensity, and its hue is 30% red, 55% green, and 15% blue. The color at the bright end of the LIRP is (115, 210, 57) and the color at the dark end of the LIRP is (12, 21, 6). If we derive the dark end color by multiplying the bright end color by 10, we get (rounded) (12, 21, 6), which is accurate. If, however, we derive the bright end color by multiplying the dark end color by 10 we get (120, 210, 60), which has 4% inaccuracy in red and 2% inaccuracy in blue.

Closer analysis of this fixed-point representation of β, however, suggests that it is a pretty sparsely utilized code space. Specifically, it would seem that the larger numbers need no more precision than the smaller numbers, i.e., the LSB's of the larger numbers could be zero without any loss in precision. So, there would appear to be a need for fewer bits of precision than for range. This points toward investigating a floating-point representation.

Clearly, the exponent of the floating point representation should be 5 bits since we need to represent numbers from 2^{-19} to 2^{0}. But how many bits of fraction do we need? I'm certain that there is some correct analytical method of determining this, but after consulting with several sources, I could not get a consensus. So, when in doubt, simulate the hell out of it: I wrote a program in C which exhaustively goes through every possible LIRP which can be generated in a 1024-pixel line with 8 bits each of R, G, and B. Unfortunately, this turned out be quite a long simulation with so many cycles of floating point arithmetic. On "Apple", a busy Vax 11/750 with network responsibilities, it would have taken 1/2 year to finish. On "BigMac", a relatively lightly loaded Vax 11/780, it would have taken about 11 days (but I had to give it lower process priority in consideration of the other users, bumping it up to about 1 or 2 months). But, on TMA1, a presently lightly utilized Cray X-MP/48, using all 4 processors it took only 2-1/2 hours (they tell me if the simulation had been written in a vectorizing Fortran it would have finished in 15 minutes!). The end result was that it needed 9 bits of fraction, but since in normalized floating point the most significant bit of the fraction is always 1, that bit can be considered 1 implicitly, resulting in only 8 bits of fraction stored. This combined with the 5 bits of exponent results in 13 bits total for β.[6]

Actually, we probably could eliminate one more bit of exponent by using non-normalized fractions with the smallest exponent code (0000). This technique would extend the 4 bit exponent range of 2^{-15} - 2^{0} down to the 2^{-19} we need at small extreme for β. Unfortunately, it would also substantially complicate the encoding and decoding of the floating point number, so it is unclear whether it is worth saving 1 bit in the representation. Preliminary simulation indicates that the loss of accuracy in the denormalized numbers would still produce results within 1/2 of the least-significant bit, but I will not go to the effort of exhaustive simulation unless we find that we really need to save that one bit of storage.

Note that with either representation, we shall need some special code to mean zero. One possible encoding which is not otherwised used in either representation is all zeros in the exponent and the mantissa.

So, in summary, to accurately represent any LIRPs of constant hue across 1024 pixels accurate 1/2 of the least-significant bit of each of the color components we need minimally:

[6] If x_0 is known to always be located at the bright end of the LIRP, we can guarantee that β is always positive, and hence does need a sign bit.

- R_0, G_0, and B_0 stored as integers of 8 bits, containing the color at the bright end of the LIRP.
- x_0 stored as an integer of 10 bits, indicating the horizontal position of R_0, G_0, and B_0.
- β stored as an unsigned floating point number of 5 bits of exponent and 8 bits of fraction with an implicit 1 in the MSB of the fraction. The possibility exists to complexly encode the floating point number with just 4 bits of exponent and 8 bits of fraction.

As a caveat, remember that we have handily dismissed LIRPs which do not have a constant hue. Although such LIRPs are not as common as the constant hue variety, there are applications where they create useful effects (e.g. modeling with multiple colored light sources). If we wanted to implement such LIRPs, we would need an independent slope for each, R, G, and B instead of a common β. Each of these slopes could be represented accurately by an 11 bit signed, fixed-point number. Note that for this LIRP model, there is no advantage to choosing the bright end of the LIRP for x_0 because C_0 is never scaled and no accuracy is lost. Furthermore, we cannot eliminate the sign bit in the slope representation by establishing a convention for the placement of x_0 because we cannot guarantee that the three R, G, B LIRPs will be either all increasing or all decreasing. So, any choice of position for x_0 will do equally well.

4. The Implementation

4.1. Hue-invariant LIRPs

The QuickScan Line Buffer Chip we are currently developing (see Figure 3) will store 25 bits for each of 1024 pixels, 8 bits each for red, green, and blue, and 1 bit to indicate if the information is RGB or an index (stored in the blue plane) for a color lookup table (CLUT). Additionally, there are left and right address comparators which select a range of the line to enable for a write operation, and finally, there is a 1 bit wide mask plane which can prevent an enabled pixel from being overwritten.

Since there is just enough RAM to support the 24 bits of R, G, and B per pixel, we clearly need more RAM to hold the extra data, x_0 and β, for the LIRPs. The easiest way to accomplish this is to simply use a second QuickScan Line Buffer Chip (see Figure 4). Coordinating the addressing between the two Line Buffers is simple: just tie the address lines together. This way, whenever we are writing into the first Line Buffer with R_0, G_0, and B_0, we write to the very same locations in the second Line Buffer with x_0 and β, which is exactly what we want. The implication is, of course, that there is a separate data bus leading into the second Line Buffer, and a separate data bus leading out with the pixel data.

This is fine for taking care of our LIRP objects, but what of the plain old solid and pattern filled objects and bit map objects? As it turns out there is no trivial way to reroute individual pixels around the hardware in the output stage of QuickScan which computes the LIRPed value of R_0, G_0, and B_0 since it means cicumventing a long pipeline (to be explained below). So, if we load up R, G, and B in the first Line Buffer without considering what is stored in the second Line Buffer, we'll get unpredictable results when the hardware considers a R, G, and B pixel as R_0, G_0, and B_0 and computes the LIRPed value from the random x_0 and β which happened to be stored at the same pixel location. But, this can be easily remedied by either storing zero for β or the current x position for x_0. Then the formula, $C_{\Delta x} = C_0 + C_0\beta\Delta x$, reduces to $C_{\Delta x} = C_0 + 0$ if either $\beta = 0$ or $x_0 = x$, yielding R, G, and B from what the LIRP mechanism thinks is R_0, G_0, and B_0.

With such a double Line Buffer arrangement, we get an output of R_0, G_0, B_0, x_0, and β for each pixel shifted out of the two Line Buffers. What do we do with this data now that we have it? Since the QuickScan chips output data at a 50 Mhz clip (20 ns per pixel!), whatever we decide to do, it had better be fast. Damn fast. Well, what does the formula call for? $C_{\Delta x} = C_0 + C_0\beta\Delta x$, $\Delta x = x - x_0$. It looks like we need an addition, a subtraction, two multiplies, and some floating-point to fixed-point conversions for each of the three color components in 20 ns. To do that we'd have to build an ALU that is over 10 times faster than the Cray's. Fat chance. Maybe we ought to approach this problem from a little different angle.

Actually, if one looks closely at the way the Cray does its arithmetic, she finds that the Cray does not (and cannot) do a complex arithmetic operation such as a multiply or divide in a single machine cycle. Rather, it breaks these operations into stages and completes them in several cycles, but nonetheless, its average throughput for such operations is one cycle per each. How can this be? The reason is that its ALU is fully pipelined, which is to say that the next operation can be fed into the ALU one cycle after the previous operation was fed in. So, while the *latency* of complex ALU operations is several cycles long, the average *throughput* is one operation per cycle. One can think of such an ALU as an assembly line of workers, one at each stage of the production line. Each worker at any moment is working on a different assembly unit, which he then hands to the next worker, while he receives an assembly unit from the previous worker. The *latency* of the assembly line is the time to go through all of the workers' hands, but the average *throughput* is one assembled unit per the time to go through one worker's hands.

Thus, pipelined ALU's can be very effective because they take very complex operations and break them down into manageable atoms without reducing throughput. However, they are only effective when applied to steady streams of data for which the same operations are to be applied repeatedly. Otherwise, the pipeline gets empty stages (like a production line gets workers with empty hands), and the throughput decreases. Fortunately, the stream of pixels which will be output from the QuickScan Line Buffers and the operations needed to be applied to

them are ideal candidates for a pipelined ALU: the data stream is constant, and the operation applied to each is identical. So, while we cannot hope to apply the full complex LIRP computation in 20ns to each pixel, we nonetheless can achieve an average throughput of 1 computed pixel per 20ns.

Before we look at the actual pipeline, we need to first understand how certain complex operations are broken into pipeline atoms. To start with, at least for our pipeline, the atomic operations are a single addition/subtraction or a single data selection stage (i.e. selection of a bit of data from one of several sources). Data paths which don't change are hardwired and take "no time", and any number of independent atomic operations can occur simultaneously in one cycle.

A pipelined fixed-point multiply takes as many pipeline stages as bits in the multiplier. Effectively, the multiplicand is multiplied by 2 at each stage, and if the bit for that power of two is set in the multiplier, then the multiplicand is added to an accumulated sum. The multiplication works by at each stage examining the next most significant bit in the multiplier while it shifts the multiplicand another bit to the left (shifting in zeros). If the multiplier bit at that stage is a one, then the shifted multiplicand is added to an accumulated sum, if the multiplier bit at that stage is a zero then the accumulated sum is unaffected. Note that while the multiplicand is shifted by one at each stage, this is a hardwired shift and thus takes "no time" to complete.

Converting floating-point to fixed-point representation simply involves shifting the fraction part of the floating point number by the amount of the exponent. This variable shifting function can realized in a barrel shifter. A pipelined barrel-shift can be implemented with a input word and a shift amount word by log n stages of banks of 2-to-1 multiplexers, where each stage has one multiplexer for each bit of the input word and where n is the maximum number of bits to be shifted. At each stage of the barrel-shift pipeline the bank of 2-to-1 multiplexers either shifts the data by a degree of a power of 2 or does not shift the data at all, as controlled by the state of the bit for that power of two in the shift amount word. In a similar way to how the pipelined multiplier determines whether or not to accumulate for each power of 2 in the multiplier, the barrel-shifter determines whether or not to shift for each power of 2 in the shift amount input. I realize this is terribly confusing in words, but it just doesn't merit a diagram, so you'll have to trust me that it works.

Okay, the rest is easy. This is the pipeline to implement:

$$C_x = C_0\,(\beta(x - x_0) + 1) \quad \text{(algebraically equivalent to} C_x = C_0 + C_0\beta\Delta x\,)$$

for all three color components ("←" means "gets"):

Operations	Clock Cycles
$\beta fix \leftarrow$ Float-to-Fix(β)	5
$\Delta x \leftarrow x - x_0$	1

Temp1 <– βfix * Δx	10
Temp2 <– Temp1 + 1	1
R_x <– R_0 * Temp2; G_x <– G_0 * Temp2; B_x <– B_0 * Temp2;	8
	25

And that's it. After a 25 stage pipeline the correctly Gouraud shaded R, G, B values are output.

4.1. General LIRPs

If we examine the general LIRP function in which R, G, and B vary independently, it is not very hard to see how a system similar to the hue-invariant system discussed above might be implemented which realizes that function. Indeed, as it turns out, such a system is simpler, involving only one multiplication in the pipeline and no floating-point to fixed point conversion. The only drawback is that it requires more RAM in the Line Buffer.

If you recall, the formula for computing a LIRP for an independent color component is simply:

$$C_{\Delta x} = C_0 + m\Delta x, \text{ where } \Delta x = x - x_0.$$

This formula is similar to the hue-invariant formula, with the major differences being the lack of a second multiplication and a slope m instead of a complex constant β. We had stored one β for all three color components, but since each color component's slope is independent of the others', each one is stored independently in the Line Buffer, as

m_R, m_G, and m_B.

Also unlike β of the hue-invariant formula, these slopes can be compactly represented as signed, fixed-point numbers, each of 11 bits. Thus, we will require 33 bits of Line Buffer storage for the slopes. And, exactly as in the hue-invariant implementation, we will require 24 bits for R_0, G_0, and B_0, 10 bits for x_0, and 1 bit for the CLUT/RGB flag (explained in section 4.1). This gives us a total of 68 bits/pixel stored in the Line Buffer.

In the current QuickScan implementation this would require 3 QuickScan Line Buffer chips of 25 bits apiece. A next generation QuickScan device could feasibly, albeit awkwardly, support 34 bits/pixel providing 68 bits in 2 chips. And, indeed, some day a single 68 bit/pixel QuickScan chip will also be feasible.

The arithmetic for the general LIRP representation is simpler, but there are no common operations for the three color components. Thus, there are three separate and independent ALU pipelines operating simultaneously. In fact, they are so independent, the three pipelines could be implemented quite feasibly in three identical chips.

The following is the ALU pipeline to implement $C_x = C_0 + m(x - x_0)$. It is one of three identical pipelines for R, G, and B ("<–" means "gets"):

Operations	Clock Cycles
$\Delta x \leftarrow x - x_0$	1
$\Delta C \leftarrow m * \Delta x$	10
$C_x \leftarrow \Delta C + \Delta x$	1
	12

And, so we have it. A much simpler pipeline results than that of the hue-invariant implementation, but one which requires 68 bits of input data instead of 48.

Memorandum

To: Jonathan Architecture Group
From: Steve Perlman
Date: 21 February 1985
Subject: Strawman Proposal for the QuickScan Display Subsystem

Attached is a Strawman Proposal for an object-oriented graphics display system, QuickScan. This is the first release of the documentation for this system, and it is somewhat disorganized, but if you at least make it through the introduction, then poke through the technical specs of interest, you can get a pretty good feeling for the characteristics of the system.

I'll be supplementing this package with additional documentation, particularly some detailed descriptions on how to put up specific kinds of graphics objects, but in the meantime I'd be most appreciative to get any feedback you may have, and I'd be delighted to answer any questions.

QuickScan VRAM Bus Arbitration

SGP 2/19/85

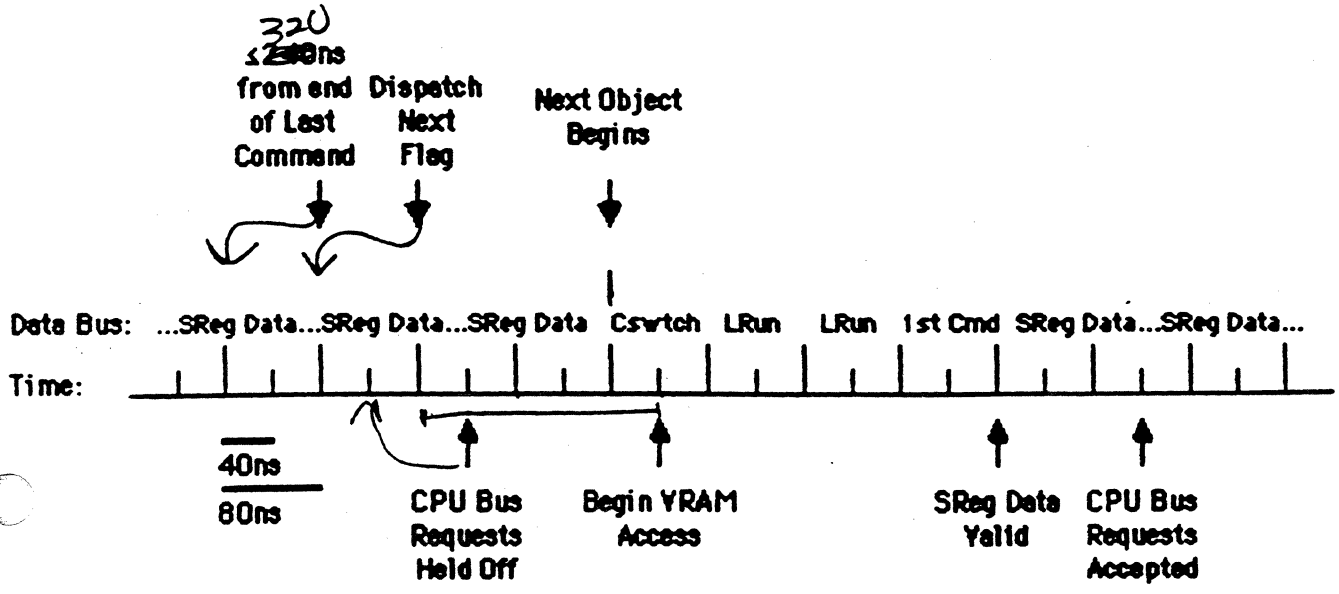
Assumptions:

1. NEC 256K VRAMs used.
2. CPU cycle time ≤280ns.
3. VRAM SReg Transfer Access time ≤280ns.
4. VRAM SReg Transfer Cycle time ≤400ns.

1. Bus Arbitration for Object Descriptions >1 Word in Length

The Line Buffer will detect internally when it is within 240ns of the end of the Last Command on a Line. This will either occur because a single word Command has its Dispatch Next bit set, or because a multi-word Command has its Dispatch Next bit set and it is within 240ns of its end. 80ns after this point, the Line Buffer will activate its Dispatch Next Flag. Within 40ns the Dispatcher will hold off any CPU bus requests (but of course will allow any in progress to complete).

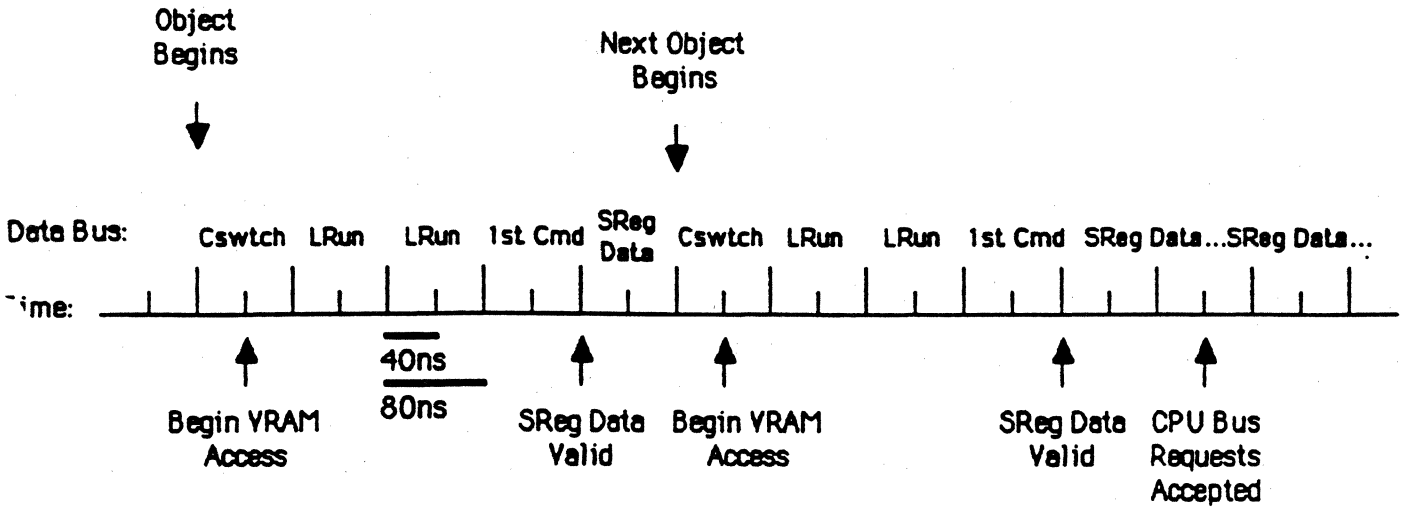
160ns after the Dispatch Next Flag, the next object will be dispatched with the Dispatcher sending a Context Switch Command. 40ns after this, the Dispatcher will commence the VRAM object data Transfer to the Shift Register. The Dispatcher will then send 2 LRun Commands (to set the Viewport) followed by the First Instruction for the object. The Shift Register data will be valid at this point, and the object load will commence. 120ns after this point, the VRAM access cycle will be complete, and CPU bus requests will be honored.



2. VRAM Arbitration for Objects of 1 Word in Length

Objects of exactly 1 Word in length are started exactly like longer objects, but instead of relinquishing control to the CPU after the VRAM Transfer to the Shift Register cycle completes, the Dispatcher retains bus control, and immediately begins the Transfer to the Shift Register. for the following object.

This allows very small objects (e.g. icons, pointers, background fills) to be processed efficiently.



QuickScan VRAM Refresh Generation

SGP 2/19/85

Since QuickScan does not follow a predictable row access pattern, it must periodically generate refresh cycles to keep the dynamic RAM intact. As it turns out, it is necessary to generate slightly more than 4 refresh cycles per line in 30Hz mode and slightly more than 2 refresh cycles per line in 60Hz mode. If we wanted to be clever, we could have QuickScan generate just 4 or 2 cycles, respectively, each line, then periodically insert an extra cycle, but its really little overhead to generate 5 or 3 cycles every line, so that's what I recommend.

The big question is: where do these refresh cycles fit in with the horizontal timing?

Well, clearly we prefer to interfere with the CPU's throughput rather than QuickScan's since we will be counting on the horizontal data load time to be very precise. Furthermore, the refresh cycle is the same length as a CPU memory cycle, yet different than a Shift Register Transfer cycle. From a state machine point of view, we'd again be better off interfering with the CPU.

There is still, however, the question of where. If a programmer chooses to hog all of the memory cycles on a line for QuickScan access, then she should be allowed to do so. Presumably, she would set up her code so that the CPU can be asleep for that line. Well, if that's possible, then where can we stick in refreshes during that line?

Well, the bottom line is: it's not possible. Let's construct the worst case scenario. It's in 60Hz mode, and she's set up all 64 objects so that they are each 1 word long, so as soon as the Dispatcher has fetched one, it immediately fetches the next, without letting the CPU get any cycles in between. Each of these Dispatch cycles is 400ns long, and 64 of them one after another amounts to 25.6 microseconds. The whole horizontal line is 31.778 µs in 60Hz mode, giving us 6.178µ left over. But, each refresh cycle is only 280 ns, and we need at worst 3 of them. Not only will we get our refresh, but we have 19 CPU cycles left over! Thus, QuickScan allocates the first 3 or 5 CPU cycles each line to refresh.

QuickScan Display Subsystem

Introduction

SGP 2/20/85

General Description and Sales Pitch

The QuickScan Display Subsystem is an object-oriented graphics generation system designed to structure graphics image representation in such a way as to relate an object's complexity to the amount of resources the object consumes. This approach tailors graphics resources to the exact needs of each object on the screen and saves us from accommodating the most general case with monstrous bit-maps, or even more monstrous graphics hacks

As a side-effect of this structuring process, QuickScan also provides us with a neater way of organizing our "frame buffer" by maintaining independent blocks for each of our graphics objects. We have the opportunity now to manage graphics memory as we do main memory, allocating it for graphics object needs as we do now for data structure needs, balancing the memory resource for the particular application at hand.

We also now have the capability to move large and complex images around the screen with little more than a change of a pointer. Sequencing through animation frames is accomplished instantly, with no redrawing or "undrawing" whatsoever. Objects with large spaces of a single color need not ever be uncompacted as QuickScan displays Run-Length Encoded (RLE) directly, and in fact displays runs of arbitrary length faster than any other display system available today.

QuickScan's bus interface was set up to be extremely general. It is capable of addressing 4 Megabytes of display memory directly, and it has hooks to be driven by graphics engines (like a 3-D polygon engine) while still displaying its conventional graphics. The nature of the system also makes it much simpler to genlock to an external video source for graphics overlays and underlays.

I have tried very hard to keep the system as general as possible so as to not lock programmers into a specific mode of generating graphics. I

regret that at this point I haven't had the time to write up lots and lots of examples to demonstrate the flexibility of the system. I believe, however, that as you tinker with what you have in mind to display, you'll find that there are routes within this system to get the image up, probably with less memory and more control than you thought. And, if there isn't a way to display what you envision, *then I want to hear about it.* There's a good chance there's something we can do about it.

Before we get into the nitty gritty of how this thing works, here's some specifications you can use to put the QuickScan approach in context with other graphics systems:

QuickScan General Specifications

Display Timing and Format:
- 640x484 60Hz non-interlaced or
- 640x484 30Hz interlaced, NTSC compatible
- Pixel clock independent of system clock
- External gen-lock and video underlay/overlay/middlelay capability
- Square pixels (with proper timing)

Output:
- Analog RGB
- NTSC Video
- (Stored internally as 5-6-5 RGB and 4-4-4 RGB with 4 bit multiplier.)

Object Capability:
- 2-1/2 D prioritization of 64 independent objects
- Objects are of arbitrary size and shape, displayed through arbitrary size and shape viewports
- Objects described through ~~bit-maps, run-lengths~~, or any combination
- Objects can be made of 5-6-5 RGB pixels, 8 bit lookup table pixels with a 4 bit multiplier, or parts of each mode 16 & 8 bpp
- Multiplier can be accessed independently to create luminance effects
- Bit-map depths supported are 1,2,4,8, and 16 bits/pixel (BPP)

Bus Interface Characteristics:
- Usually only interferes with CPU RAM access when starting an object description

Loading the object description occurs in full parallel with CPU access
Manages bus arbitration and dynamic RAM refresh
Uses NECμPD41264 256K Video RAMs

Performance Parameters:

Object dispatch overhead: 320 to 480ns/line of object
Bit Map overhead: 80ns/line of bit map
Bit Map draw rate:

1 BPP :	1 pixel each	2.5ns	(400 Million Pixels/second (MPS))
2 BPP :	1 pixel each	2.5ns	(400 MPS)
4 BPP :	1 pixel each	5 ns	(200 MPS)
8 BPP :	1 pixel each	10 ns	(100 MPS)
16 BPP :	1 pixel each	20 ns	(50 MPS)

Run Length overhead: 80ns/sequence of runs/line
Run Length draw rate:
80ns per run of arbitrary length at 16 BPP
(This figure cannot be compared with other graphics systems since they figure their runs in pixels/second. So...)
12.5 MPS min at 16 BPP
8000 MPS max at 16 BPP
4006 MPS ave at 16 BPP (a Cray can't write to memory this fast)

In case you need a basis of comparison for the above performance figures, consider the fact that we recently had a visit from a high-end graphics board manufacturer who was certain we'd be blown away by the drawing speed of their awesome new display chip. In its run length mode in certain conditions it could hit almost 50 MPS at 8 BPP, and in its bit-map mode it could get up to around 12 MPS at 8 BPP. This device used all of the bus time. The processor could not run in display RAM whatsoever. In addition, the device supported no objects. QuickScan needs very little bus time, supports 64 independent objects, and is significantly faster than their "state-of-the-art" engine.

But, to be fair, their system was much "smarter" than QuickScan. It could draw some simple figures as well as manage a display. (Even so, as Toby pointed out, our 68020 will blow their silicon away in complex drawing speed anyway.) This does, however, underscore a point. Unlike most of the recent display processors to hit the market, QuickScan does not have the ability to draw independently. It relies entirely on the CPU to give it

instructions effectively will be necessary in order to use the device effectively.

The object description is usually organized so that the instructions for a given line are immediately followed by the instructions for the next line which are followed by the instructions for the next line, and so on until the last line of the object (there are exceptions to this in advanced applications). Thus, the Start Address in VRAM pointer in the Object Dispatch Table (the ODT) should point to the instructions for the first line of the object, and instructions for each of the rest of the lines should follow in order.

Let's consider a simple example: the sky. The sky in this picture is light blue all the way across the screen. It happens to be object zero because it is the background-most object. Well, it starts at the top of the screen, and continues down to line 200 before it is covered by the water. So, we specify the Start Line as 0 and the End Line as 200. Horizontally, the sky begins at pixel 0, so let's specify the Absolute Origin to be 0. Let's put our object description at address 100 in RAM, so we set the Start Address parameter to the value of 100. That takes care of the ODT. Now, let's prepare the object description.

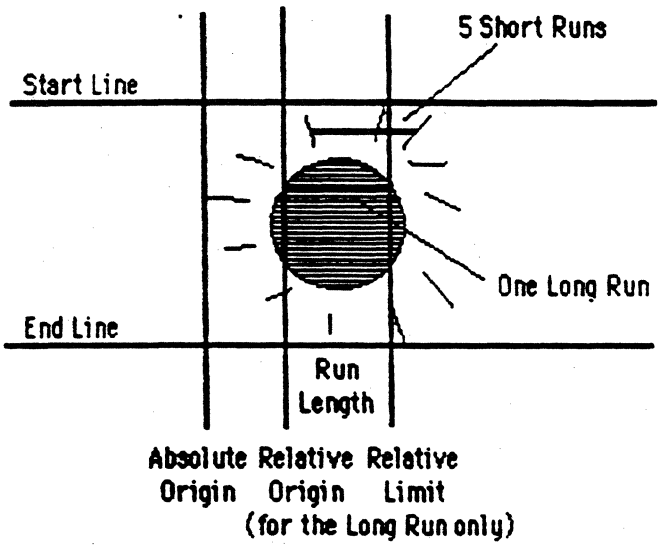
Since the sky is the same color (in this simple example, anyway) all the way across the screen, it is the ideal situation for using a Run Length. So, our very first instruction is a Run Length of light blue from pixel 0 to pixel 640. And, that's it for the first line. By setting a bit (the Dispatch Next bit) in the Run Instruction we let QuickScan know that we are all done for the line. We place the instruction for the next line immediately following the instruction we just put in, and sure enough it's the same exact instruction since the sky is light blue straight across on this line as well. Like the first, we set the Dispatch Next bit so that QuickScan realizes that it is at the last instruction in a line, and then we follow it with the Run instruction for the next line, the next, and so on until we have enough instructions for every line in the object. We don't need to tell QuickScan that we have reached the end of an object description. It determines this from the End Line value in the ODT.

Since we have 201 lines in this object, we'll need 201 Run instructions to describe it. Each Run instruction takes 1 word (32 bit word), so the whole light blue sky object takes only 201 words, yet it

contains some 128,000 pixels. Indeed, when you get more advanced in using QuickScan, you'll find there is a way to draw the whole light blue sky with just 1 word in the object description!

Let's now consider something a little more tricky, the sun. This particular sun object extends from line 40 to line 180, so that tells us that Start Line and End Line in the ODT should be set to 40 and 180, respectively. Unlike the sky object, however, the sun is not aligned with the left side of the screen, its left-most rays extend only to pixel 400. That tells us that its Absolute Origin should be set to 400. From now on, all horizontal coordinates we specify in relation to this object will be referenced to pixel 400. Let's have this object's description start at address 3000 in RAM, and we will set the Start Vram Pointer accordingly.

Refer the following diagram as we discuss how we set up the object description.



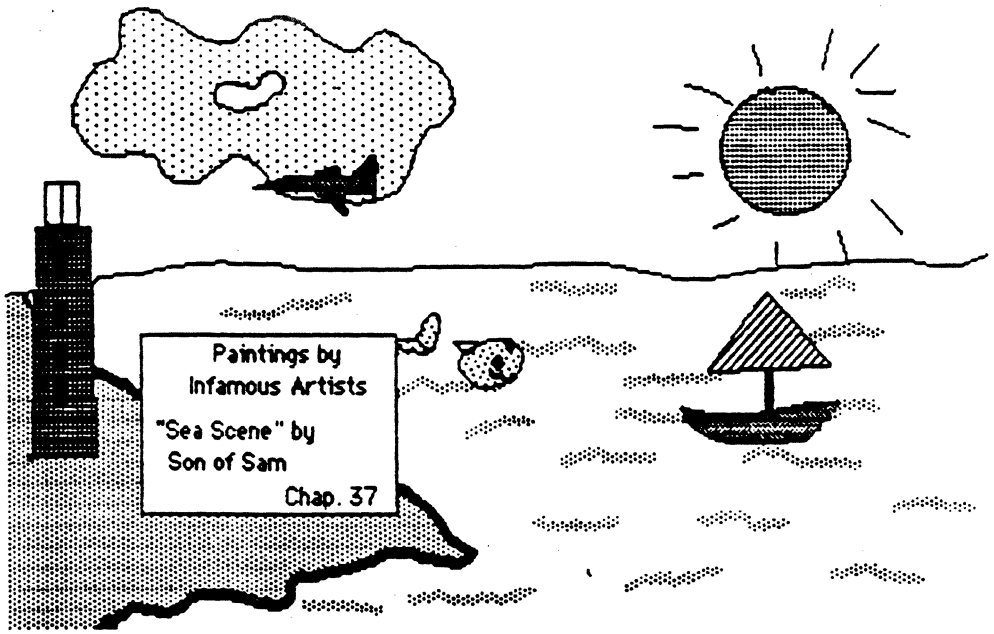
The Absolute Origin as a start position for all runs was sufficient for the sky object because all runs began at the same point on every line. Let's consider just the ball part of the sun object for the moment. The first thing we see is that the Absolute Origin designation is insufficient for the runs (i.e. the horizontal lines) that make up the ball because each run on each line begins at a different point. To accommodate this characteristic QuickScan supports a second Origin local to each subunit on

direction. QuickScan only provides a structured image space model for the CPU, and through this organization eliminates some space- and time-consuming operations otherwise necessary with a vanilla bit-map.

But we're jumping ahead. QuickScan can best be introduced with an example.

A QuickScan Example

Consider the following scene:



This scene can be considered to be made up of 11 objects. These are, listed background to foreground: The sky, the sun, a cloud, a jet, water, waves, a fish, a ship, land, a light house, and a text window. You could specify each of these objects as an independent entity to QuickScan, and given the appropropriate instructions, it would generate a composite image just as you see above. Let's look into how we would do that.

First of all, we have to present the object list to QuickScan in the order in which we'd like to see the objects prioritized, background to foreground (just as we listed the objects above). The order is significant because if we decided that we wanted to move an object we'd want it to

appear on top of the objects behind it and behind the objects in front of it. (For example, we'd want the plane to appear in front of the Sun, but not in front of the text window.) The name of this ordered object list is the Object Dispatch Table, and we may build this table at any address in the 256K Video RAM that is an even multiple of 1K.

Each object is provided with a 4 word (32 bit word) entry in the object Dispatch table. The background-most object uses the first entry, the next-to background-most gets the second entry, and so on until the foreground-most object has been entered. QuickScan supports up to 64 objects per frame, but you do not have to use them all. In this case we specify only 11, and that is fine.

Each Dispatch Table entry has enough information to tell QuickScan what it needs to know about the position and the characteristics of the object the entry refers to. If you want to jump ahead there is a diagram of the full entry format, but for our concerns right now I'll just discuss the basics.

To begin with, there is a Start VRAM Address pointer which tells QuickScan where in RAM it can find the beginning of the object's description. Next there are 2 values, Start Line and End Line, which tell QuickScan between which lines of the screen the object is to be displayed. And finally (for our purposes) there is a value called the Absolute Origin which tells QuickScan what horizontal point in screen space it should use as a reference point for positioning this object left and right.

The object description is a line-by-line sequence of instructions that tells QuickScan how to draw the object. Don't worry! These aren't instructions like you've come to expect from a microprocessor or a high level language. They are just very simple primitives which instruct QuickScan to draw either Bit-Maps or Run Lengths, nothing fancy. Also, don't think you need a sequence of instructions all of the time. If all you have to display is a plain old rectangular bit map, or a regular sequence of runs, then you just have to store the data. You don't need to worry about instructions at all. Nonetheless, it is important to understand that QuickScan <u>is</u> an instruction-driven machine; the rectangular bit map happens to be a simple case where the instructions are effectively hidden. As you get into the more complex applications of QuickScan, utilizing the

each line of an object description, the Relative Origin. The Relative Origin shown in the diagram is associated with only the particular run of the sun highlighted in a thick black line. The run above it, the run below it, and indeed every other run (or, as we'll see shortly, run sequence) in the object description has its own particular Relative Origin.

The Relative Origin is to object coordinates exactly as the Absolute Origin is to screen coordinates. That is to say, just as the Absolute Origin defines an offset from the left edge of the screen, the Relative Origin defines an offset from the left edge of the object (which is defined by the Absolute Origin). So, if at any time you need to know the screen position pointed to by a Relative Origin, you simply add it to the Absolute Origin and you'll get the exact pixel position on the screen.

Now, that we've specified the start of the run, we need to specify its end. There are 2 ways to do this: either specify its Run Length or specify its Relative Limit. The Run Length says how long the run is, and the Relative Limit says where the run ends (relative to the Absolute Origin). There are reasons for specifying runs in either of these ways, and as you get to QuickScan nitty-gritties, you find there are a few other implications. But the end result, whichever way you specify it, is the same.

So, if we are just drawing the ball part of the sun object , we find that we once again need only one instruction per line (the Relative Origin is specified in the Run instruction), only unlike the sky object's one instruction, ~~each one~~ the ball objects instruction is different for each line, reflecting the varying shape of the object. But, we still are faced with the problem of the suns rays. How do we describe these strangely shaped things?

The way to approach the problem is to consider how QuickScan sees the rays: it sees them each line-by-line, so it is only concerned with the individual pixel or pixels which appear from the rays on each line. Now, we could specify a small bit-map for each of the individual pixels that appear on each line, but that would be somewhat wasteful of RAM since the rays themselves have no internal details. We might as well use the run generation facility to simply make short runs to draw the individual pixels, using the Relative Origin to position a run at the intersection of each ray with each line.

Now it is perfectly reasonable to generate an individual run instruction for each of these little rays, but there is another approach. It is sort of a Run Length shorthand useful in describing a sequence of runs (this case isn't the greatest example, though -- it's especially handy with cartoons). In the diagram I've shown a sequence of 5 short runs. The first run is a teeny one to draw the first ray's intersection with the line, the second run is a "transparent" run which just skips over to the next ray, the third run draws the second ray, there is another "transparent" run, then finally, the last run draws the third ray. The short run sequences encode in roughly half the memory space of the individual long runs (although they take just as long to draw), and they make it possible to tie adjacent areas of color together. For example, if I change the Relative Origin of the run sequence all 5 runs are affected, but their position relative to each other stays the same.

Whatever approach we decide to use to encode the rays we are now faced with the problem of combining the rays with the ball part of the sun. This can be handled in 2 ways. First, we could keep the ball runs and the ray runs independent. In this case each line of the object description would have first one run instruction, and then would end with the other run instruction. If the rays require several run instructions for that line then these can be inserted in any order. The key thing is make sure all instructions get in before the end of line bit (the Dispatch Next bit) is put in. Second, we could encode the entire line as a sequence of runs, including the run defining the ball. Then, we'd have just one instruction per line, and the description would be very neat (though not necessarily optimally compact).

In any case you may have noted that we no longer have a uniform number of words per line of object description. If you are wondering, no, it's not a problem. QuickScan will simply count the words until the Dispatch Next flag is picked up, then will update its internal state accordingly.

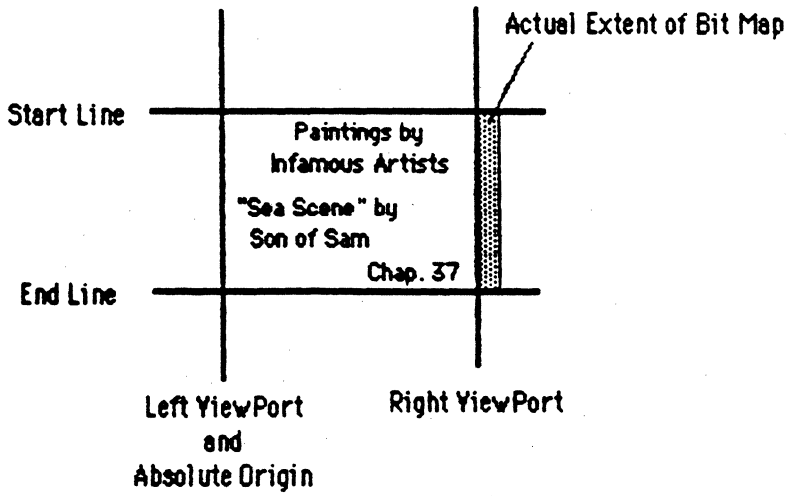
So, how many words would the full sun object description require? It is 140 lines tall, and considering how I drew it, I figure there's about an average of 3 runs per line. Encoding each of the runs individually, we'd use exactly 1 word per run. So, that's 3 words per line times 140 lines gives us about 420 words. Not too bad for an object that takes up about 1/6 of

the screen.

Okay, suppose that after we'd defined the sun as above we wanted it to set. How would we go about it? No problem. If you recall we defined the sun to be way in the background; it moves in front of only the sky. So, if we reposition it lower in the screen, it will be overlapped by any objects which are positioned in front of it, in this case, by the water. Repositioning it vertically only requires changing its Start Line and End Line parameters in the ODT. The object description and everything else remains the same. If, for some reason we wanted to reposition the sun horizontally behind the cloud, then all we have to do is change its Absolute Origin to some lower value. Since the object has been described relative to this parameter, the various parts of the object will move to left along with the Absolute Origin, maintaining the same horizontal spacing among themselves.

Okay, let's jump ahead and take a look at how we generated the Bit Map object (the text window) in the very foreground. To understand this we need to unveil 3 more parameters of the ODT, the Left ViewPort, the Right ViewPort, and the First Instruction.

Refer to the diagram below for the following discussions.



Although representing complex objects with instructions provides us

with a useful organization, representing plain old bit-maps with instructions could be very cumbersome. We want bit-maps to be stored linearly in memory, with the last word of one line being followed directly by the first word on the next line. The QuickScan system, as it's been so far described, minimally requires one instruction on each line. If we expect to have linear bit-maps as described above, imbedding an instruction prior to each line of bit-map is out of the question (besides that QuickDraw would have a bird).

To get around this problem (and also help out in other ways) there is a First Instruction parameter for each object in the ODT. This instruction is the first instruction executed at beginning of every line for the entire object description, regardless of what data or instructions are to follow on a particular line. Now, an obvious question is, what if you don't want the same first instruction on every line? Then, you'd make the First Instruction a NOP and there'd be no problem.

For our concerns with bit-maps it so happens that the Bit Map instruction for every line of a linear bit-map is exactly the same. And the data for the Bit Map instruction is set up in such a way that its meets the linearity criteria set above in the way that it is organized in RAM. Thus, the plain linear bit-map is a specific case that falls out of the QuickScan general object description format.

The only constraint that QuickScan does impose upon bit-map organization is that each line of the bit-map must end evenly on a 32 bit word boundary. Now this doesn't mean that all bit-maps that QuickScan displays must have horizontal dimensions in multiples of 32 bit words (as we'll see in a minute). It simply means that if your horizontal dimension ends up with some fraction of a 32 bit word, then you have to waste the remaining number of bits in the word to even out the line. Presently, QuickDraw stores bit-maps aligned to 16 bit boundaries. I don't imagine the change to 32 bits would be enormously difficult (famous last words).

Considering the text window diagrammed above, we see that it bears many similarities to the sun object description. Like the sun object, the Start Line and End Line parameters define the vertical limits of the object, the Absolute Origin parameter defines the left limit of the object, and (not diagrammed) the Start VRAM Address parameter points to the

start of the object description (in this case it points to the first word of the linear bit-map stored in RAM). And, like the sky object, all lines begin at the same pixel so there is no need to specify a Relative Origin for each line. What distinguishes the ODT entry for this object from the others is its ViewPort parameters.

A ViewPort is just what it sounds like it is, a limited view into another space. QuickScan has an extremely general ViewPort facility which allows us to specify ViewPorts of arbitrary shape and size (the cloud, for example, could be a ViewPort into a live video image), but for the most part we only need rectangular ViewPorts. Folks at Apple call such things "windows."

The rectangular ViewPort is so common in AppleLand that it seemed to me that it would be good marketing sense to include an automatic rectangular ViewPort facility as part of each object dispatch. Seriously, though, such a capability is fundamental when working with bit-map objects anyway. Which leads us back to the problem at hand:

It so happens that this particular bit-map has a horizontal dimension of 230 pixels. It is a 1 bit/pixel bit map so it takes up 7 words and 6 bits for each line. As stated before QuickScan requires each horizontal line to end exactly on a 32 bit word boundary, so we can say that this object has 8 words per line with the last 26 bits of the 8th word unused.

If we simply draw this bit-map object with blind abandon we will find out that those 26 bits had some value, and they will clobber whatever should have been directly to the right of the true bit-map image. This is where the ViewPort fits in at its most simple application: it crops the unused bits off of bit-maps so that only the true bit-map data makes it to the display. Thus, by setting the Right ViewPort parameter in the ODT to 230 (yes, it is relative to the Absolute Origin), we will crop the unused 26 bits off the bit-map, and we will see displayed only the data in the true bit-map.

So, how do the Left, Top and Bottom ViewPorts fit in, and how do we specify the Top and Bottom if there is no direct parameter? Well, the Left ViewPort is not needed in this example so we tuck it away out of trouble at the Absolute Origin. It is used, as well as the Top and Bottom

ViewPorts, but we'll have to wait till a little later to get into it. To give you a hint at some of the possibilities, you get horizontal and vertical scrolling within the your rectangular (or any shape) ViewPort without having to move any data around.

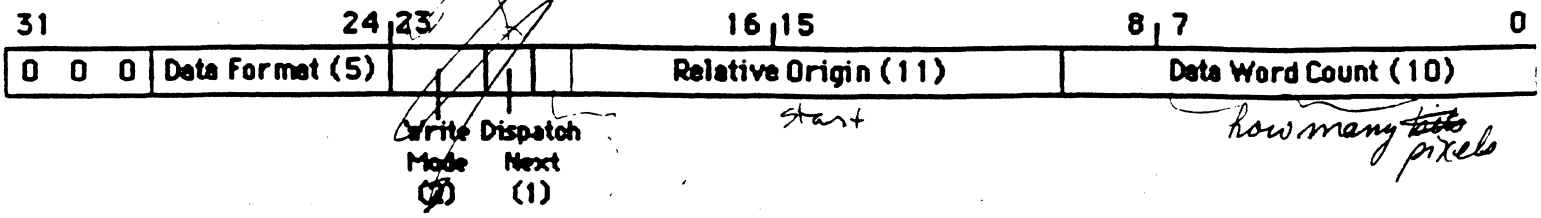
Okay, this is where I'm going to leave off describing QuickScan for this release. I realize there are many unanswered questions, but there is enough here to start on until I have time to get the rest out.

QuickScan Line Buffer Instruction Set

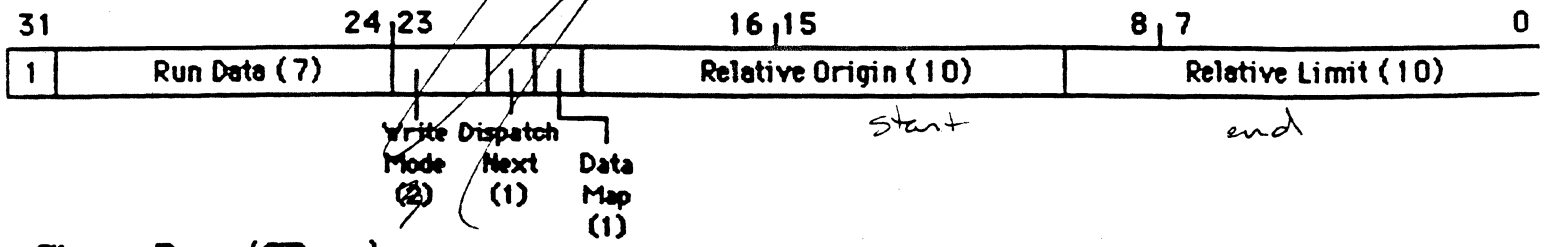
Command Word Format

SGP 2/17/85

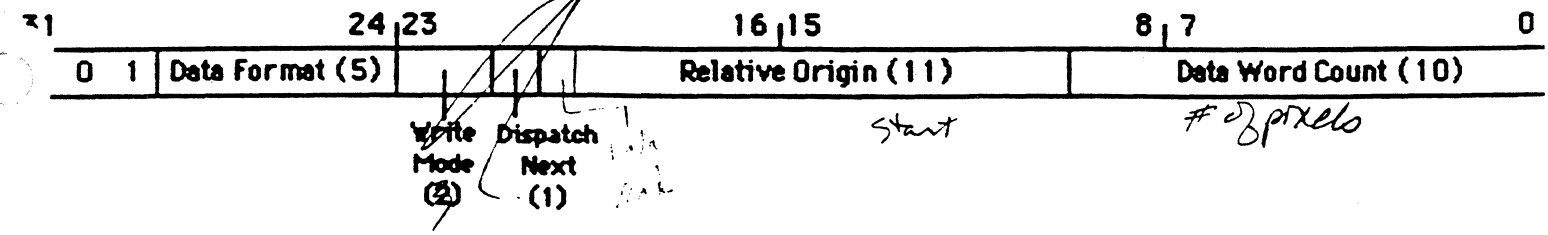
Bit Map (BMap)



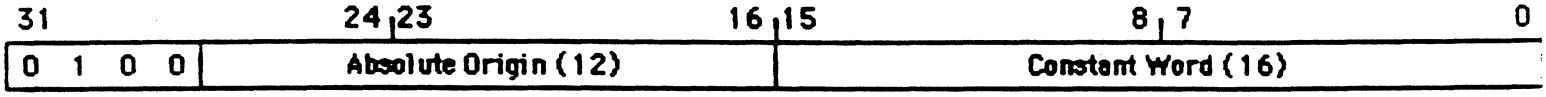
Long Run (LRun)



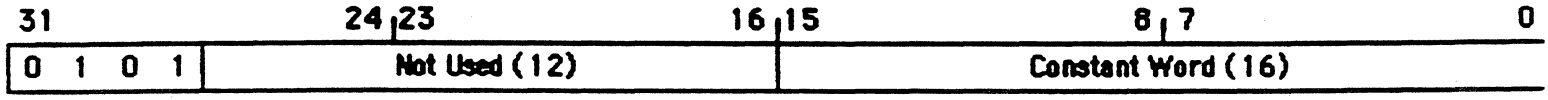
Short Run (SRun)



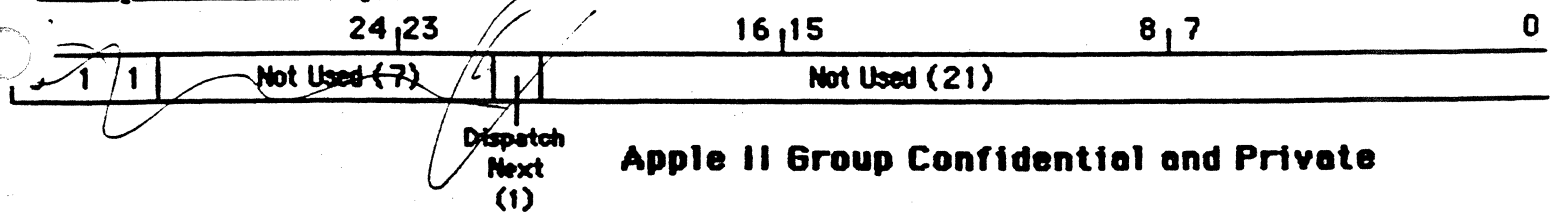
Context Switch (CSwitch)



Replace Constant (RConst)



No Operation (NOp)



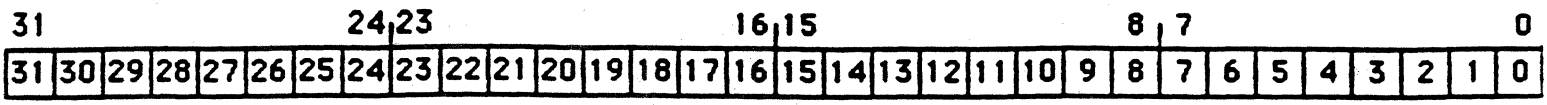
QuickScan Line Buffer Instruction Set

Data Word Format

SGP 2/17/85

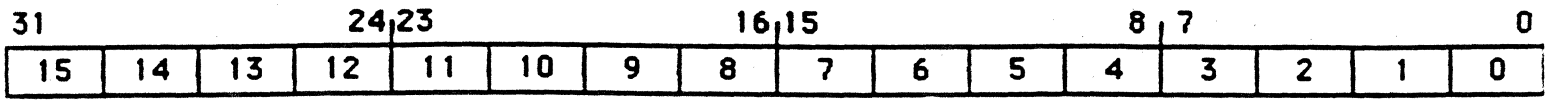
Bit Map Data Word Formats

1 Bit/Pixel



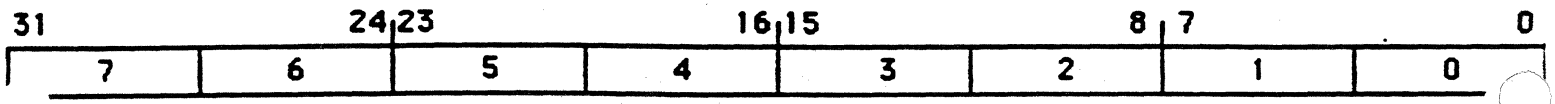
32 1-Bit Pixels

2 Bits/Pixel



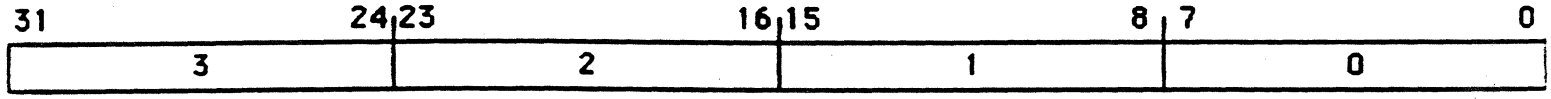
16 2-Bit Pixels

4 Bits/Pixel



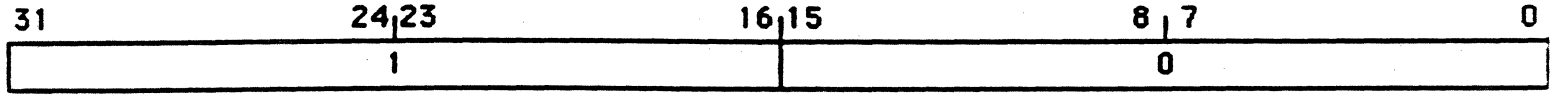
8 4-Bit Pixels

8 Bits/Pixel



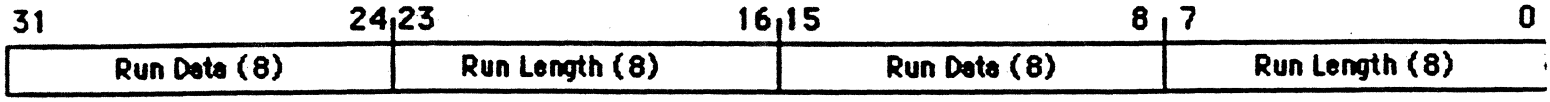
4 8-Bit Pixels

16 Bits/Pixel



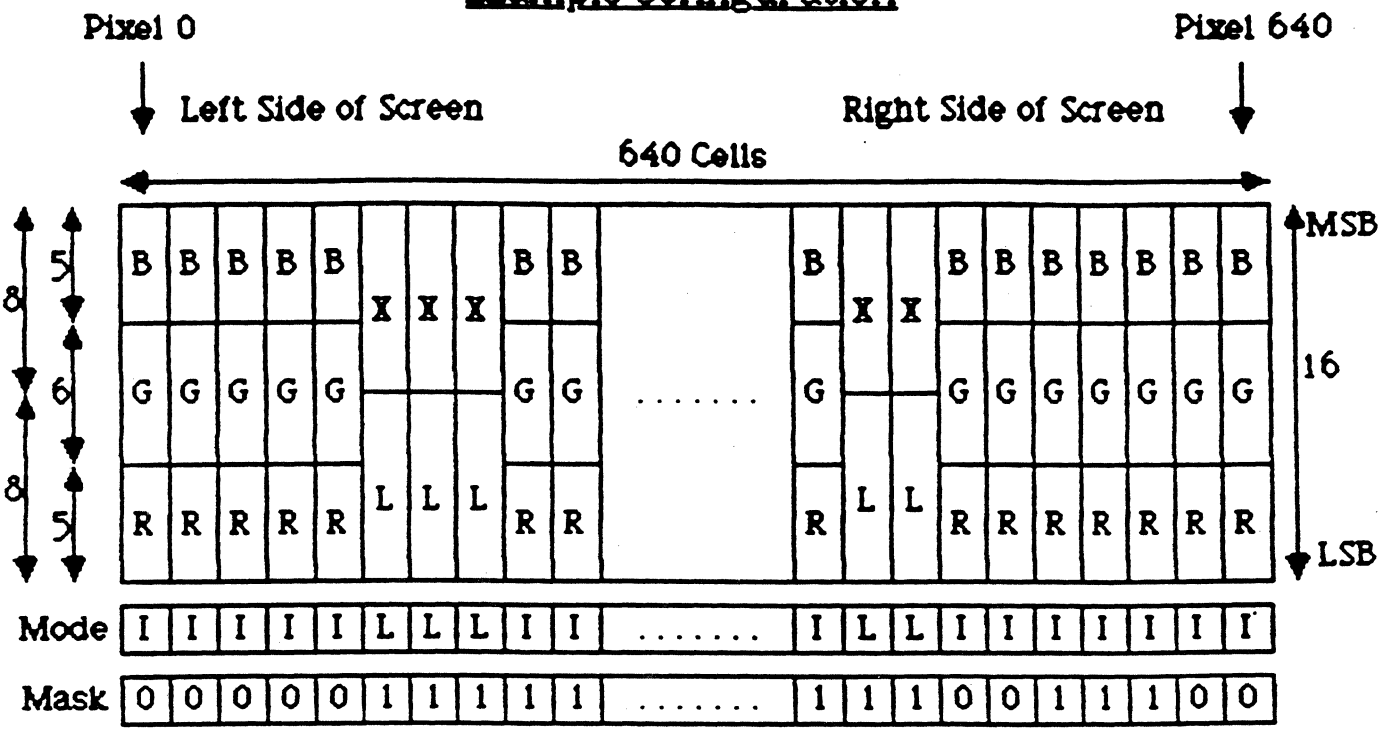
2 16-Bit Pixels

Short Run Data Word Format

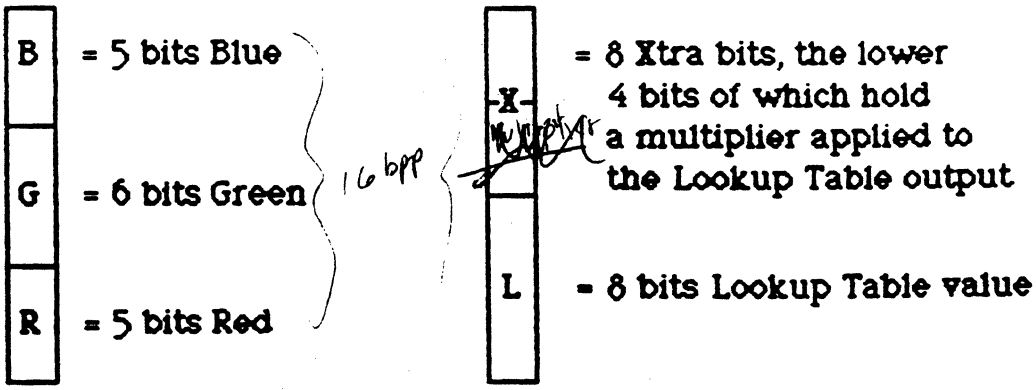


2 Short Runs

Example Configuration



Cell Descriptions



QuickScan Line Buffer

Programming Model

SGP 2/19/85

QuickScan Line Buffer Instruction Set

Instruction Descriptions

SGP 2/20/85

The following are descriptions of the 6 instructions supported by the QuickScan Line Buffer. All data access to the Line Buffer is carried out through these instructions, and understanding them is fundamental to understanding how QuickScan objects are displayed.

This document discusses the overall effect of each of the instructions. Refer to the related Line Buffer Instruction Set documents, Command Word Format, Data Word Format, and Field Descriptions for diagrams and further details.

Instructions execute in the same time that they take to load, so only instruction load times are given.

Context Switch

CSwitch <Absolute Origin>, <Constant Word>

The Context Switch single word instruction redefines the Line Buffer Absolute Origin and Constant Word, generally in preparation for a new object description (refer to the Field Description document for an explanation of the Absolute Origin and Constant Word). This instruction is automatically generated by the Dispatcher when dispatching a new object, but can also be specified within an object description for some other purpose.

This instruction takes 80ns to load and cannot be the last instruction in an object description.

Replace Constant

RConst <Constant Word>

This single-word instruction replaces the value of the Constant Word. It is functionally equivalent to Context Switch except that it does not affect the Absolute Origin.

This instruction takes 80ns to load and cannot be the last ~~Command~~ instruction in an object description.

Bit Map

BMap <Data Format>, <Write Mode>, <Dispatch Next>, <Relative Origin>, <Data Word Count>

This multiword instruction provides the means to display Bit Map images. A single Command Word describes the characteristics of the Bit Map data (the Data Format), the origin of the Bit Map relative to the Absolute Origin (the Relative Origin), and the number of Data Words to follow (the Data Word Count). The Command Word is then followed directly by the specified number of Data Words, and these words provide the raw data necessary to generate the Bit Map. A Bit Map instruction may be the final instruction for an object (i.e. by using its Dispatch Next bit).

QuickScan supports 5 different bit depths in its Bit Map displays. These are: 1, 2, 4, 8, and 16 BPP (Bits/Pixel). Although the Bit Map Command Word is the same for all depths, there are differences in the Data Words. First of all, pixels are packed in different densities in the various formats. Secondly, the rate that Data Words load into the Line Buffer varies between the formats. The results are summarized below:

		Load Time (ns)		
Depth (BPP)	Pixels/ D. Word	Command Word	Data Words except last	Last Data Word
1	32	80	80	80
2	16	80	40	80
4	8	80	40	80
8	4	80	40	80
16	2	80	40	80

(Note that it takes an equal amount of time to load a 2 BPP Bit Map as a 1 BPP Bit Map, even though it's twice as much data.)

Like all QuickScan ~~Commands~~ instructions, Bit Map loads an image for a single line only. If more than one line of Bit Map is desired, then either a Bit Map Command instruction

must be specified for each line, or the Bit Map ~~Command~~ instruction must be the First ~~Command~~ instruction in the Object Dispatch Table Entry.

Long Run

LRun <Run Data>, <Write Mode>, <Dispatch Next>, <Data Map>, <Relative Origin>, <Relative Limit>

This single word instruction loads one run of a single input data value (the Run Data) into the Line Buffer. The run may be up to 1023 pixels long. All pixels from the beginning to the end of the run will be written with the given input data value at once, and no other pixels will be affected.

A run is specified by its left limit relative to the Absolute Origin (the Relative Origin), and its right limit-1 relative to the Absolute Origin (the Relative Limit). If a run's right limit is specified to be to the left of it's left limit, then the run is ignored.

This instruction takes 80ns to load, and it can be the last instruction in an object description.

Short Run

SRun <Data Format>, <Write Mode>, <Dispatch Next>, <Relative Origin>, <Data Word Count>

This multi-word instruction loads a sequence of consecutive runs into the Line Buffer. The first run begins at an origin specified relative to the Absolute Origin (the Relative Origin) and writes an input data value (the Run Data) at once to a number of pixels (the Run Length) to the right of the origin. The second run begins at the pixel following the last pixel written by the first run and writes its input data value to a number of pixels to the right of that point, and the process continues until each of the runs has been loaded (the Data Word Count+2). Runs of zero length are ignored, and processing continues with the following run.

Runs can be a maximum of 255 pixels long, and 2 runs are encoded in each Data Word. If an odd number of runs is desired, then the second run of the last Data Word should have a length of zero.

Runs can be "transparent." That is to say, a run can be specified which extends across a number of pixels but does not write anything to these pixels. This comes in very handy when there is a sequence of runs, a

gap, and then another sequence of runs. The gap can be crossed with a transparent run, continuing the same Short Run instruction into the second sequence. Transparent is specified by a Run Data value of 255.

The Short Run Command Word takes 80ns to load, and each Data Word takes 160ns. It can be the last instruction in an object description.

No Operation

NOp <Dispatch Next>

This single word instruction is a place marker; it has no function. NOp takes 80ns to load, and it can be the last instruction in an object description.

QuickScan Line Buffer Instruction Set

Field Descriptions

SGP 2/17/85

Absolute Origin

This 12 bit word defines the horizontal display space origin from which all object positioning calculations will be made. It is a 2's complement number with the leftmost pixel (pixel 0) on the screen mapped to position 0, increasing with positive values to the right and decreasing with negative values to the left. Thus, objects can be positioned relative to a point up to 2048 pixels to the left of the screen and to a point up to 1408 pixels (+2047 less 640 plus 1) to the right of the screen.

There is more room provided on the left side of the screen because objects are always generated left-to-right, never extending further left than the Absolute Origin, and thus, we need more room to move objects off-screen on the left than the right. The screen position is maintained internally in the Line Buffer in such a way that an object extending past pixel +2047 will not wrap around to the left side.

Relative Origin

This 10 or 11 bit word defines the pixel offset from the Absolute Origin at which to begin writing the forthcoming data. In the case of a Bit Map ~~Command~~, this defines the pixel addressed by the first Bit Map Data Word, and in a Run ~~Command~~, this defines the leftmost pixel of the first run. The Relative Origin word is a positive integer, summed internally with the Absolute Origin to get the resulting pixel address.

Note that the resulting pixel address from the sum of the Absolute and Relative Origins need not actually be on-screen for the ~~Command~~ to be executed apppropriately. If, for example, a ~~Command~~ specifies its Relative Origin to be to the left of the screen, and part of its generated image is off-screen and part of it is on-screen, the QuickScan Line Buffer will generate the on-screen part of the image appropriately, even though the screen border may fall right in the middle of a run or a Bit Map word. If, however, the resulting pixel address is off the right side of the screen, there is no on-screen part of the image, and QuickScan will just skip the ~~Command~~.

Constant Word

When the Line Buffer is forming a 16 bit word to write to a pixel or a ~~group~~ run of pixels, it has to provide a full 16 bits for the write operation even though the input data may provide less than 16 bits. The Constant Word provides these additional bits. Its function is best described by an example:

If the Line Buffer is loading in a 4 bit/pixel Bit Map, then the input data is providing 4 bits to write to each 16 bit pixel. The Bit Map attribute field "Data Format" indicates to the Line Buffer which 4 bits of the 16 in each pixel cell the input data refers to, but it is still faced with the problem of what values to assign to the remaining 12 bits. This is where the Constant Word comes in. Whichever 12 bits happen to not be specified by input data (after the data has been formatted) come directly from the corresponding 12 bits of the Constant Word. So, if the input data provides bits 0 through 3, then bits 4 through 15 would be provided by bits 4 through 15 of the Constant Word.

The same applies analogously to input data widths of 1,2,7, and 8 bits. Note, however, that at 16 bits/pixel, the Constant Word is not used at all.

Data Format

This field indicates the input data width and alignment in the 16-bit pixel word. Available widths are 1,2,4,8, and 16 Bits/Pixel (BPP). An alignment value of 0 indicates the data bits are aligned flush with the LSB of the pixel word (Bit 0 of L-Byte), and increasing alignment values place these data bits incrementally closer to flush alignment with the MSB of the pixel word. Input data can only be aligned on bits which are multiples of its width (e.g. if the data width is 4 BPP, then there are 4 alignment positions, but if the width is 1 BPP, then there are 16 alignment positions). Those bits of the pixel word not provided by the input data are provided from corresponding bits of the Constant Word.

Width	Encoding
1 BPP	1 A A A A
2 BPP	0 1 A A A
4 BPP	0 0 1 A A
8 BPP	0 0 0 1 A
16 BPP	0 0 0 0 1

Where AA... is the alignment value.

Write Mode

This field indicates which sections of the pixel word are to be affected by the forthcoming data writes. ~~All~~ sections not selected in the write mode will not be affected at all by the forthcoming writes.

When M write mode is selected (for writing to the Mask and Mode bits), the Mask bit is written by bit ~~0~~ of the resulting 16 bit pixel word and the Mode bit is written by bit ~~1~~. Bits ~~2~~ through ~~15~~ are ignored. If the data width is 1 BPP, the data writes will affect only the Mask Bit; the Mode bit will be left as is.

Mode	Encoding	Sections Written
M	0 0	Mask and ~~Mode Bits Only~~
L	0 1	L-Byte Only
X	1 0	X-Byte Only
LX	1 1	L- and X-Bytes Only

Data Map

This field of the Long Run ~~Command~~ Word provides a limited means of mapping the 7 bits of Run Data in the 16 bit internal pixel word. If the Write Mode is L, X, or LX, then when the Data Map bit is 0, it will map the Run Data to the lower 7 bits of the 16 bits, and when it is 1, to the upper 7 bits. If the Write Mode is M, then when the Data Map bit is 0, it will limit the Run Data to 1 BPP with its LSB mapped to internal bit ~~0~~ (thereby restricting it to the Mask bit), and when the Map bit is 1, it will map the Run Data directly to the internal lower 7 bits (thereby allowing it to affect the Mask Bit in bit ~~0~~ and the Mode Bit in bit ~~1~~).

Run Data

~~This~~ field provides an imbedded 7 or 8 bits of input data in a Run ~~Command~~ to be used in generating a word for writing to the group of pixels addressed by the run. The use of this field varies between the Short and Long Run Commands.

For Short Runs, this field is 8 bits long. It is formatted into the pixel word following the width and alignment rules given above in the "Data Format" paragraph -- with the following restriction: 16 BPP mode should not be used (the resulting pixel value is indeterminate). If input data widths of less than 8 bits are specified, these are taken from the Run Data

aligned to the LSB, and the upper bits of the Run Data are ignored.

For Long Runs, the 7 data bits are mapped based on the value of the Map Data bit. The remaining 9 bits are provided by the corresponding bits of the Constant Word.

Relative Limit

This 10 bit field of a Long Run Command Word indicates the pixel number-1, relative to the Absolute Origin, which is the last pixel of a run. All pixels from the Absolute Origin+Relative Origin to Absolute Origin+Relative Limit-1 will be written in accordance with the Write Mode selected.

Note that Relative Limit does not specify "Run Length" but rather specifies a fixed pixel position. If a Run Length is desired, then one or more Short Runs must be specified.

Run Length

This field, as its name implies, specifies the length of a run. Or, more precisely, it determines the right limit of a Short Run relative to the Run Start, an internal register initially loaded with the sum of the Absolute and Relative Origins. By summing the Run Length with the Run Start value, the Line Buffer calculates this right limit. After the run has been written to the pixel cells (using the Run Start value as the left limit), the Run Start register is loaded with the right limit value, and this value becomes the Run Start for the next run in the same Short Run ~~command~~. Note that Run Lengths of zero are degenerate and neither write anything to the line buffer nor change the value of the Run Start register.

Subsequent runs do not overlap even though the right limit of the first becomes the left limit of the second. This is because the left limit points to the pixel of its value, and the right limit points to the pixel of its value-1.

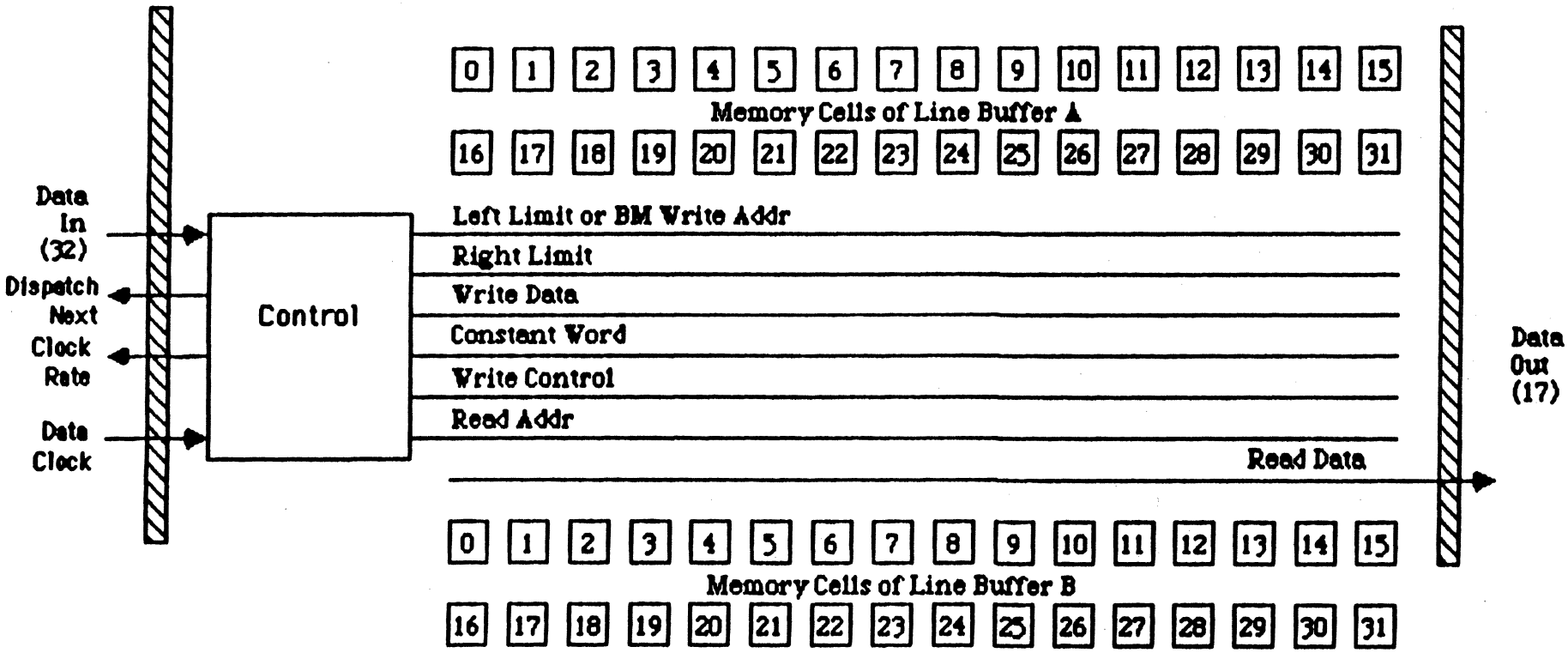
Dispatch Next

This bit, when set in a Command Word, notifies the Line Buffer that the ~~Command~~ is the last on this line for the currently loading object, and that the Dispatcher should be signaled to dispatch the next object. If the ~~Command~~ is a single word ~~Command~~, then the Line Buffer will signal the Dispatcher immediately, but if it is a multiple word command, the Dispatcher will be signaled 240ns before the ~~Command~~ ends.

When the Dispatch Next bit is set in a single word Command, there is a 160ns delay before the next object is dispatched, with multiple word Commands (except 2 or 3 word ones) there is no delay.

Data Word Count

This value indicates the number of Data Words less one to follow the Command word. Thus, 0 indicates 1 Data Word shall follow, and 255 indicates 256 Data Words shall follow.

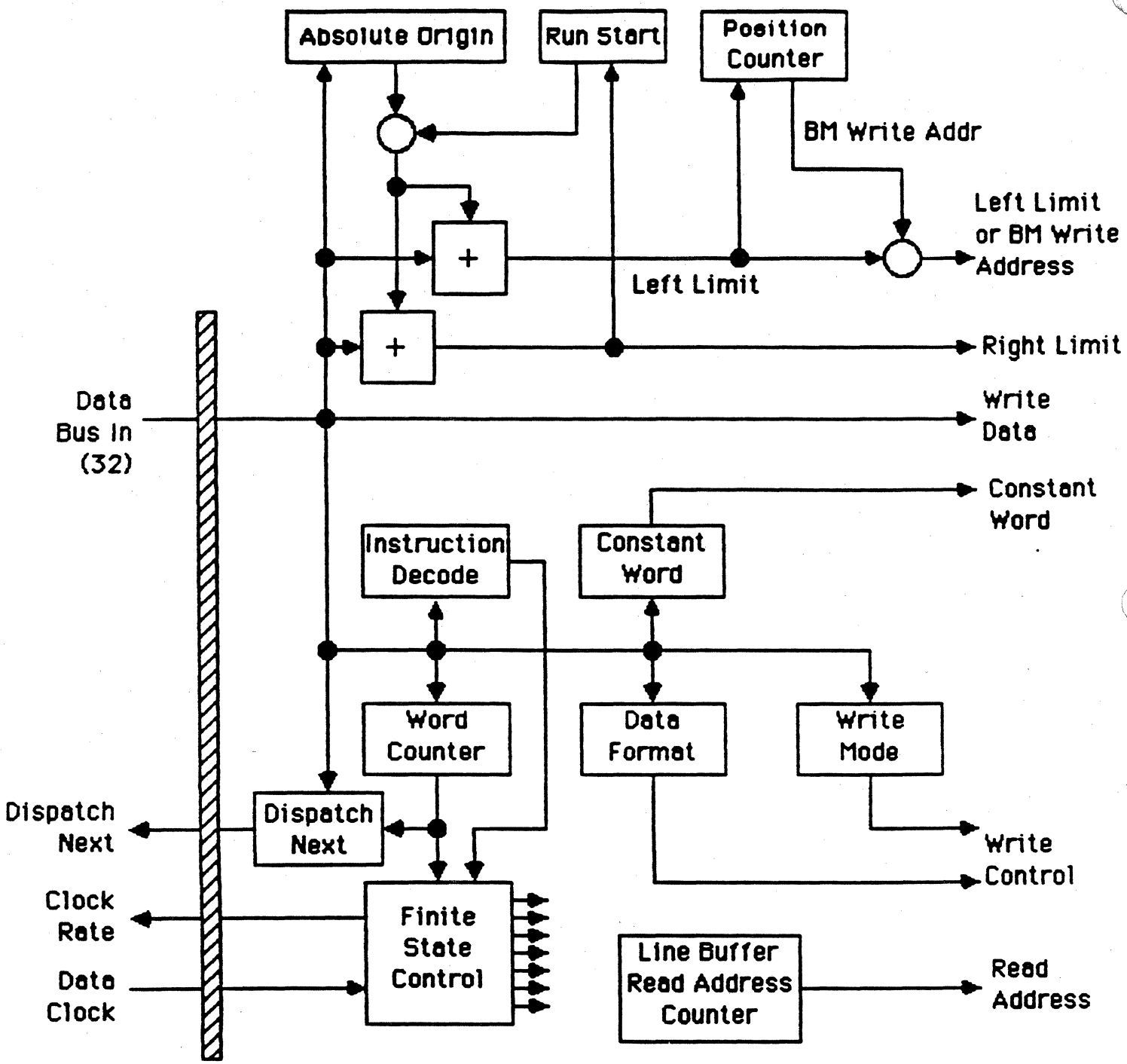


QuickScan Line Buffer

Block Diagram

SGP 2/18/85

Apple II Group Confidential and Private



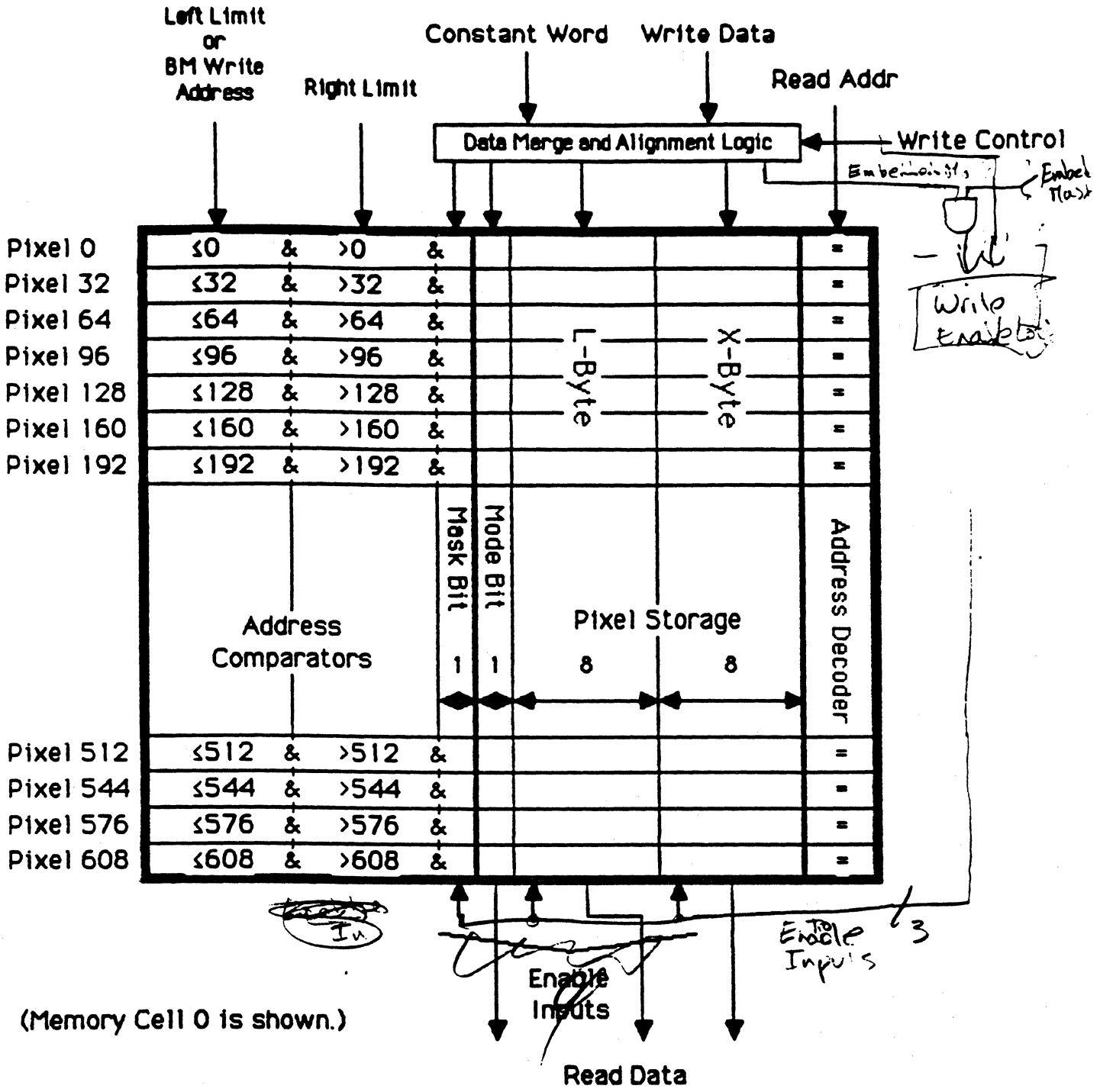
QuickScan Line Buffer Control

Block Diagram

○ = Multiple Source "OR" Junction

● = Single Source Junction

SGP 2/18/85



QuickScan Line Buffer Memory Cell

Block Diagram

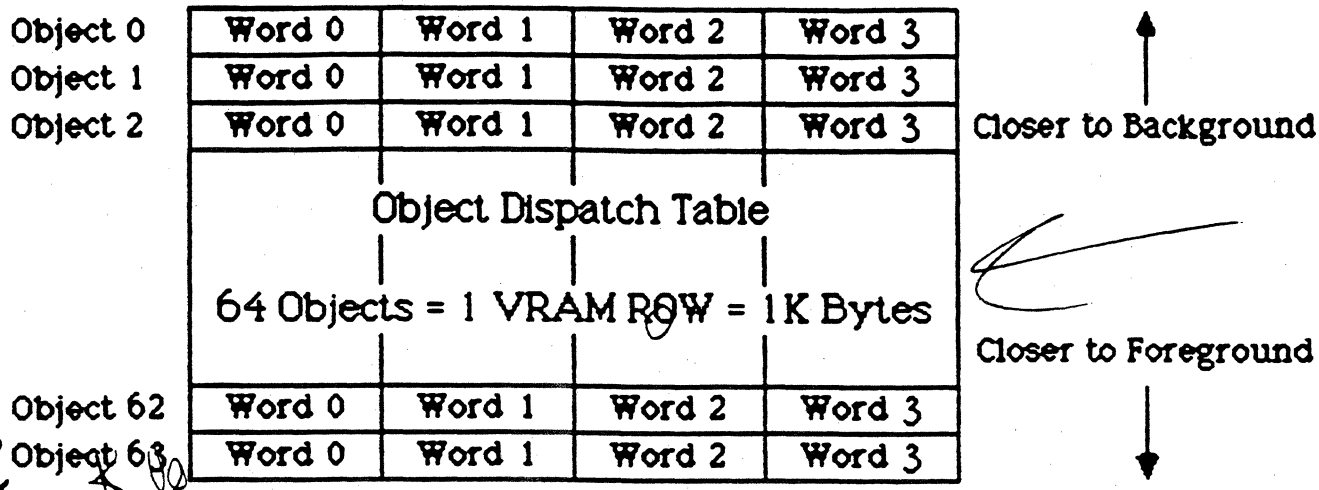
SGP 2/18/85

Apple II Group Confidential and Private

QuickScan Object Dispatch Table

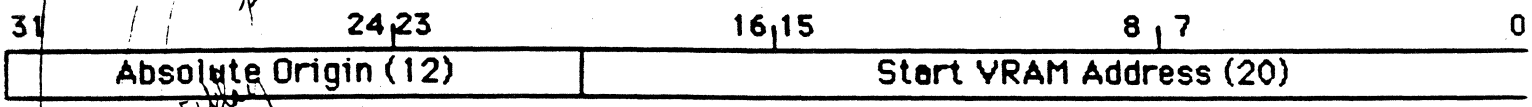
Dispatch Table Format

SGP 2/18/85

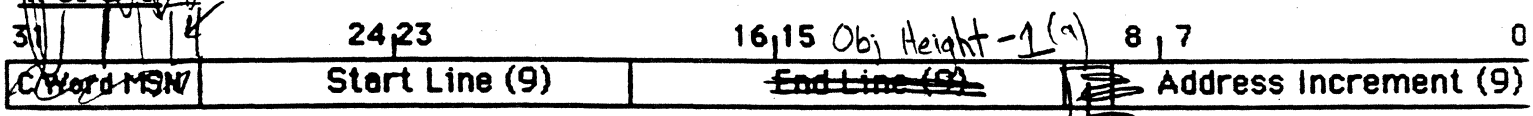


Dispatch Table Entry Format

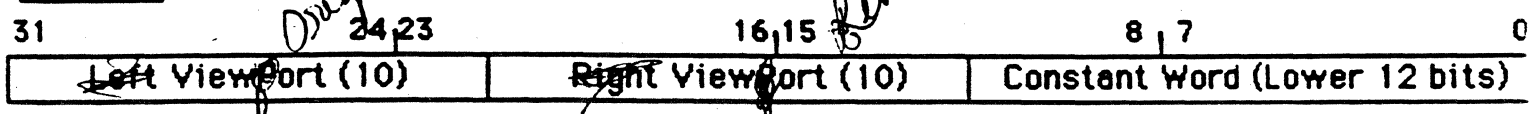
Word 0



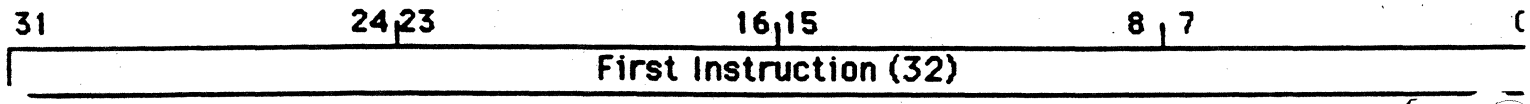
Word 1

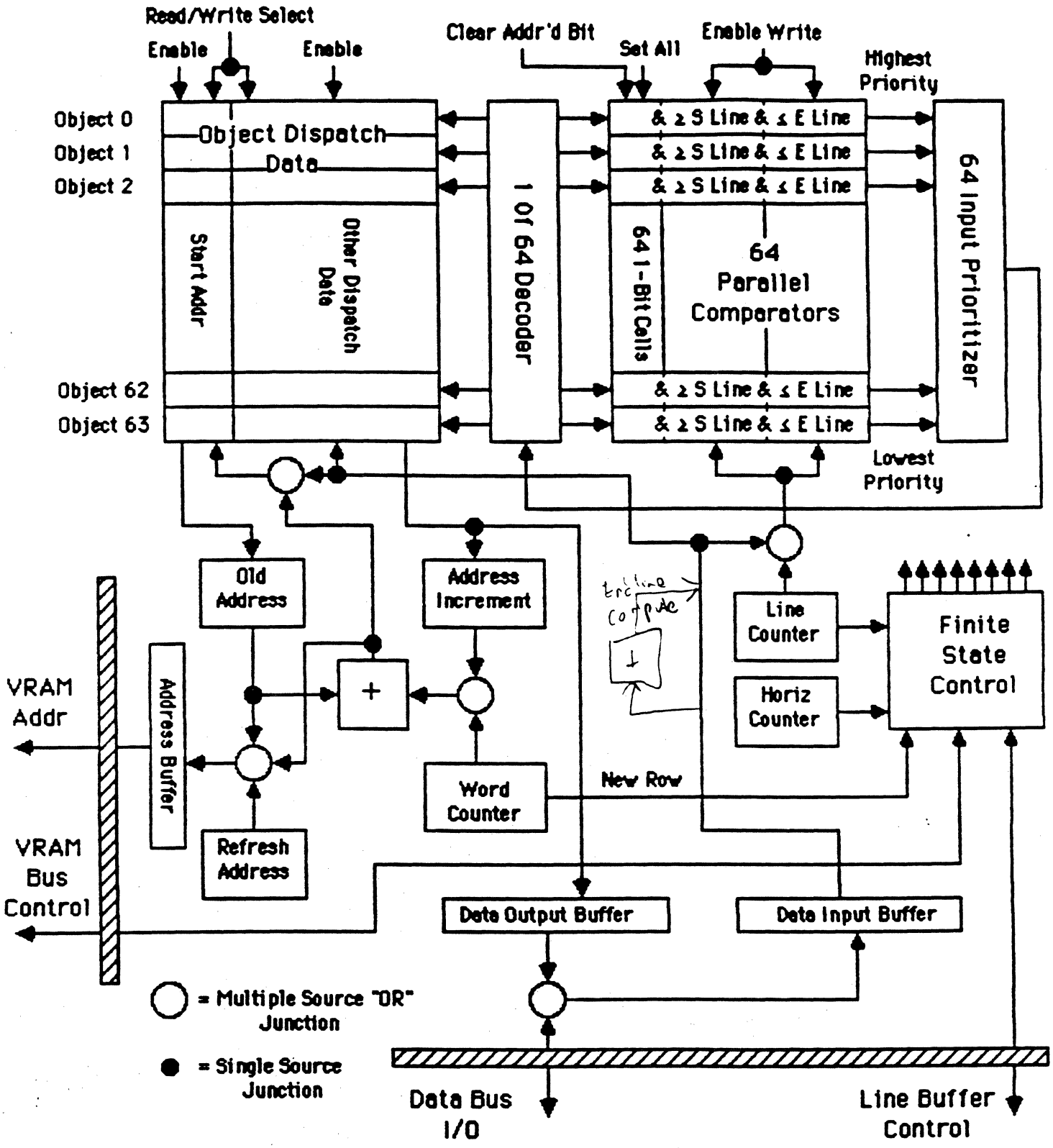


Word 2



Word 3





QuickScan Dispatcher

Block Diagram

SGP 2/18/85

Apple II Group Confidential and Private

QuickScan Dispatcher

Functional Description

SGP 2/20/85

Introduction

The QuickScan Dispatcher's primary function is to start up object descriptions object-by-object in a line and line-by-line in a frame. To accomplish this function, it must determine:

a) which object is the next one to load on the current line,
b) where in the Graphics memory that line of the object is stored,
and c) when it can access that memory and not interfere with the CPU.

This accomplished, it must access the data, send the appropriate initializing information to the Line Buffer, then commence loading the object description. Each startup operation like this is termed a "dispatch".

The Object Dispatch Table

The Dispatcher is configured at the end of each Vertical Blanking Interval. First it accesses a fixed address and loads in a few words of control information (e.g. 30Hz/60Hz mode select, external genlock select, etc.) as well as the row address of the Object Dispatch Table (the ODT). Then it accesses the ODT row (the ODT takes up exactly one row), and loads in the all 256 words. ~~At~~ this ~~point~~ the Dispatcher has all of the information it needs to dispatch all 64 objects in the ODT for the entire forthcoming frame.

Please refer to the Dispatcher Block Diagram and the ODT Dispatch Table Format diagram during the following discussions.

The data in an ODT ~~entry that is of~~ interest to the Dispatcher is the Start VRAM Address, the Address Increment, the Count Words Flag, the Start Line, and the End Line. The rest of the information is simply stored by the Dispatcher, and sent directly to the Line Buffer during a dispatch,virtually without evaluating its content whatsoever. The five operating data fields listed above are divided into two groups, the address information, and the line information.

Determining the Next Object to Dispatch

The line information group (Start Line and End Line) tell the Dispatcher those lines on which an object is displayed. These two values are stored in special memory cells which are actually ≥ and ≤ comparators respectively with the current line number fed in continuously (see Block Diagram). Thus, the Start Line value for every entry in the ODT is constantly tested to be ≥ the current line value, and the End Line value for every entry in the ODT is constantly tested to be ≤ the current line value. The AND (&) of these two tests is generated, and so, on the output of each ODT line information entry we effectively have a bit that says whether or not the object for that entry appears on the current line.

Before these logical values are carried out of the structure they are also each ANDed with a special 1 bit cell. These special 1 bit cells have unique access properties: a single common input sets all cells to logic 1 state, and another single common input causes any individual cell that is selected to go to a logic 0 state. The cells are selected by the same address decoder which selects the entries in the ODT (other than the line information). Thus, we have the capability to set all cells to logic 1, then clear the individual cell which corresponds to the currently addressed ODT entry.

As stated above, a bit cell is ANDed with the result of the line comparators for each entry. The resulting outputs feed into a 64 input prioritizer with entry 0 having the highest priority and entry 63 the lowest. The output of this prioritizer (a number between 0 and 63) is fed into the address decoder which selects the ODT entries and the 1 bit cells. So, the highest priority input running into the prioritizer determines the entry selected by the address decoder (e.g. if the prioritizer input from line comparators of entry 23 is the highest priority input, then ODT entry 23 will be selected by the address decoder).

The system works in the following way:

At the beginning of a line the Dispatcher state machine sets all of the special bit cells to logic one, and the new current line is fed into the line comparators. Since none of the bit cells affect the AND evaluations, the line comparators output their logical result to the prioritizer without

interference. Clearly, the prioritizer will output the entry number of the highest priority object which appears on the current line. This number will go to the address decoder, so the highest priority ODT entry which appears on the current line will be selected.

Well, that's convenient. It just so happens that this is the first object we want to dispatch! The ODT entry already selected, the Dispatcher state machine reads this data and deals with it accordingly (discussed below).

Then, the entry still selected, the state machine activates the control signal which clears the selected special bit cell to logic 0. Now, the line comparator entry which had just been selected by the prioritizer is turned off: the bit cell forces the AND result to logic 0. But, all of the other line comparators are still enabled, and the prioritizer outputs the entry number of the next highest priority object which appears on the current line.

Curiously, this is exactly the next object that we want to dispatch! The object is then dispatched at the appropriate time, its corresponding special bit cell is set to zero, and the next highest priority object which appears on the current line is selected. And, so on until all of the objects on the current line have been dispatched. At this point the Dispatcher state machine waits until the next line starts to begin the process again.

Handling Object Start Addresses

The address information group (the VRAM Start Address, the Address Increment, and the Count Words Flag) holds the information the Dispatcher needs to determine the start address of each object on each line.

The Dispatcher works from the paradigm that the Start Address currently stored in the ODT for a given object holds the address that should be accessed when the object is next dispatched. This works fine for the first dispatch of a frame; the ODT Start Address still holds the value loaded in during Vertical Blanking which points to the first line of the object description. The problem is, how can we make sure that the Start Address holds the correct address for the second and subsequent lines of the object description when those lines of the object are dispatched? Well, there are two ways, depending on the nature of the object.

The first way is opted if the Count Words Flag in the ODT entry is 0 (negatively asserted). This means that the address increment from one line's start to the next line's start in the object description is a fixed amount. This amount is contained in the Address Increment. When the object is dispatched, the Start Address goes into the Old Address register (see Block Diagram), and the Address Increment goes into a register of the same name. These 2 values are summed, and while the ODT entry is still selected (prior to clearing the special bit), the sum is written back to the Start Address field, replacing the old Start Address. Thus, when the next line comes along and the object is dispatched again, the Start Address field will point to the proper address for the next line's data.

The second way is opted if the Count Words Flag in the ODT entry is 1 (positively asserted). This means that the address increment from one line's start to the next line's start is a variable amount (often the case in run-length descriptions). As in the first way the ODT Start Address is put in the Old Address register. But in this case the Address Increment is ignored, and a counter clocked by the Shift Register clock (the Word Counter) is cleared to zero, and as the data load for the object description is carried out, it keeps track of how many words are actually loaded into the Line Buffer. When the Line Buffer signals that it wants the Dispatcher to dispatch the next object, the word count is summed with the Old Address, and the result is placed in the Start Address field in the ODT. (Then, the special bit is cleared, and the next object is dispatched.) Thus, when the next line comes around, and the object is dispatched again, the Start Address in the ODT will be exactly the address following the last address loaded into the Line Buffer, regardless of what the increment was from the address at the beginning of the previous line.

Now that we have gone through each way independently, we have the perpective to see that both ways are identical except in state machine execution except for one register transfer. It works like this: Upon object dispatch the Old Address Register is loaded with the Start Address, the Address Increment Register is loaded with the Address Increment, and the Word Counter is cleared to zero. Then, nothing happens (except for the Word Counter counting) until the Line Buffer sends a Dispatch Next signal to indicate the object description's completion. Then, either the Address Increment register or the Word Counter is selected to sum with the Old

Address depending on the Count Words Flag, and in a single cycle both the sum (the new Start Address) is written and the special bit is cleared. The very next cycle the dispatch data for the next object is read from the ODT, and the next object is dispatched.

There is one unresolved issue in this address increment process: when the Dispatch Next Flag is received from the Line Buffer, there may be 0,1,2,3,4 or 5 words following, depending on the particular instruction that the object description ends on. One approach would be to wait until the object description ends before updating the ODT, but this is problematic because we really need the time to complete the sum, update the Start Address, and let the Prioritizer and Decoder settle in their new state. Another approach would be to have the Line Buffer tell the Dispatcher how many words are left, but this means pins. Another way is to have the instructions partially decoded in the Dispatcher so it knows what is going on and can figure out the number of words. But, I think the simplest approach is to put the burden on the 68020 and require that all variable length object descriptions end lines in such a way that there are 5 words after the word at which Line Buffer will send Dispatch Next until the first word of the next line in the object description. This will waste a little RAM if the object descriptions are not planned well, but presumably the variable length objects are pretty compact anyway.

VRAM Bus Arbitration and Refresh

VRAM Bus Arbitration and VRAM Refresh Generation are each discussed in separate individual documents. It would be redundant to discuss them here.

Dispatching an Object

After what it takes to get up to dispatching an object this part is very simple. Essentially, an object is dispatched by sending four normal instructions to the Line Buffer that happen to prepare it for the forthcoming object load. The Line Buffer actually does not "know" that these instructions are not part of an object description, and indeed, it contains no special logic to support the dispatch process.

The four instructions that dispatch an object are as follows:

1. CSwitch Absolute Origin,Constant Word
2. LRun 0,M,0,0,0,641

3. LRun 1,M,0,0,Left ViewPort, Right ViewPort
4. First Instruction

Immediately following the First Instruction, the first word from VRAM will load in, and the object description will continue loading until a Dispatch Next bit is set in an instruction.

The explanation of the four instructions goes as follows: Instruction 1 defines the horizontal reference point and the default input data, the Absolute Origin and the Constant Word, respectively. Instruction 2 clears the Mask bit in every pixel cell in the Line Buffer, then Instruction 3 sets the Mask bit in those cells between the Left ViewPort and the Right ViewPort. This has the net effect of allowing writes to only those cells within the ViewPort. Then, finally, the First Instruction is just that, the first instruction of the object description. It can be anything.

There is an exception to this dispatch process worth mentioning. If the object description requires a ViewPort more complex than the simple one provided by this mechanism, the user can set up her own ViewPort in a higher priority object description, then disable the automatic ViewPort mechanism from clobbering the one she just set up. This is accomplished by setting the Right ViewPort value to -1. The Dispatcher, upon detecting this value will send NOp's instead of LRun's for instructions 2 and 3.

Handling Row Crossing Conditions

The Video RAMs specified for the QuickScan system are set up in such a way that data is only rapidly accessible if it happens to be sequential and all in one row. If a line of an object description is entirely contained in one row, then managing the VRAM is no more complex than as it is already described here. If, however, an object description of a line does cross a row boundary, then a) a performance penalty will be applied, and b) the Dispatcher will have to load in the next row.

If you are familiar with the NEC VRAM devices, you will know that if you anticipate crossing over the end of a row, you can start a Transfer to Shift Register cycle early and seamlessly switch from the end of one row to the beginning of the next. With QuickScan this can't quite be achieved. First of all, the Shift Register is often being pushed to its maximum speed; a seamless switch at that clock rate is virtually impossible. And, second of all, the Dispatcher can't always anticipate that an object description is going to cross a row boundary; it doesn't know the extent of

a variable increment object description until the last instruction is loaded.

As a straightforward solution to this problem I recommend that the Dispatcher monitor the position in the row through some function connected with its Word Counter. If the row boundary is actually crossed, the Dispatcher will only then initiate the Transfer to the shift register. Considering worst cases for DMA latency, we have to allow 560ns to get the VRAM "back on line" with the next row's data. I've considered at least a half dozen approaches to make this row transition less painful, but this is by far the simplest, neatest, and most consistent in timing. It also is nice because it follows the same timing chain that is used when the Dispatch Next Flag is detected. I recommend that we just warn off programmers from crossing row boundaries, and let them know it'll cost them 560ns every time they do.

Memorandum

To: Jonathan Architecture Committee, et al
From: Steve Perlman
Date: 3/8/85
Subject: QuickScan Programming Manual

Attached is a copy of the QuickScan Programming Manual. This document throughly describes the details of programming QuickScan without going into any hardware implementation issues. Except for a few minor details the functional specification of QuickScan is complete in this document, and the system is ready for critical evaluation.

Although the functional specification is complete, I haven't quite finished the Applications Chapter, but believe me, there's plenty here to go through! Moreover, what is here really covers the core of QuickScan applications, and I wanted to get these ideas into people's thinking as soon as possible. I'll be getting the last few pages covering the esoteric stuff out as soon as I can.

S-6 (Supplement)

QuickScan Display Subsystem

Programming Manual

SGP 3/5/85

Revision History

First Draft - Missing
some Applications and Appendix B 3/8/85 Steve Perlman

Abstract

Although a detailed hardware description of QuickScan is the best way to establish its feasibility, a detailed software description of the subsystem's operation is the best way to establish its usefulness. The first Strawman release of QuickScan had an introduction that went into some of the salient features of the system and gave a few examples of how to program it, but the bulk of the document package focussed on the implementation details. This document is a "Not for Programmers Only" description of QuickScan's operation and software model. An extensive Applications chapter shows practical implementations with thorough discussions of CPU overhead, QuickScan loading, and RAM utilization, but from a programmer's point of view. This document is intended to serve not only as a programming guide, but, especially at this early stage, as a means of evaluation.

S-6 (#3)

Memorandum

To: Jonathan Architecture Committee, et al
From: Steve Perlman
Date: 3/18/85
Subject: QuickScan Programming Manual Supplement

Attached is section 7.2 of the QuickScan Programming Manual as well as an updated Table of Contents. Please replace your Table of Contents page ii with the updated one, and then insert the text pages after page 67. Some of the people who received double-sided copies of the Programming Manual are missing pages 29, 30, and 31. I've included these pages at the end of this packet for those of you who fall in this category.

This section covers applications of QuickScan's fully parallel run generation mechanism including real-time cartoons, backgrounds, and real-time 3-D solid polygon modeling. The capabilities of this mechanism, more than any other particular feature, distinguish QuickScan from any other display subsystem that exists commercially, at any price. If you're interested in graphics, please take a moment to look through it.

Contents

* Forthcoming † Included, but not numbered

1. Introduction

This document is a detailed description of how one goes about programming QuickScan. It pretty much goes through the translation of QuickScan's hardware functionality into software capability. I've avoided as much as possible the discussing of actual issues in the silicon, addressing any such constraints rather as fixed limitations of the architecture. Thus, this document is a "how-to" guide to QuickScan. For a "why-it-works" guide refer to the additional documentation.

Although this document can be used as a reference and thumbed-through in any order, I recommend that you read it at first starting from the beginning and working your way to the end. I have been careful not to use terms and concepts before they are defined, and if you don't skip any sections, then you should be able to understand each new section as it is discussed.

To keep us in a 68020 frame of mind, I refer to a 32 bit long word when I say "word" unless I qualify it as 16 bits. Also, when I qualify a statement with the phrase, "in general," then I mean that the statement holds true for uses by normal people, but beware: there are hooks for hacks to mess around with things so that the statement might not be true.

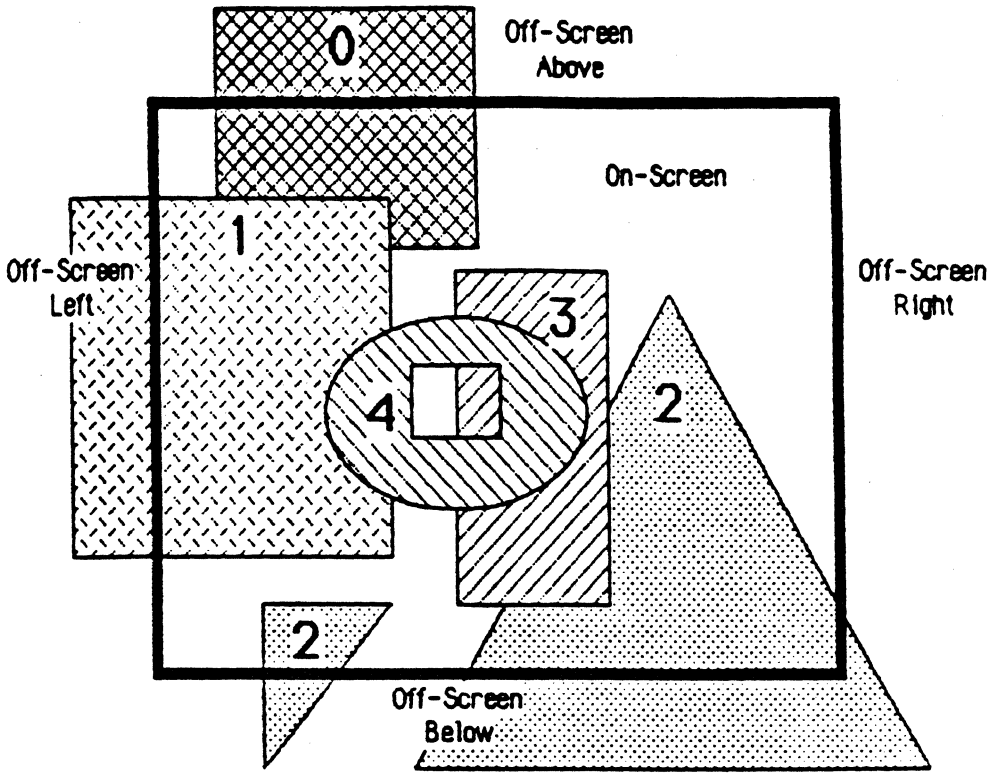
Information contained in this document supersedes information contained in the documentation packet released 2/21/85. An updated hardware specification is forthcoming.

SGP 8 March 1985

2. Objects

2.1 Object Descriptions

The following figure shows an example of a QuickScan display:



QuickScan Display Example 1

All QuickScan displays are made up a collection of *objects*. Each individual object in Example 1 is identified by a pattern. Note that objects can be of any shape and size and need not even be contiguous. Objects may be entirely visible on the display screen (objects 3 and 4 above), they may be partially visible on the display screen (objects 0, 1, and 2 above), or they may be entirely off the display screen. Any part of an object which is Off-Screen is automatically cropped by QuickScan.

We assign to each object a *priority level*. The **priority level** tells

QuickScan which object to put in front of another when 2 or more objects overlap. **Priority levels** are from 0 to 63: the higher the **priority level**, the closer to foreground QuickScan will place the object. The number on each of the objects in the above diagram indicates its respective **priority level**. There may be only one object to a **priority level**. (But, in advanced applications, there may be more than one **priority level** to an object.) **Priority levels** also serve as identification for objects in the text of this document. Thus, Object 2 refers to the object at **priority level** 2.

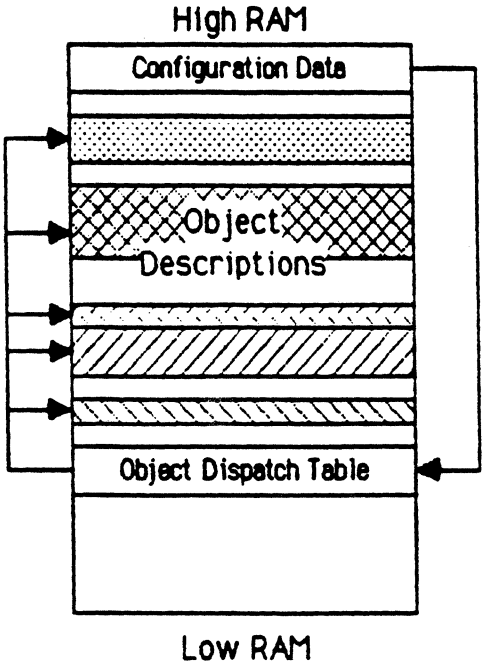
All objects are made up of a contiguous sequence of words of arbitrary length plus 4 words of control data. The former data is called the *object description*, and the latter is called the *dispatch table entry*. An **object description** can be placed anywhere in RAM (sorry, not in ROM), although there are some areas in RAM which are best avoided to optimize performance. And, if it serves some hacker's end, **object descriptions** can even overlap.

The **dispatch table entries** for all of the objects to appear in a given video frame are collected in the *Object Dispatch Table* (the ODT). Up to 64 **entries** may be sequenced, one after another for each frame. Each **entry** identifies a particular object, and the order of the **entries** indicates the **priority levels** assigned to the objects. The first **entry** in the ODT is **priority level** 0, the second is **priority** 1, and so on, until the last **entry** is **priority level** 63. The 4 words of a **dispatch table entry** contain the attribute information for an object as well as a pointer to the beginning of the **object description**. (Note that the same **object description** can be referred to by more than one **dispatch table entry** if multiple copies of the same object is desired.) The ODT may begin in RAM at any address that is a multiple of 1024, and it need extend only so far as there are **dispatch table entries**. Note that there is only one ODT for each frame displayed by QuickScan.

The *configuration data* is a contiguous sequence of words at a fixed address in RAM which contains fundamental control information for the QuickScan chip set. Most of the contents of this data are not very relevant at this point in our discussions, but it is important to note that a)

this is the only data at a fixed address in RAM used by QuickScan, and b) this data contains the pointer to the Object Dispatch Table.

This could be a memory map of what we've discussed thus far:



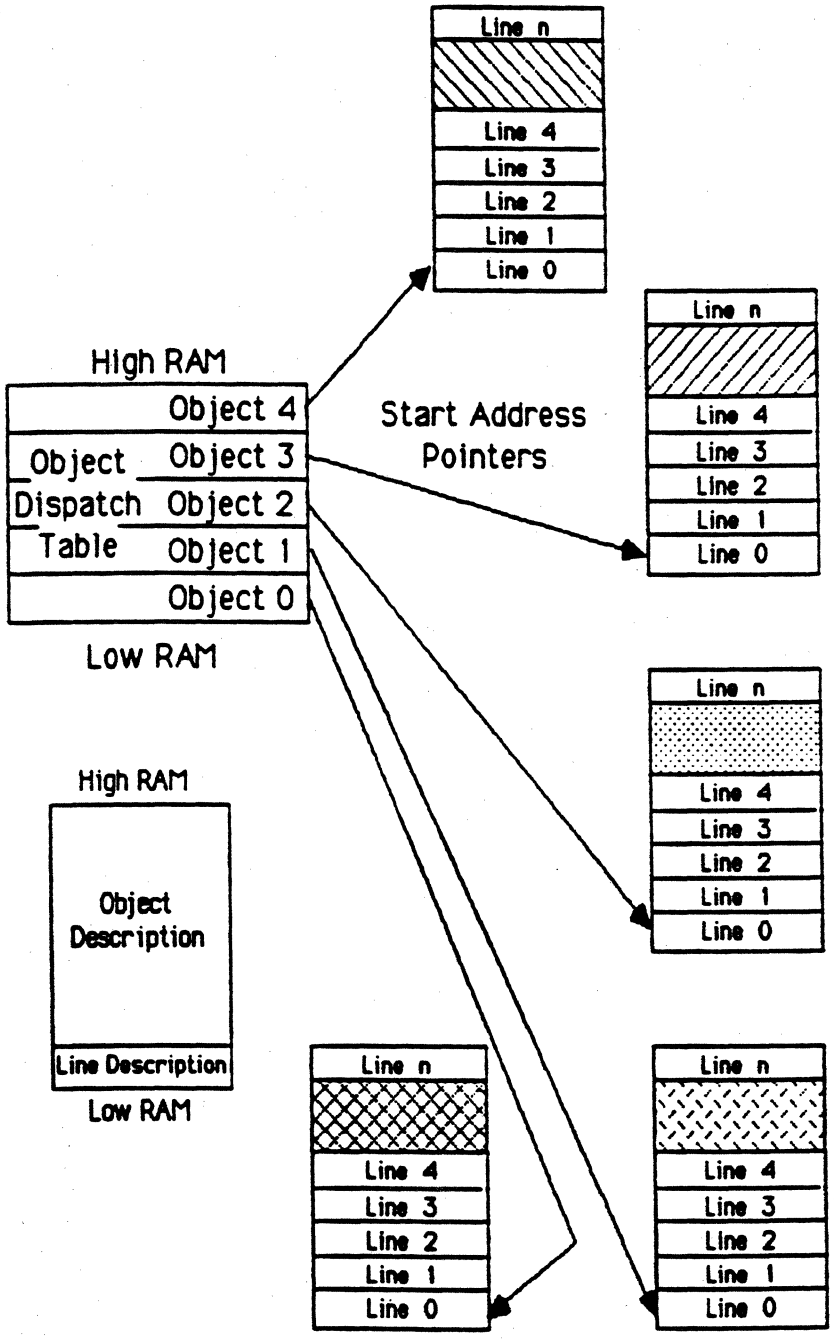
Memory Map for Example 1

2.2 Line Descriptions

When QuickScan displays objects, it processes them each line-by-line. (I use the word "line" here (and throughout this document) to refer specifically to a 1 pixel tall horizontal row of pixels extending from the far left side of an object to the far right side. Note that each line of an object is coincident with a line on the monitor or TV screen when the object is On-Screen.) More specifically, QuickScan processes each object from its top line to its bottom line.

Looking closer at the **object descriptions**, we find that they are each a sequence of independent *line descriptions* to accommodate the nature of this line-by-line processing. The first data in an **object description** is the **line description** for the top line (line 0), it is

directly followed (generally) by the data for the next line (line 1), and so on, until the very last data is the **line description** for the bottom line (line n). We end up with an **object description** memory map that looks like this:

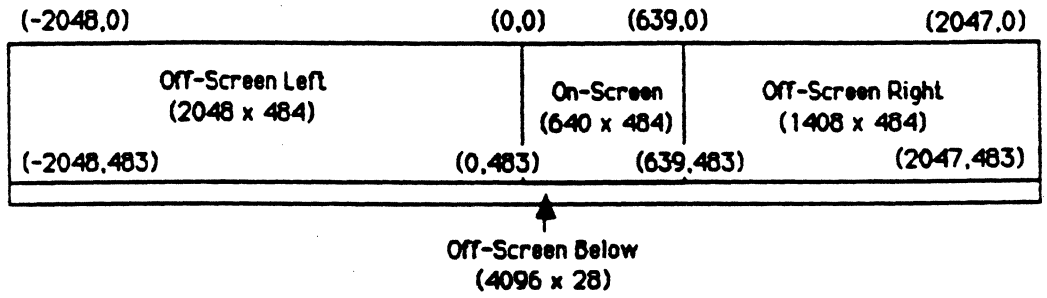


Each **line description** of an **object description** is independent of the other **line descriptions** in that **object description**; what is happening on one line of an object has no effect on what is happening on any other line of the object. Indeed, it is quite correct to say that QuickScan's fundamental independent display entity is an object line, and that an object is simply an ordered collection of lines. Remember this concept - the QuickScan architecture revolves around it.

An object's **line descriptions** may either be all of the same length or all of variable length, and in both cases the chosen lengths are arbitrary. **Fixed** or **variable length mode** is specified in the **dispatch table entry** for each object, and if the lines are of fixed length, then the **line length** is specified in the **entry** as well.

2.3 QuickScan Display Space

Consider the following figure:



QuickScan Display Space

The above figure diagrams the display space managed by QuickScan. The area labeled "On-Screen" identifies the region of the display space which actually appears on the monitor display screen (a centered subset of this area, about 512x210, is visible on a television screen). The Off-Screen regions, although processed internally exactly like the On-Screen regions, do not result in any visible display. Any **object descriptions** which begin within the defined QuickScan display space, yet extend outside of this space will be truncated at the display space limits.

They will not "wrap around" to the other side.

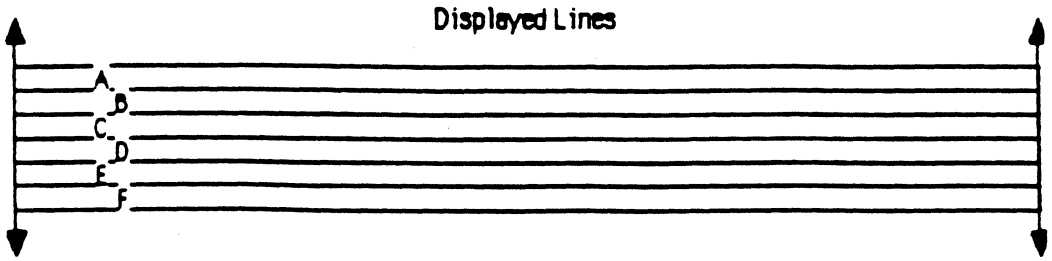
Since QuickScan **object descriptions** always progress to the right and downward, by far the most important Off-Screen region is Off-Screen Left. It allows an **object description** to begin far to the left of On-Screen and extend into the visible display space. This capability is fundamental for panning large backgrounds and for moving objects gradually in from the left of the screen.

It is also vital to be able to move in objects from above the screen, but this can be accomplished by the 68020 finding the address of the first **line description** which is On-Screen, and replacing the **start address** (of the **object description**) in the **dispatch table entry** with this value. It is a simple problem for the 68020 to crop the top lines off of an object in this way, but the same operation is a difficult problem for QuickScan. Conversely, cropping the left pixels off of an object is a trivial problem for QuickScan, yet a potentially monstrous problem for the 68020 (as you shall see). Hence, we have QuickScan manage Off-Screen Left and the 68020 manage Off-Screen Above.

Notice also that Off-Screen Right and Off-Screen Below are really not very useful regions of the display space; their inclusion in the QuickScan display space is more or less vestigial. Although it is valid to specify an **object description** which starts in one of these regions, an object so described will not result in any visible display. These regions exist because a) we get them for free, and b) they might simplify the coding of objects which are frequently moved On- and Off-Screen.

2.4 The Line Buffer

As noted above, QuickScan processes **object descriptions** line-by-line. To be more precise, QuickScan processes a given **line description** while the line directly above it is being displayed. That is to say, QuickScan's line processing is always one line ahead; it has one line's time to prepare a line before it is displayed. This function is known as *single line buffering*, and can be seen pictorially in the following diagram:

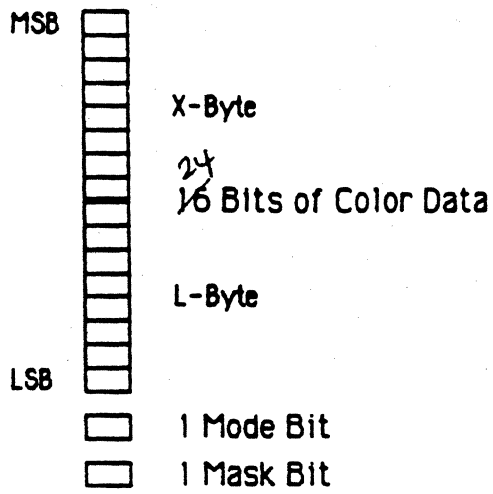


While this line is being prepared	This line is being displayed
B	A
C	B
D	C
E	D
F	E

The Concept of Single-Line Buffering

In order to accomplish **single line buffering**, we need a temporary place to store the line being prepared, holding it until the next line time when it will be displayed. This temporary storage area is called the *line buffer*, and it is in this subsystem that all QuickScan video is generated.

The **line buffer** is 640 pixels long, maintaining 1 *pixel storage cell* for each pixel in a horizontal line across the On-Screen region. Each pixel contains 18 bits, arranged in the following manner:



A Pixel Storage Cell in the Line Buffer (1 of 640)

The 16 Bits of **color data** hold the information that, in one of two ways, represents the particular color for that pixel. The **mode bit** indicates which of these two representations shall be used for that pixel. And, finally, the **mask bit** controls whether the **color data** and **mode bit** can be overwritten or not.

Considering the **mask bit** in detail we have:

[1] = Writes to this Pixel Accepted

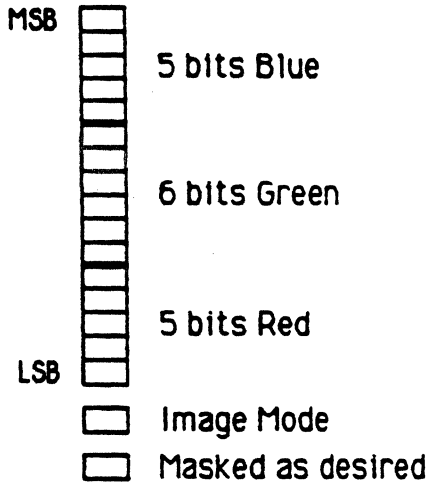
[0] = Writes to this Pixel Ignored

The Mask Bit

It operates exactly as stated: If an attempt is made to write to the **color data** (and consequently the **mode bit**) and the **mask bit**'s value is 1, then the existing **color data** and **mode bit** shall be replaced with the data being written. If the same write is attempted and the **mode bit's** value is 0, then the **color data** and **mode bit** shall remain as they are. As

we shall soon see, the **mask bit** is vitally important in the display of bit-maps and complexly-shaped objects.

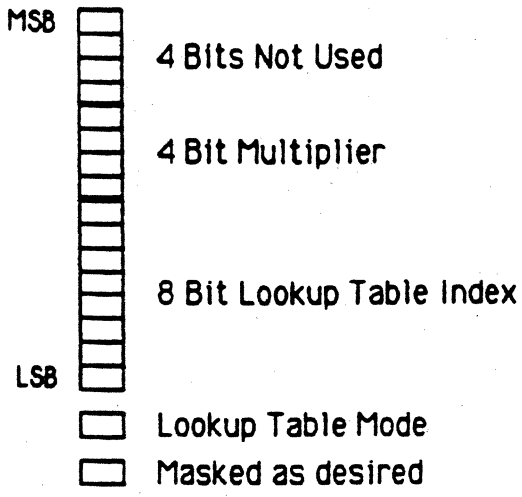
There are two modes in which a pixel's color may be represented by the 16 bits of **color data**. The first is *image mode*. Here the **color data** is divided into 3 fields: 5 Bits for R, 6 Bits for G, and 5 Bits for B, as shown below:



A Line Buffer Pixel in Image Mode

The RGB value contained in the **color data** represents exactly the color to be displayed on the monitor at this pixel; it is a direct mapping. The **mode bit** is automatically set to **image mode** when the 16 bits of **color data** are written in this mode, and the **mask bit** may be set to what ever value is needed.

The second mode of representation is *lookup table mode*. Here only the lower 12 bits of the **color data** are used, the lower 8 bits representing an *index* to a 256-color Lookup Table, and the upper 4 bits representing a *multiplier* value to apply to the color selected by the **index**:



A Pixel in Lookup Table Mode

The Lookup Table holds 256 colors represented as 4 bits R, 4 bits G, and 4 bits B, and it is loaded from a table in RAM prior to the start of each video frame. The 4 bit **multiplier** is applied independently to each R, G, and B of the table entry selected by the **index**, multiplying each by a value between 0 and 15. This has the effect of accordingly brightening or darkening the nominal color, an effect very useful in 3-D shading models, anti-aliasing, and interlace de-flickering.

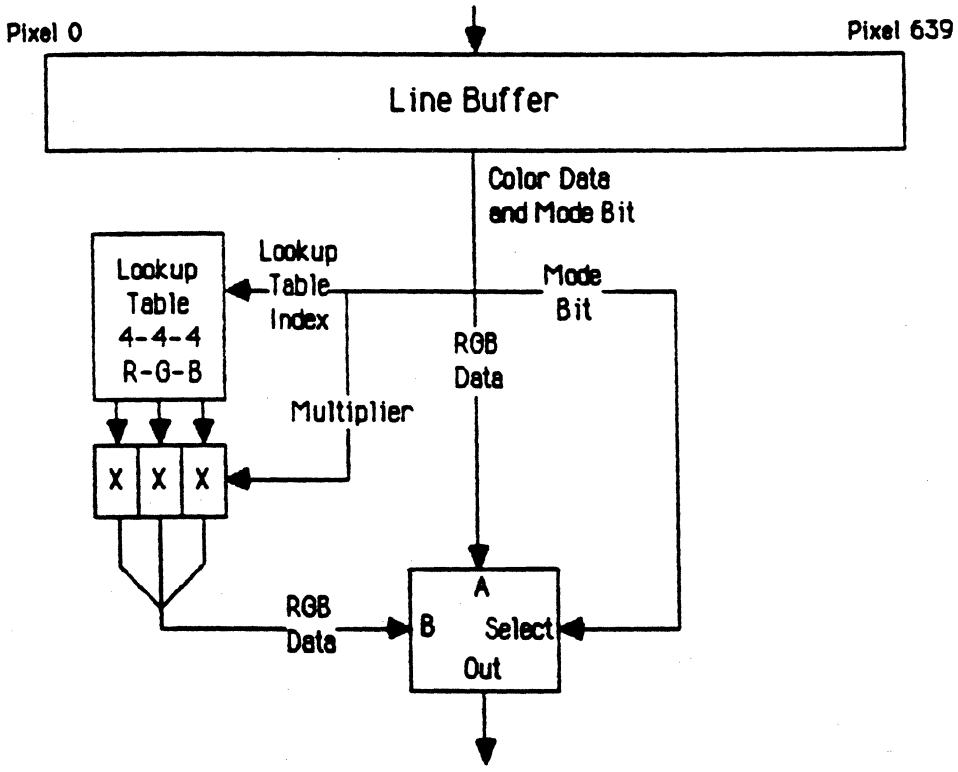
Unlike in **image mode**, the **color data** in **lookup table mode** represents colors indirectly: first by selecting a nominal color with the **index**, and second by altering that nominal color with the **multiplier**. The mode bit is automatically set to **lookup table mode** when the **color data** is written in this mode, and the **mask bit** can be set as needed. The upper 4 bits of the **color data** are not used, but should be set to zeros.

Now, as we noted before, QuickScan objects are processed line-by-line, from top to bottom. What is perhaps not obvious from this, however, is that the processing of all of the objects is interleaved, such that each object which appears on a given line loads its **line description** for that line into the **line buffer** before the line is displayed. While this is occurring, the line just above this line is being displayed. Furthermore, the processing of the objects' **line descriptions** is done in the order that

the **dispatch table entries** appear in the ODT. This serves to prioritize the objects just as we expect by overwriting those objects of lower priority where there is an overlap.

To clarify the previous paragraph, flip back to page 3 and the QuickScan Display Example 1. Consider for a moment the line dead center in the On-Screen region. Objects 1, 2, 3, and 4 all appear on this line, and we expect them to be prioritized as shown in the picture. How would this work? Well, first a **line description** of object 1 is loaded into the **line buffer**, then one for objects 2 and 3. Then, the **line description** object 4 is loaded, and it overwrites some pixels which were written into the **line buffer** by objects 1 and 3 at pixels of overlap, the prioritization desired. A conceptual diagram follows:

Line Descriptions for all Objects on Line



Line of Video to Monitor or TV

Line Buffer Conceptual Block Diagram

As shown above, a line of 640 pixels is prepared by the **line descriptions** for all of the objects appearing on that line. Then, the preparation complete, the **color data** is output as a line of video. The **color data** can either follow a direct path to the video output if it is in **image mode**, or it follows an indirect path, through the Lookup Table and the Multipliers, if it is in **lookup table mode**. Note that a line can switch between **image** and **lookup table modes** at any pixel; there are no restrictions in this regard. So, **image mode** objects and **lookup table mode** objects can be intermixed on a line as is desired.

3. Pixels

3.1. Pixel Data Write Formats

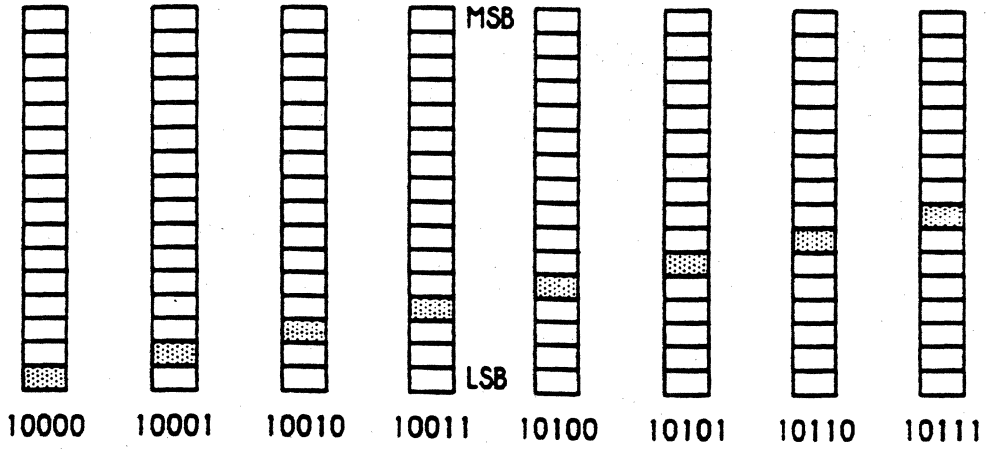
When we write pixel data to the **line buffer** from a **line description**, we generally don't specify all 16 bits of data to write. We can, of course, specify 16 bits for each pixel if we want, but the amount of RAM and CPU overhead necessary to support such **line descriptions** is extraordinary, and as a result we avoid such large **line descriptions** whenever possible.

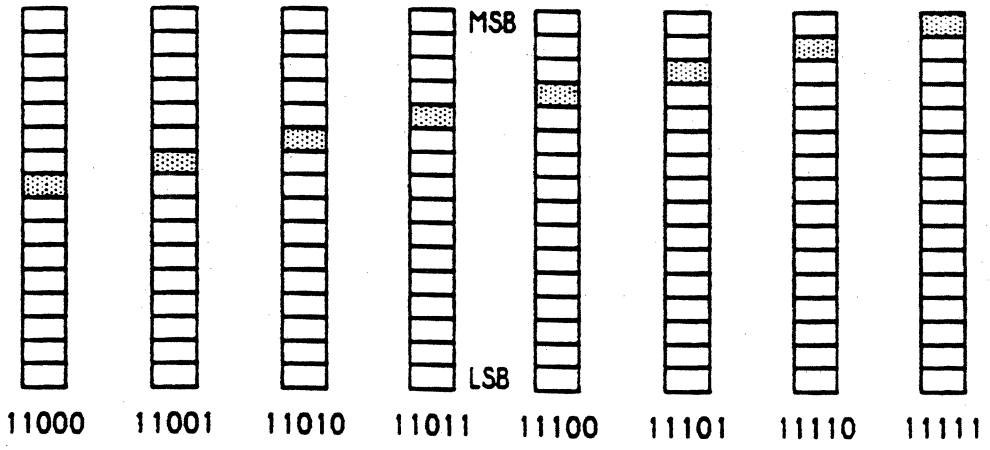
So, if we specify fewer than 16 bits for each pixel, how can we control what values are placed in the bits of the **color data** that we don't specify? The function is accomplished with the *constant word*, a 16 bit word which provides any bits in the **color data** which are not provided by the pixel data in the **line description**. For example, if our **line description** provides 1 bit of data for each pixel, then the other 15 bits of each pixel's **color data** will come from the **constant word**. Note that all pixels written by the **line description** will receive the same **constant word** bits while they each receive different pixel data bits (although the **constant word** can be changed in the middle of a **line description** if you wish).

The only ambiguity remaining in this scheme is determining which bits of the **color data** shall be specified by the **line description** pixel data and which bits shall be specified by the **constant word**. This is resolved by a parameter in the **line description** which specifies the *data format* of the forthcoming data. The **data format** ~~first~~ specifies the pixel data *width* (1, 2, 4, 8, or 16 Bits/ Pixel), ~~and then specifies the~~ *alignment* of the pixel data bits within the **color data** bits. (There is also a special **data format** with a **width** of 7 bits/pixel which is used in a particular circumstance and is coded specially.) Notice that the **alignment** affects only the **line description** pixel data. The **constant word** is always aligned at bit 0 in the **color data**.

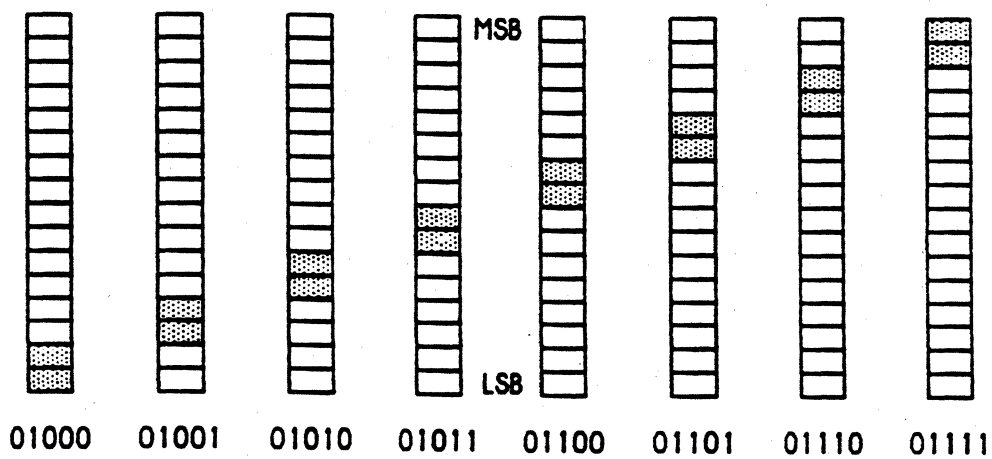
The following diagrams show the various **alignment** permutations available for each of the pixel data **widths**. In each 16 bit **color data** word shown, the bits written from the **line description** pixel data are shaded, and the bits written from the **constant word** are left white. The **data format** code is listed below each 16 bit **color data** word. (In the 7 bits/pixel **width** the code shown is actually a special *data alignment*

code, to be explained in the Instruction Set section.)

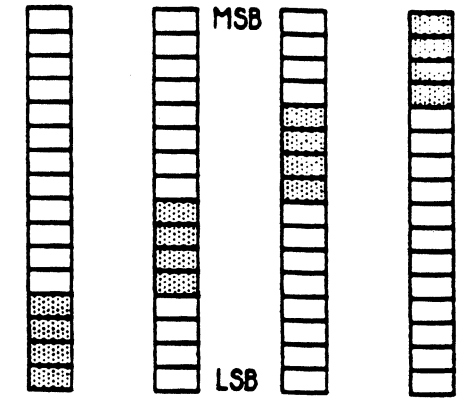




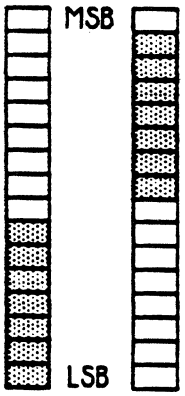
1 Bit/Pixel Data Formats



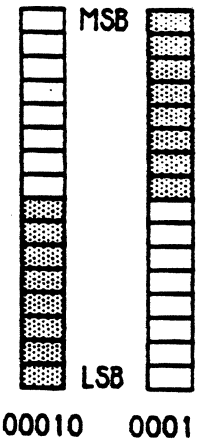
2 Bits/Pixel Data Formats



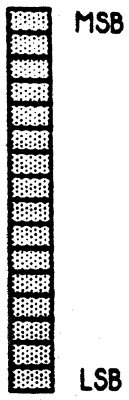
4 Bits/Pixel Data Formats



7 Bits/Pixel Data Formats



8 Bits/Pixel Data Formats



16 Bits/Pixel Data Format

Note that at 16 bits/pixel data **width** the **constant word** is not used as all 16 bits of **color data** are provided by the **line description** pixel data.

3.2. Pixel Write Modes

Since the upper and lower bytes of the **color data** word have different meanings in **lookup table mode**, sometimes it is desirable to write to one byte, but not to the other. Also, since the **mask bit** needs to be set up by the **line description** before it is used, it is necessary to have some way to access it. These pixel cell access paths are called *write modes* and are selected in the **line description** by a 2 bit **write mode** parameter. The encoding of the bits is as follows:

Mode	Encoding	Pixel Sections Written
M	0 0	Mask Bit Only
L	0 1	L-Byte (Color Data L.S. Byte) and Mode Bit Only
X	1 0	X-Byte (Color Data M.S. Byte) and Mode Bit Only
LX	1 1	L- and X-Bytes (Color Data Word) and Mode Bit Only

The previous section defined how to map the **line description** data to the **color data** word, but so far we haven't discussed how to map **line description** data to the **mode** and **mask bits**. The **display mode** of a given **object description** is specified in its **dispatch table entry** in the ODT. Whenever **write modes** L, X, or LX are specified and data is written to a pixel (i.e. the pixel is not masked), then the pixel's **mode bit**

is automatically set to the object's **display mode**. (If you're a real hack, there is even a way to change an object's **display mode** inside a **line description**).

Whenever **write mode** M is selected, the least significant bit of the **line description** pixel data is written to the **mask bit** unless the write is masked by an **embedded mask** (to be explained in the next section). Note that the prior state of the **mask bit** has no effect on this operation; the **mask bit** cannot mask itself. However, it can be masked by an **embedded mask**, and in such a case the write would not occur. Note also that the **data alignment** and the **constant word** are irrelevant to this write operation, and further, all bits of the **line description** data except for the LSB (and possibly the **embedded mask bit**) are ignored as well.

3.3. Embedded Masks

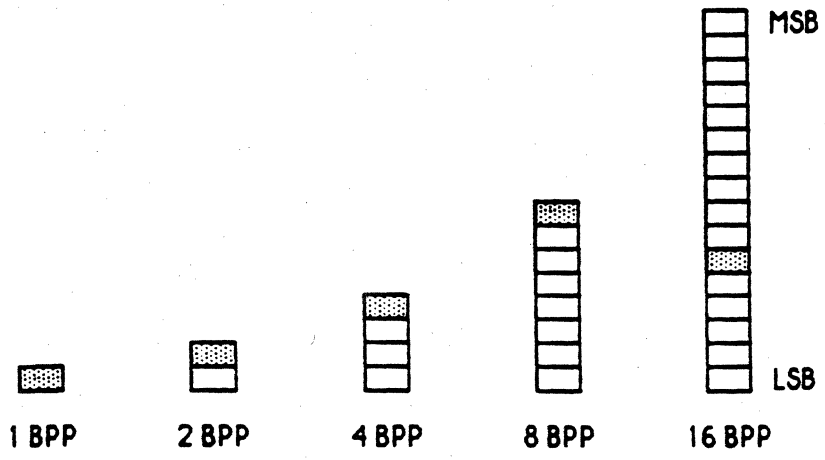
In addition to providing a write masking facility with the **mask bit** in the **pixel storage cell**, QuickScan provides a masking facility within the **line description** pixel data. These masks are formed by *embedded mask bits*.

If the **embed mask mode** is activated in the **dispatch table entry** or the **line description** , then the **line buffer** will consider the forthcoming pixel data to be "self- masked". Specifically, the **line buffer** will use one bit of each pixel's data as a mask: if the bit is 1, then it will attempt a write in accordance with the **write mode**; it the bit is 0, then the write attempt will be inhibited (the polarity can be reversed with an option in the object's **dispatch table entry**.)

I've used the word "attempt" in the previous sentence for a very specific reason. If the **write mode** is L, X, or LX, then the resulting mask for a pixel is the logical AND of the the **mask bit** in the **pixel storage cell** and the **embedded mask bit** of the pixel data being written. Thus, even if the **embedded mask bit** is set to 1 (normal polarity), the write could still be inhibited by the pixel cell's **mask bit** being set to 0. If the **write mode** is M, however, then the current state of the pixel cell **mask bit** is irrelevant.

The embedded mask function is independent of the pixel data write function, except insofar as to determine the pixel data **width**. The **embedded mask bit** is a particular bit of the **line description** pixel

data for each different **data width**. This bit is in a fixed position in the pixel data for each **data width** regardless of the **alignment**, the **write mode**, or the **display mode** of the data. The particular bit for each **data width** is shown below (7 bits/pixel **width** cannot have an **embedded mask bit**):



Embedded Mask Bit Placement in Line Description Data

The independence of the embedded mask function from the pixel write function is so complete that the **embedded mask bit** will be written into the pixel along with the other bits of the pixel data. Whichever mask value (0 or 1) permits writes will be the bit value written to the pixel at the location of the **embedded mask bit** after **alignment** and **write mode** adjustments.

This characteristic must be accounted for in the organization of the Color Lookup Table in **lookup table mode** and in the assignment of color values in **image mode**. The particular bit positions in the pixel data for the **embedded mask bits** were chosen with cognizance of the fact that 1, 2, 4, and 8 bits/pixel **widths** will primarily be used in **lookup table mode**, and the 16 bits/pixel **width** will primarily be used in **image mode**. Remember that the **color data** that is written for an object in **embed mask** mode shall have the **embedded mask bit** set to a constant. If the most significant bit of the lookup table **index** holds a constant value, then the group of colors selectable by the rest of the bits will be contiguous (assuming **alignment** = 0), a useful organization. If the least significant bit of Green in a 5-6-5 RGB designation is held to a constant, then we effectively reduced the RGB designation to 5-5-5, still quite

usable. Hence, the rationale for the **embedded mask** bit positions selected for the pixel data.

4 Positioning

4.1 Horizontal Object Positioning

A QuickScan object's **object description** , in general, is independent of the object's absolute position in display space. However, an object's **line descriptions** are, in general, dependent upon being positioned a certain way in display space relative to each other. Stated simply, when an object is repositioned, it should look the same except for regions of interaction with other overlapping objects. The characteristic of an object's subparts to maintain consistent positioning relative to each other despite the repositioning of the object as a whole I call **coherence** .

Maintaining vertical coherence is easy because QuickScan draws each object line-by-line without ever skipping or repeating any lines, no matter where an object is positioned. (Vertical positioning techniques will be discussed presently.)

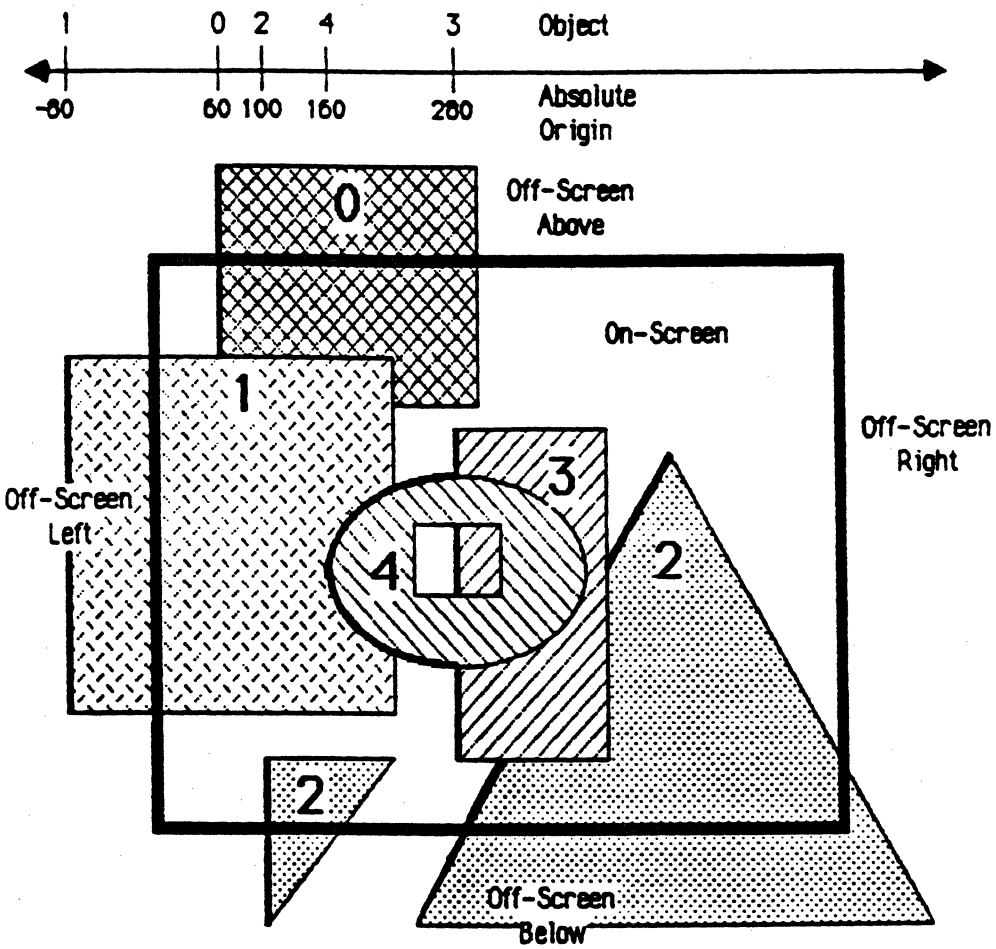
Maintaining horizontal coherence, however, is another story. **Line descriptions** can become exceedingly complex, often beginning at varying horizontal positions within the same object. Positioning **line descriptions** correctly requires a more powerful model that simply a fixed horizontal position for each object. QuickScan has two horizontal position descriptors to accommodate this requirement: the *absolute origin* and the *relative origin*.

The **absolute origin** is horizontal reference point in display space to which all horizontal positioning in the **object description** shall be referenced. If the **absolute origin** of an object is changed then the entire object shall move left or right without the loss of any coherence (except by deliberate hacks). The **absolute origin** of an object is specified in its **dispatch table entry** and holds for every line in the object (although it can be altered within a **line description** - the deliberate hack parenthetically referred to in the last sentence).

Having the same **absolute origin** for every line in an object is fine for a restricted class of objects (Mac windows fall in this class), but is insufficient for many useful object shapes. To accommodate variable horizontal positioning of each line in an object, yet maintain a global horizontal position for the object as a whole, we augment the object's

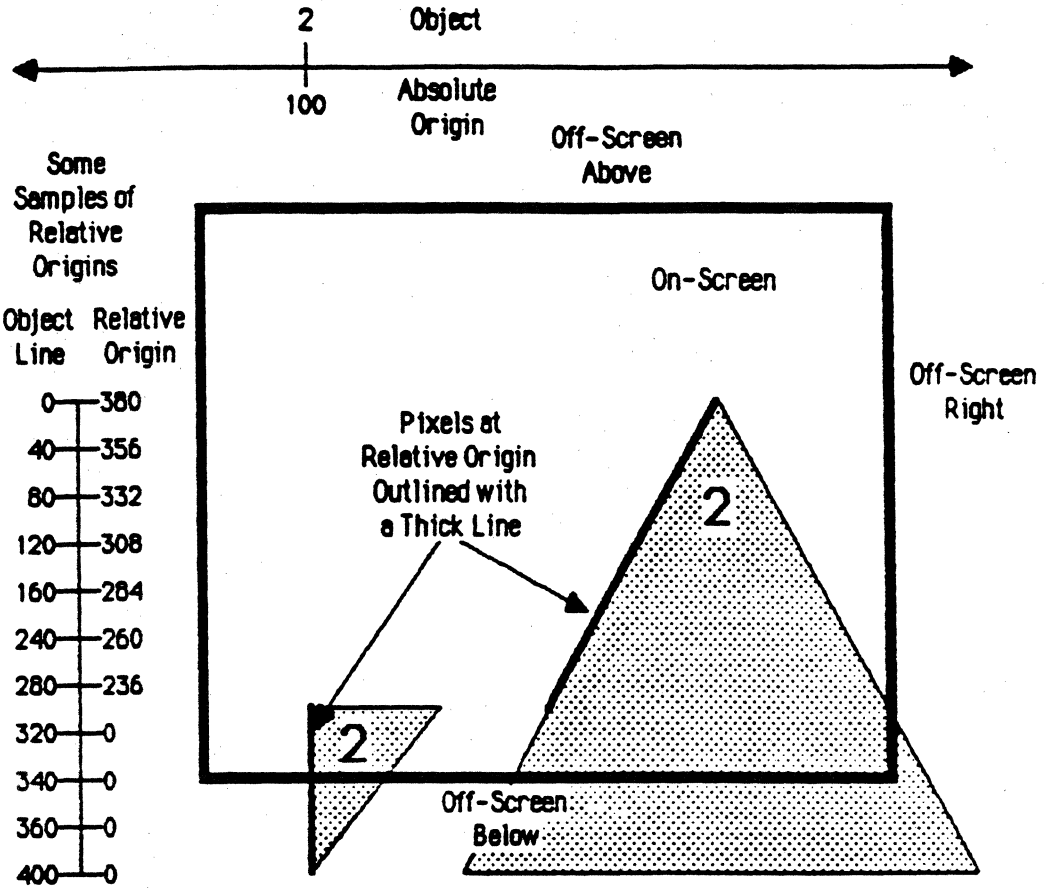
absolute origin with a relative origin for each line of the object.

A relative origin is specified in the line description for each line, and defines an offset to the right of the absolute origin at which to place the forthcoming line description data. Note that the absolute origin may be positive or negative and is referenced to the leftmost On-Screen pixel, but the relative origin may be only positive and is referenced to the absolute origin. Thus, objects may be positioned anywhere is display space, but an object's line descriptions must all lie aligned to or to the right of its absolute origin. The following diagram maps typical absolute and relative origins for the objects of Example 1:



Example 1 Absolute and Relative Origins

Note that the **relative origins** for the rectangular objects, 0, 1, and 3, all have zero value; the **absolute origin** is sufficient for such objects. Objects 2 and 3, however, have different **relative origins** for just about every line. The following diagram explicitly shows some of the **relative origins** in object 2:



Relative Origins for Object 2

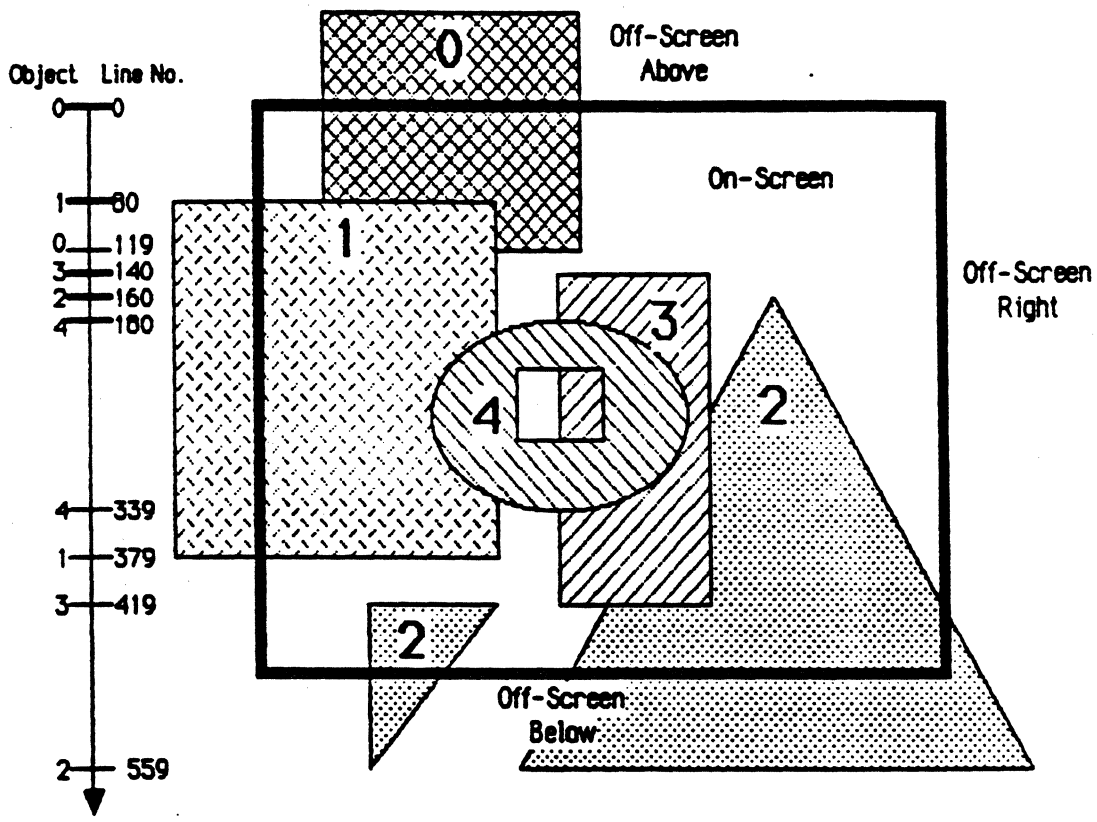
Although in general, each **line description** of an object has a single **relative origin**, in a complex **line description** each subpart of the **line description** has its own **relative origin**. Each of these **relative origins** are independently relative to the **absolute origin**, not to each other.

Note also, there is no rule requiring an object's **absolute origin** to

be aligned with the object's leftmost displayable pixel. The above diagrams show this situation to reduce confusion, but the only requirement on the location of an object's **absolute origin** is that it is aligned to or to the left of the leftmost pixel of the object. There could very well be a significant gap between the **absolute origin** and the leftmost pixel in the object.

4.2 Vertical Object Positioning

QuickScan objects are positioned by an absolute offset from the top of display space. The parameter for this specification, the *start line,* is a non-negative number and identifies the line number in display space at which the first line of the object shall be displayed. This **start line** parameter is specified in a field in the **dispatch table entry** for the object. The **start line**, and other, values for Example 1 are shown below:



Example 1 Objects Start and End Lines

Unlike the horizontal extent of an object, the vertical extent of an

object is not implicit in the **object description** . Consequently, an additional parameter must be specified in the object's **dispatch table entry** which identifies the last line of an object. Although we could specify the "end line" of an object (see the end lines of Example 1 in the above diagram), we would then have to change both the **start line** and the end line when the object moved vertically. Thus, the parameter specified is the **object height** (actually the **object height** less 1), a non-negative number which indicates the number of lines in an object (-1). To move an object vertically we need only change its **start line** parameter; the **object height** will stay the same.

You may have noticed in the diagram above that object 0 starts 80 lines before the first line of the screen, yet its **start line** parameter points to line 0. This is because no object may start before the first line of display space. What is not shown in this diagram is the fact that the **start address** parameter (to be covered shortly) in object 0's **dispatch table entry** has been changed by the 68020 to indicate the **object description** for this object begins at the 81st line of the actual **object description** , thereby "pulling a fast one" on QuickScan so that the proper image is displayed. Note that QuickScan now knows of only the lower portion of the object, and as such, the **object height** parameter has been adjusted accordingly.

Although this vertical cropping procedure appears to be an onerous burden for the 68020, bear in mind two things. Firstly, a window in Appleland cannot extend above the first line of the screen; indeed it can't even go above the menu bar. So, we never face this problem when our objects are windows or fully contained in windows. Secondly, **line descriptions** , by definition, are stored in independent, successive regions of memory. Finding the nth **line description** when we want to crop n-1 lines is at worst a linear search, assuming we have no higher level information obout the **object description's** organization (as you'll see shortly, the search is often as simple as one multiplication).

5. Instructions

5.1. Instructions and Execution Time

A QuickScan **line description** is a contiguous sequence of one or more *instructions*. Each **instruction** is either exactly one word long (a **single-word instruction**), or one or more words long (a **multi-word instruction**). **Single-word instructions** have only a **command word** (i.e. the instruction is the **command word**), but **multi-word instructions** have a **command word** and zero or more **data words** following the **command word**.

If you recall from the section on **line descriptions**, QuickScan employs a **single-line buffering** mechanism which loads a line into the **line buffer** while a previously loaded line is being displayed (see page 8). Consequently, QuickScan has exactly one display line's time to load each line into the **line buffer**. The next line cannot wait if a given line takes too long.

One display line's time is QuickScan's fundamental limitation in its ability to display objects. This constant is 31.778 microseconds (although it can be doubled for special TV-only displays). All **instructions** of all **line descriptions** which need to be displayed in a given line must complete their execution, with all associated overhead, within this time limit. Otherwise, not only will some foreground objects disappear for that line, but their display on lines below will be shifted down by one line.

Calculating the execution time of a **line description** is fairly straightforward. Each word of an **instruction** completes execution before the next word is loaded in, and every word in an **instruction** takes a determinate amount of time to execute. Each **instruction** in a **line description** completes execution before the next **instruction** is loaded in. Then, there is a certain overhead associated with ending one object's **line description** and starting up the **line description** of the next object to be loaded (i.e. the object of the next **priority** which is displayed on that line). Additionally, there is also some overhead incurred if the **line description** crosses a 1K byte boundary in RAM. Adding up these various times for all of the **line descriptions** on a line, we get the total execution time to load that line into the **line buffer**. This amount must be less than 31.778 µsec.

In the following sections, I will present the time overhead associated with each operation I am relating. Once we get through these sections we will be in a position to directly calculate exactly what QuickScan's object display limitations are. I think you'll be impressed.

5.2 The Instruction Set

This section describes the 6 instructions and 1 pseudo-instruction supported by QuickScan. The word formats can be found in Appendix A.

5.2.1 Context Switch

CSwitch a_origin, c_word_12, d_mode, e_polarity

where:

a_origin is the 12 bit, 2's complement **absolute origin**
c_word_12 is the lower 12 bits of the **constant word**
d_mode is the one bit **display mode** (1 = **image mode**, 0= **lookup table mode**)
e_polarity is the one bit **embedded mask** polarity (1 = 1 permits writes and 0 inhibits, 0 = 1 inhibits writes and 0 permits)

The Context Switch **single-word instruction** redefines the **absolute origin**, the **constant word**, the **display mode**, and the **embedded mask polarity**, generally in preparation for a forthcoming **line description**. Only the lower 12 bits of the **constant word** can be specified with this **instruction**. The upper 4 bits are automatically set to zeros.

This instruction may not be the last **instruction** in a **line description**. It takes 80ns (nanoseconds=10^{-9} seconds) to execute.

5.2.2 Replace Constant

RConst c_word,d_mode,e_polarity

where:

c_word is the 16 bit **constant word**
d_mode and **e_polarity** are as in the **CSwitch instruction**

Replace Constant is functionally identical to Context Switch except

that all 16 bits of the constant word can be specified and the absolute origin is not affected. This instruction allows you to change the constant word within a line description without knowing what the object's absolute origin is currently set to. You can also mess around with the the display mode and embedded mask polarity if you so desire.

This instruction may not be the last instruction in a line description . It takes 80ns to execute.

5.2.3. Bit Map

BMap d_format,w_mode,r_origin,dw_count,e_mode,end_line

where

d_format is the 5 bit **data format**
~~**w_mode is the 2 bit write mode**~~ 1 bit
r_origin is the 10 bit non-negative **relative origin**
dw_count is the 10 bit **data word** count
~~**e_mode is the 1 bit embedded mask mode select** (1= embed **masks**, 0= don't embed **masks**)~~
end_line is the 1 bit end of **line description** flag (1= last **instruction** in the **line description** , 0= not the last)

This **multi-word instruction** causes a bit-map to be loaded into the **line buffer**. The **data words** (see Appendix A for formats) following the **command word** actually contain the data that makes up the bit-map, and the **dw_count** parameter indicates how many of these words shall follow. If the **dw_count** parameter is zero, then the **instruction** will be ignored. If the **end_line** bit is 1, then after all of the **data words** have been loaded, QuickScan will initiate loading the next **line description** on the line.

The **d_format** parameter indicates the **data width** of the pixel data in the **data words** and the **alignment** of this pixel data in the **color data** of the **pixel storage cells** (see the section Pixel Data Write Formats for details). The encoding is as follows:

Width	Encoding
1BPP	1 A A A A
2BPP	0 1 A A A
4BPP	0 0 1 A A
8BPP	0 0 0 1 A
16BPP	0 0 0 0 1

only need 3 bits for format only
what about CLUT index?

A's indicate alignment code - see diagrams on pages 15-17.
Code 00000 is reserved.

The **w_mode** parameter determines to which part or parts of the pixel cell the pixel data will be written. It is encoded as follows:

Mode	Encoding	Pixel Sections Written
M	0 0	Mask Bit Only
L	0 1	L-Byte (Color Data L.S. Byte) and Mode Bit Only
X	1 0	X-Byte (Color Data M.S. Byte) and Mode Bit Only
LX	1 1	L- and X-Bytes (Color Data Word) and Mode Bit Only

The **e_mode** parameter determines whether the pixel data shall be considered to have **embedded mask bits** (see the section on **embedded masks** for details).

The **r_origin** parameter specifies the offset to the right of the current **absolute origin** at which to begin writing to **pixel storage cells**. Each subsequent write of pixel data shall be one **pixel cell** to the right of the **cell** just written. Thus, bit-maps are loaded into the **line buffer** left-to-right starting at **r_origin**.

All pixel data from the **data words** specified with **dw_count** shall be loaded into the **line buffer**. Hence, all Bit Map **instructions** specify bit-maps which are made up of an integral number of ~~32 bit~~ words. If you require a bit-map which ends or begins with less than a full 32 bit word, you must provide masking for the undesired bits.

why not pixel count mask before bit map then unmask

The execution time for the Bit Map **instruction** varies from a number of conditions. The **command word** has a fixed execution time, then each **data word** has a execution time determined by a) the data **width**, and b) whether the **data word** is the last word in the **line description**. The various permutations with executions times are shown in the following

table:

Depth (BPP)	Pixels per Data Word	Execution Time (ns) Command Word	Each Data Word Except Last of Line	Last Data Word of Line
1	32	80	80	80
2	16	80	40	80
4	8	80	40	80
8	4	80	40	80
16	2	80	40	80

5.2.5. Run

Run w_mode,d_align,r_origin,r_limit,data_7,end_line

where

w_mode is the 2 bit write mode
d_align is a 1 bit data alignment code (1= align to X-Byte, 0= align to 1-Byte)
r_origin is the 10 bit non-negative relative origin
r_limit is the 10 bit non-negative relative limit
data_7 is the 7 bit run data
end_line is the 1 bit end of line description flag (1= last instruction in the line description , 0= not the last)

This single-word instruction specifies a contiguous sequence of pixel cells which are to be written to with the same pixel data. The extent of this multi-pixel write is called a *run*. Formally, as Bennet so astutely observed, a run is 0 bit/pixel bit-map (2^0 colors = 1 color), and this instruction provides a short-hand means to specify that 1 color and the limits of the 0 bit/pixel bit-map. The Run instruction is invaluable in efficiently laying down large expanses of a single color and in setting up large masks. There is no other display subsystem commercially available that acheives this function nearly as fast as QuickScan.

Since so much information is packed into this **instruction's** single word, we have to compromise some of the data format flexibility we have with the other **instructions** . The Sequential Runs and Run Screen **instructions** provides runs without loss of data generality.

When the **end_line** flag is set, QuickScan will end the current **line description** and initiate the loading of the next **line description** in the **line buffer** immediately after completing the run.

The **data_7** parameter provides 7 bits to be used as the pixel data for all **pixel cells** affected by the run. This data is called the **run data** and is handled very much like 8 bit/pixel pixel data except that the most significant ~~bit of the 8 bits~~ written to the **pixel cells** is provided by the constant ~~**word**, not~~ the pixel data.

The **w_mode** parameter determines to which part or parts of the **pixel cells** the **run data** shall be written. It follows the same coding as in the Bit Map **instruction** .

The **d_align** parameter is a 1 bit data **alignment** code that provides a means to align the **run data** in the **pixel cells**. See the diagram on Page 16 for details.

The **r_origin** parameter specifies the **relative origin**, an offset to the right of the current **absolute origin** at which to begin the run.

The **r_limit** parameter specifies the **relative limit**, an offset plus one to the right of the current **absolute origin**, at which to end the run. If r_limit is less than or equal to r_origin, then no pixel cells will be written. This is because QuickScan can only generate runs from left-to-right. Also note that **r_limit** is not relative to the **relative origin**, but rather to the **absolute origin**. Hence, if the **relative origin** is changed, the run's length will change accordingly.

Embedded masks may not be specified ~~in a Run **instruction**~~ (such a capability is not useful in a single run). ~~A Run **instruction** takes 80ns to~~ execute.

5.2.5. Sequential Runs

SRuns d_format;w_mode,r_origin,dw_count,e_mode,end_line

where

d_format is the 5 bit **data format**
w_mode is the 2 bit **write mode**
r_origin is the 10 bit non-negative **relative origin**
dw_count is the 10 bit **data word count**
e_mode is the 1 bit **embedded mask mode** select (1= **embed masks**, 0= don't **embed masks**)
end_line is the 1 bit end of **line description** flag (1= last **instruction** in the **line description**, 0= not the last)

This **multi-word instruction** provides a means to efficiently specify a contiguous sequence of runs. It also allows full **data format** and **embedded mask** capability with runs (except 16 bits/pixel data **width**). Sequential Runs are very useful for efficiently describing adjacent regions of color, complex masks, and cartoons.

The Sequential Runs **command word** sets up the forthcoming run sequence almost exactly as the Bit Map **command word** sets up the forthcoming bit-map. The only difference is the **relative origin** indicates the first pixel of the run sequence rather than the first pixel of the bit-map, and the forthcoming **data words** contain run descriptions rather than bit-map descriptions.

So, for the details of the Sequential Run parameters, see the Bit Map instruction. The only restriction is that you may not specify 16 bits/pixel data **width** in the **data format**. If you do, the resulting writes to the **pixel cells** are indeterminate. When the **end_line** bit is set, this **line description** will end, and the next **line description** will be initiated to begin loading as soon as the last run specified in this **instruction** has completed.

Each **data word** holds 2 16 bit *run descriptions*. Each **run description** is made up of an 8 bit **run data** field called **data_8**, and an 8 bit **run length** field which specifies the length of the run (see Appendix A for a word format diagram). Runs are sequenced in order of the **data words**, and then within each **data word**, first the low-order run

description and second the high-order **run description**.

The very first run begins at the **relative origin** and extends to the right the number of pixels of its **run length**. Then, the second run begins at the pixel directly to the right of the last pixel of the first run and extends the number of pixels of its **run length**. The third run starts immediately to the right of the second run, and so on, until all of the **data words** specified in the **dw_count** have been loaded. If we specify a run with a **run length** of zero, then no pixels will be written with its **run data**, and the succeeding run will begin at the pixel where the run would have begun. ~~If we specify a run when **embed mask mode** is selected and the **embedded mask bit** in the run's **run data** is set to the write inhibit state, then no pixels will be written, but the succeeding run will begin where the run would have ended.~~

The **run data** of all runs in the run sequence will be adjusted for the **pixel cells** by the **data format** and **write mode** specifications exactly the same as Bit Map pixel data is adjusted. Although runs are formally 0 bit/pixel bit-maps, the width specified in the **data format** shall be used to determine how many bits of the **run data** shall be used and how many shall be be provided by the **constant word**. If not all 8 bits of the **run data** are used (i.e. in 4 BPP mode), then the least significant bits of the **run data** shall be used as the pixel data and the most significant bits will be ignored.

Sequential Runs always specifies an even number of runs. If an odd number is desired, then the last run should be either masked or given a length of zero. If **dw_count** is zero then the **instruction** will be ignored. A Sequential Run **command word** takes 80ns to execute, and each **data word** takes 160ns to execute.

5.2.6. Run Screen

RScreen d_format,w_mode,data_16,e_mode,end_line

where

d_format is the 5 bit **data format**
w_mode is the 2 bit **write mode**
data_16 is a 16 bit **run data**
~~**e_mode** is the 1 bit **embedded mask mode**~~ select (1= **embed**

masks, 0= don't embed masks)
end_line is the 1 bit end of line description flag (1= last instruction in the line description, 0= not the last)

This instruction generates a run across the entire On-Screen region of display space. It is useful for setting the background color or initializing all of the mask bits. Note that this run's position is fixed from pixel 0 to pixel 639, regardless of the value of the absolute origin.

The parameters d_format, w_mode, e_mode, and end_line function exactly as they do in the Sequential Runs instruction except for the fact that they apply to this single run, and that the 16 bit/pixel width is allowed. The data_16 field provides 16 bits of run data, utilized by the same rules as the Sequential Runs instruction.

A Run Screen instruction takes 80ns to execute.

5.2.7. No Operation

NOp end_line

where
end_line is the 1 bit end of line description flag (1= last instruction in the line description, 0= not the last)

This single-word instruction serves as a place holder in a line description. It is coded as either a Bit Map or Sequential Runs instruction with zero data words, so the only useful parameter is the end_line parameter for if you want the No Operation as the last instruction in a line description.

No Operation, no matter how it is coded, takes 80ns to execute.

6. Dispatching

6.1 The Dispatch Table Entry

As mentioned in the early parts of this document, each object has associated with it a 4 word **dispatch table entry** which defines the attributes of the object and identifies where the object may be found in RAM. This section discusses the content of these 4 words in detail. A word format diagram may be found in Appendix A.

Each **dispatch table entry** contains the following fields:

Field name	Bits
start address	20
line mode	1
line length	10
start line	9
object height	9
absolute origin	12
constant word	12
viewport origin	10
viewport limit	10
display mode	1
~~e_polarity~~	~~1~~
first word	32
bus_access	1

6.1.1 Start Address

This parameter is a pointer to the word (32 bit) in RAM (generally) which is the beginning of the first **line description** of the **object description** . The rest of the words in the **object description** follow forth from this address.

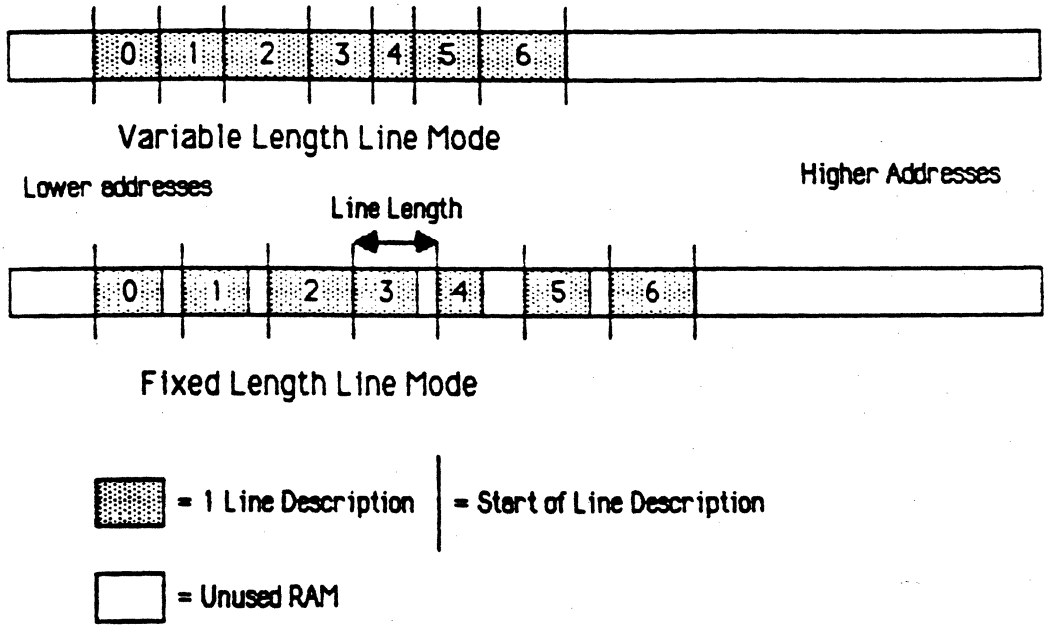
(The reason I qualified the term "RAM" in the above paragraph is because when we add graphics engines to the display subsystem, the **start address** pointer can point to *synthetic* objects generated by the engines as well as *actual* objects specified by **object descriptions** in RAM. Just as we address I/O ports in I/O devices as well as bytes in RAM in a microprocessor's address space, we address **synthetic** objects in graphics engines as well as **actual** objects in RAM in QuickScan's address space. For the purposes of learning QuickScan assume all objects are

actual and reside in RAM. See the discussion of **synthetic** objects in Appendix B.)

6.1.2. Line Mode and Line Length

The **line mode** bit specifies how QuickScan shall determine at what address in RAM to find each successive **line description** after the preceding **line description** ends. If this bit is 0, then the **line mode** shall be *variable length*, and a succeeding **line description** shall begin at the word following the last word of the preceding **line description** . If this bit is 1, then the **line mode** shall be *fixed length*, and a succeeding **line description** shall begin at the address determined by the sum of the address of the start of the preceding **line description** plus the **line length** . In **variable length** mode the **line length** parameter is ignored.

The following diagram shows a comparison between the two **line modes** . Note that while **variable length** mode uses RAM more efficiently, **fixed length** mode structures the **line descriptions** so that they are easier to locate by the 68020 (e.g. for vertical cropping).



Line Mode Comparison

Note that the **line length** parameter is independent of the **end_line** bit specified in an **instruction** . The end of a **line description** is specified by the **end_line** bit, and the address increment to the next **line**

description is specified by the **line length** bit. But by fiddling with the relationship between these two parameters we can get some interesting effects (see Applications, below).

6.1.3. Start Line and Object Height

The **start line** parameter is a non-negative integer which specifies the line of display space on which the object's first **line description** is to be displayed. The **object height** is a non-negative integer which, when summed with the value of **start line** specifies the line of display space on which the object's last line is to be displayed. Note that **object height** specifies the object's actual height in lines minus 1. Note also that there are only 484 displayable lines, so if you specify **start line** to be 484 or greater, then the object will not be displayed at all. (This is, in fact, the recommended technique for blanking an object.)

6.1.4. Absolute Origin

The **absolute origin** parameter is a 12 bit 2's complement value which specifies the **absolute origin** for the object. See the section, Horizontal Object Positioning, for details on the **absolute origin**.

In almost all **object descriptions** that I can envision, we would not want to change the **absolute origin** within the **object description**; the **dispatch table entry** specification will be sufficient. But, if you like to hack, the facility exists to change it with the CSwitch **instruction**. Note, however, that the **absolute origin** will revert back to the value specified in this **dispatch table entry** parameter before executing each successive **line description**.

6.1.5. Constant Word

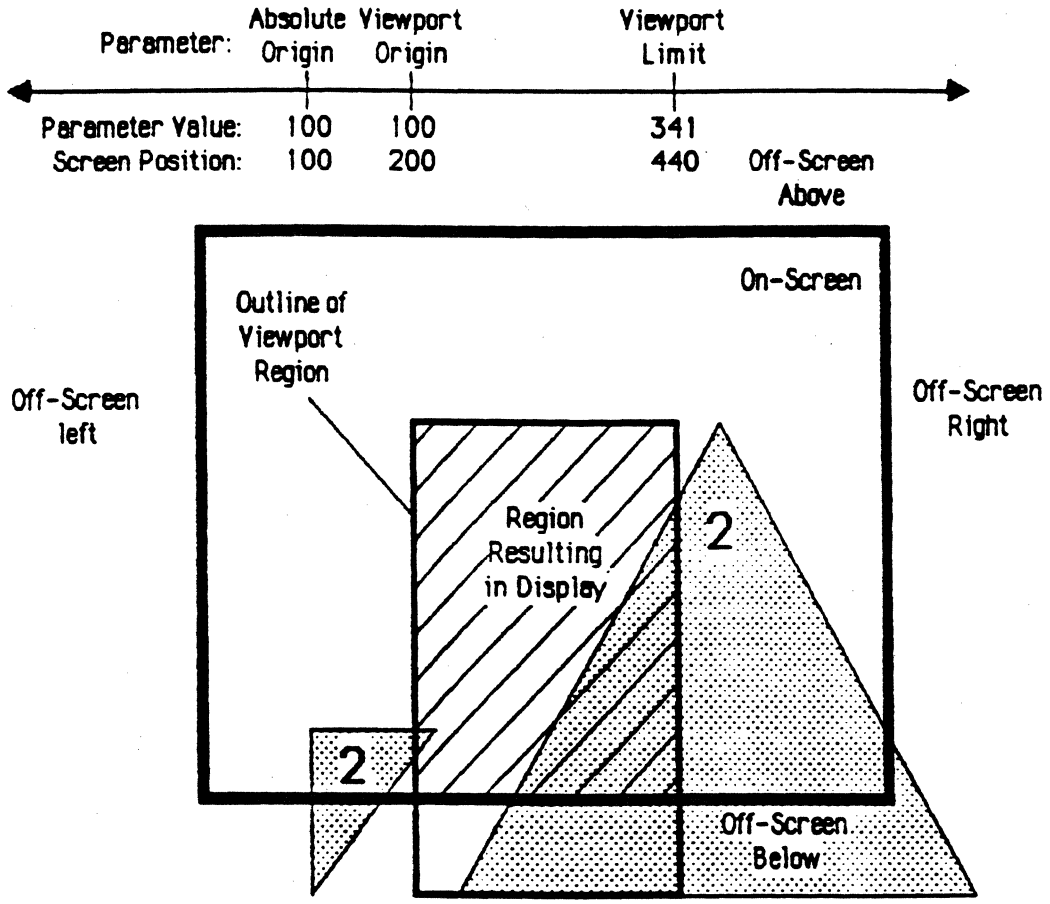
The **constant word** parameter specifies the lower 12 bits of the **constant word** for the object. The upper 4 bits are automatically forced to zeros. If you wish to change the **constant word** within a **line description** or if you want to give a value to all 16 bits, then you must use the RConst **instruction**. Note, however, that the **constant word** will revert back to the value specified in this parameter before executing each successive **line description**. See the section, Pixel Data Write Formats, for details on the **constant word**.

6.1.6. Viewport Origin and Limit

The **viewport origin** and **limit** parameters are each 10 bit

non-negative integers and specify an automatic viewport for every line of the object. A viewport is a region in display space wherein an object may be displayed. Any parts of the object outside of the viewport will not displayed. Viewports are created by clearing all mask bits on the screen (diabling writes to all pixels), then selectively setting those mask bits within the region where the viewport is desired.

The automatic viewport provided by QuickScan is simply a rectangular area of the same height and vertical position as the object with a width and horizontal position defined by the **viewport origin** and **limit**. The following diagram shows an example of such an automatic viewport:



An Automatic Viewport Applied to Object 2

Note in this diagram that a viewport may extend into Off-Screen area, and only the portion of the viewport that is On-Screen will result in a

displayable image.

The **viewport origin** is an offset to the **absolute origin** that specifies the leftmost edge of the viewport. The **viewport limit** is an offset to the **absolute origin** that specifies the rightmost edge plus 1 of the viewport. If the **viewport limit** has the value of zero, then the automatic viewport will not be activated, and the pixel **mask bits** in the **line buffer** will retain the value they had at the end of the preceding **line description** . If the **viewport limit** is less than or equal to the **viewport origin** , but not equal to zero, then all pixel **mask bits** will be cleared and writing to all pixel **color words** will be inhibited.

6.1.7. Display Mode

The **display mode** bit specifies whether the object is an **image mode** (set to 1) or **lookup mode** (set to 0) object. The **display mode** can be changed within an **object description** , but it will revert back to this value at the beginning of each **line description** .

6.1.8. Embedded Mask Polarity

The **e_polarity** bit specifies the polarity of the **embedded mask bits** if **embed mask mode** is selected in the **object description** . The coding is shown in the following table:

	E_mask Bit State	
E_polarity State	Inhibit writes	Permit writes
1	0	1
0	1	0

The **e_polarity** may be changed at any time in an **object description** with the CSwitch or RConst **instructions** , but note that it will revert back to the state defined by this parameter at the beginning of each **line description** .

6.1.9. First Word

The **first word** parameter provides the first word of the first **instruction** of each **line description** in the **object description** . Only the second and subsequent words of each **line description** are stored in non-**dispatch table entry** part of RAM, as the first word of all **line descriptions** is kept in common in this **first word** parameter.

This parameter provides the means to specify a common initial **command word** for all **line descriptions** , thereby reducing storage requirements for objects whose lines are similar in structure. It also allows us to do large backgrounds with just using the 4 words in RAM needed for the **dispatch table entry**. See Applications for details.

If the **line descriptions** for the object have each only a single **instruction** (as is very often the case), then the **end_line** bit should be set in the **command word** specified by the **first word**. Then the **line description** will end with the completion of this single **instruction** , and it will be the only **instruction** executed in the **line description** .

As you can see, QuickScan may not get in the last word, but it always gets in the first...

6.1.10. Bus_Access

The **bus_access** parameter indicates that this **object description** is completely contained in the **dispatch table entry**, and no RAM bus access is necessary to load the **line descriptions** . The implication here is, of course, that the **first word** is a **single-word instruction** and happens to be the only **instruction** on every line (such is the case when an object draws a background). The reason this bit exists is because QuickScan can minimize the overhead in switching between a no **bus_access** object and the **line descriptions** of a yes **bus_access** object and also minimize interrupting the 68020's access to RAM.

If **bus_access** is 1, then bus access is necessary, if **bus_access** is 0, then no bus access is necessary.

6.2. Object Dispatch Overhead

As alluded to previously, there is a certain execution time overhead associated with ending one **line description** and starting the next. This overhead is a function of how the ending **line description** terminates and somewhat how the starting **line description** begins. The process of ending one **line description** and starting the next is called an *object dispatch*, the object whose **line description** is about to start is called the *dispatching object*, and the one whose **line description** has just ended is called the *terminating object*. The time lost in **dispatching** an object is called the *object dispatch overhead*.

There is a minimum **object dispatch overhead** of 320ns, and the following "IF statement" adds various amounts of time to this base:

IF the **dispatching object** is a no **bus_access** object (i.e. its **bus_access** bit is set to 0) **THEN** there is no additional overhead.

ELSE

IF the **terminating object** has exactly 0 words (after the **first word**) in its **line description** **THEN** the additional overhead will be 80ns.

ELSE

IF the **terminating object** has exactly 1 word (after the **first word**) in its **line description** **THEN** there will be no additional overhead.

ELSE

IF the **terminating object** has as its last **instruction** :

- Run
- Run Screen
- No Operation

THEN the additional overhead shall be 240ns.

ELSE

IF the **terminating object** has as its last **instruction** Bit Map
THEN
BEGIN
If the data **width** is 1 bit/pixel **THEN** the additional overhead shall be:

dw_count value	Additional overhead (ns)
0	240
1	160
2	80
≥3	0

ELSE

If the data **width** is 2,4,8, or 16 bits/pixel **THEN** the additional overhead shall be:

dw_count value	Additional overhead (ns)
0	240
1	200
2	160
3	120
4	80
5	40
≥6	0

END

ELSE

IF the **terminating object** has as its last **instruction** Sequential Runs **THEN** the additional overhead shall be:

dw_count value	Additional overhead (ns)
0	240
1	80
≥2	0

6.3. Row Boundary Overhead

QuickScan's RAM is organized into rows of 1K bytes each, and there is an overhead associated with a **line description** which crosses a row boundary. It is 560ns. Needless to say, you should plan your objects to not cross these boundaries.

6.4. CPU Bus Overhead

Since QuickScan shares the same RAM array as the 68020 CPU, QuickScan "steals" a certain number of memory bus cycles from the CPU. If the CPU is running out of ROM, or out of another memory array when these bus cycles are stolen, then its performance will not be affected. But, if, however, it wanted to get to the RAM array when QuickScan is using the RAM, then it will enter a wait state until QuickScan completes

its memory access.

It is difficult to assess precisely how QuickScan will affect the CPU's performance since we as yet don't have any hard specifications on the CPU board architecture. But, we can get some feeling of the percentage of available CPU bus cycles that will be stolen for a given collection of QuickScan objects.

For each **object dispatch** (that is a **bus_access** object) QuickScan steals the bus for 400ns. Additionally, there are 3 memory refresh cycles each line @280ns apiece (although the 68020 has this overhead anyway). And, finally there is a 400ns cycle stolen whenever a **line description** crosses a row boundary.

Each line is 31.778 µsec long, and each field is 16.66 ms long. There are 484 active lines, and there are 525 total lines. During inactive lines, memory refresh still continues, but QuickScan only does 3 memory accesses (@400ns apiece) for the Configuration Data, The Color Lookup Table, and the Object Dispatch Table during this time.

We'll make the conservative assumption that a CPU memory cycle is 280ns.

There are 59286 possible CPU memory cycles each frame time. Of these, 1575 cycles, or 2.7% go to memory refresh. 3 cycles, or .005% go for QuickScan configuration.

Each **object dispatch** (if the object is a **bus_access** object) takes 1.4 cycles so we can determine the total number of cycles for an object by multiplying its number of lines (except those Off-Screen Below) by 1.4. An object which is half the height of the screen (242 lines) takes 338.8 cycles, or .57%, an object which is the full height of the screen (and this is the worst case) takes 677.6 cycles, or 1.14%. Of course, we have to include row boundary crossings, but these shouldn't arise much in practice, and even if they did, they would happen only every few lines (1K bytes is a lot of **line descriptions**).

So let's take an absolutely worst case: Assume 64 objects, each **bus_access** and 484 lines tall. Assume that every row boundary is crossed (there are 256 in the memory array). Then we have 1575 cycles

for refresh, 3 cycles for QuickScan configuration, 43366.4 (677.6 * 64) cycles for objects, and 358.4 cycles (256 * 1.4) for row crossings. That gives us a grand total of 45302.8 cycles per frame or 76% of the available CPU cycles.

Now, if 76% seems like a monstrous number, consider that we have 64 484-line objects gobbling up an entire 256K RAM array, and the CPU still gets in there almost 1/4 of the time. It can still run out of ROM or another RAM array at full tilt. Or, if you consider that our 68020 will be running about 6 times the speed of the Mac without cycle-stealing, then it would still be running about 1.5 times the speed of the Mac in this absolutely worst-case scenario if it were running solely out of the shared RAM array.

For any practical display that I've thrown together, the total CPU cycles stolen rarely go beyond 15 or 20%. (Note that the Mac itself loses about 25% of its CPU cycles to its 1 bit/pixel vanilla graphics display.) In comparison to any shared-memory display device that I've seen, QuickScan is extraordinarily efficient for what it puts up on the screen.

7 Applications

Now that you know all about programming QuickScan, it will help cement your knowledge to consider a few examples. The following sections show how to generate and manipulate some simple objects with QuickScan. Hopefully, some of the more obscure modes and functions we've discussed in the preceding chapters will show their usefulness here, and you'll get an idea of what I had in mind when I dreamed them up.

7.1 Rectangular Bit-Maps

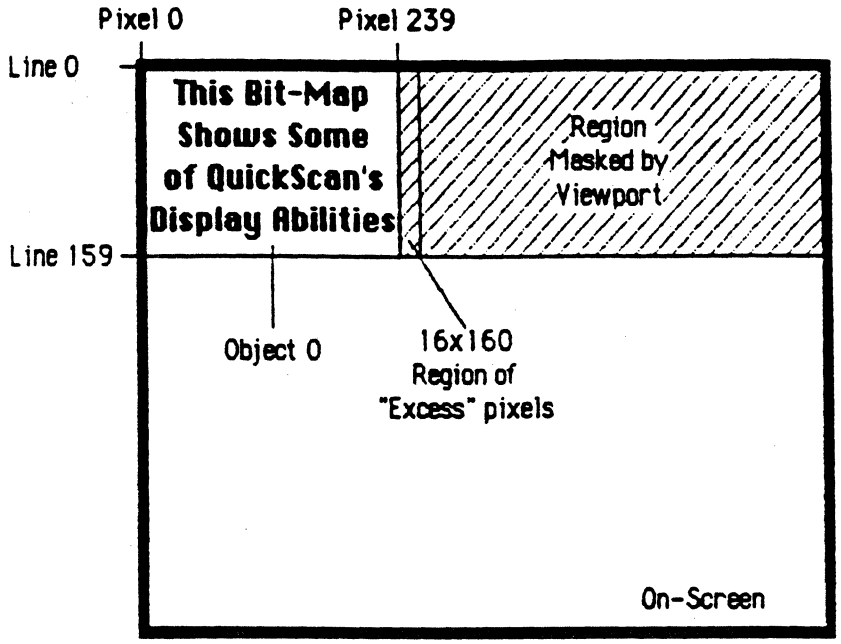
QuickScan has been especially optimized to support rectangular bit-maps, providing convenient, linear RAM organization and manipulation primitives with as little regard to the physical position of the bit-map as possible. At the same time QuickScan supports bit-maps with full generality to allow their inclusion as sub-units of complex **object descriptions** .

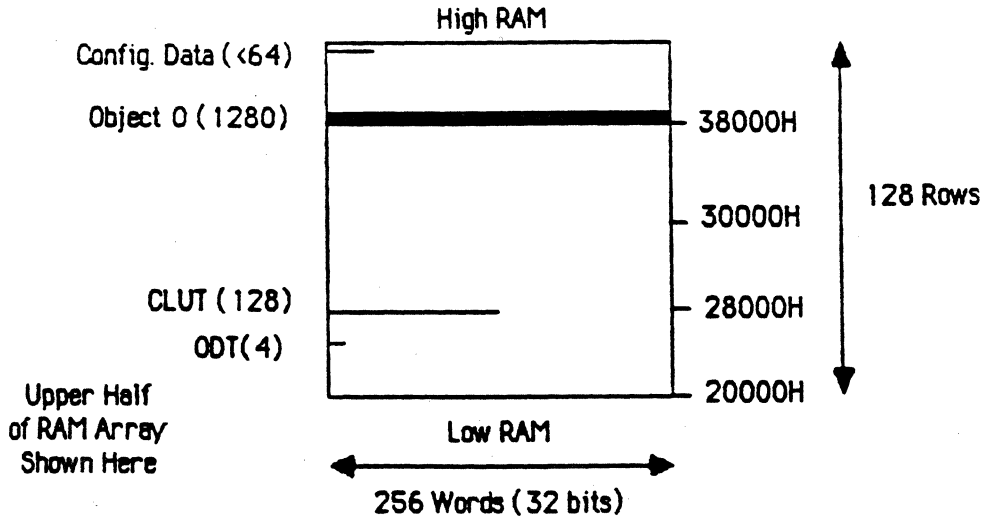
The tricky thing about maintaining both a nice linear bit-map array and full generality for complex **object descriptions** is that the former requires that the bit-map image in memory be entirely of data packed line-by-line, yet the latter requires that the bit-map image in memory be one or more **instructions** , thereby allowing the bit-map to be separated from other sub-units of the **object description** when it is decoded. Clearly, each representation has its place: we want the linear array when we have Mac-like windows with text and presentation graphics, we want the complex **object description** when we have a "freeze-dried" object downloaded from an application because of its compactness and ease of manipulation. How can we resolve this philosophical discrepancy and still maintain consistency?

To the rescue comes the **first word** of the **dispatch table entry** . The deal is: all bit-maps, like all QuickScan graphics primitives, are specified with **instructions** . When a bit-map is needed, then a Bit Map **instruction** is specified in a **line description** , precisely as it has been described in Chapter 5. This, of course, takes care of the complex **object description** requirement; now you can put a bit-map within an object as desired. And, it takes care of the linear array requirement because such a data structure results when the first word is a Bit Map instruction command word. Let's take a closer look at exactly how this is so by working through an example.

7.1.1. The Basic Rectangular Bit-Map

The figure below shows a simple 1 bit/pixel bit-map with dimensions of 240 horizontal and 160 vertical. The content of the bit-map happens to be a text message of black letters on a white background. A memory map is also shown detailing where memory is RAM is used to support this display:





Note: RAM array proportions are realistic: one line (——) is one row thick.

First of all notice that we are looking at the upper half of a 256K RAM area, and that the memory is divided into rows of 256 32-bit words (128 rows are shown, 256 rows are available). Notice also that the black area allocated for each block of data is pretty accurate, so you can think about how much RAM is takes to store things as you work through this example (but the ODT is longer than it should be so as to make it visible).

Some terms: the ODT is the Object Dispatch Table (see sections 2.1, 2.2, and 6.1) and the CLUT is the Color Lookup Table (see section 2.4). The Configuration Data is not yet completely defined, but for our purposes, we shall say that it contains pointers to the ODT and the CLUT.

In setting up this display, first we decide where we want to put the CLUT and the ODT. The CLUT is 128 words long, and can be placed at any place in RAM provided that it does not cross a row boundary. We place it here at 28000H (note that QuickScan measures data in 32-bit words, yet I specify byte addresses). The ODT must begin at a multiple of 1024 bytes in RAM, so we see it here placed at 26000H.

Next we allocate some space for the bit-map. I claim that the bit-map can be set up as a linear array, one line following the next in memory, each line rounded up to an integral number of words. Since the horizontal dimension is 240 pixels, and we have 1 bit/pixel, then we need 240÷32 = 7.5 words to hold each line. We must round up to a whole word, so we need 8 words to hold each line. There are 160 lines, so the total RAM requirement for this bit-map is 160*8 = 1280 words. Let's place this data at 38000H. It extends to 384FFH.

Now we need to set up the **dispatch table entry** for the object. This is essentially the <u>definition</u> of the object. Let's go through each parameter (reference section 6.1).

<u>Start Address</u>

This parameter points to the beginning of the **object description** : address 38000H. Notice, however, that the number coded is D000H (38000H÷4) because we are specifying a word address, not a byte address.

<u>Line Mode</u>

This parameter specifies whether the **line descriptions** are

fixed length or **variable length**. In this case, either mode will work because the bit-map **line descriptions** are of fixed length, so we could specify the length in **fixed length** mode, or let QuickScan figure out the length by specifying **variable length** mode. But, why bother specifying the length? Well, this first part of the example doesn't show why, but you'll see why it's important in a little bit. Thus, for this example we'll specify "1" for **fixed length** mode.

Line Length

The length of each **line description** in RAM is 8 words. We need to specify this parameter because we are in **fixed length** line mode. Notice that this parameter does not include the **first word** as part of the length.

Start Line

This object begins at the first line of the On-Screen area, line 0 (see diagram).

Object Height

The vertical dimension of this object is 160, so that is its height. But, QuickScan requires that when this parameter is summed with the **start line** that the result is the end line, line 159. So, the amount coded for this parameter is the height minus 1, or 159.

Absolute Origin

This object's leftmost pixels are at pixel 0 of display space. We could specify the **absolute origin** to be any value that is 0 or smaller, but for the sake of simplicity we shall specify 0.

Constant Word

Since we only have 2 colors in this example, black and white, we might as well put them at the beginning of the CLUT. Let's plan on aligning the 1 bit of the **pixel data** with the LSB of the **color data** word. So, setting the lower 8 bits of the **constant word** to 0 will cause the **pixel data** to select between the first and second CLUT entries.

The next 4 bits of the **constant word** will hold the **multiplier** we plan to apply to the output of the CLUT (see diagram on page 12) because at 1 bit/pixel, we haven't enough data to specify this value for every pixel individually. So, we'll assign each pixel the same

value for its **multiplier** by putting the desired value in the **constant word**. Now, this is jumping ahead to the Multiplier Applications section, but just understand that the 4 bit **multiplier** of the **color data** word will affect the *value*, or brightness, of the CLUT output. This doesn't bother our black CLUT entry, because black is black no matter how bright, but it will affect the intensity of our white backdrop, determining whether we have black, hot white, or one of 14 grey levels in between. Let's opt for average **value**, so let's specify 8 for the **multiplier**. This we place in the least significant nibble (LSN) of the upper byte of the **constant word**, and every pixel written will be given this same brightness.

The MSN of the **constant word** cannot be specified in this parameter, it will be set to 0 - which is just as well since those 4 bits have no meaning in a **lookup table mode** pixel. So, the **constant word** parameter is set to 800H.

Viewport Origin and Limit

These parameters specify what horizontal region of the bit-map **pixel data** will actually be displayed. If you recall, this bit-map was actually 240 pixels horizontally, yet we had to round up to the nearest whole word, as if the bit-map was 256 pixels horizontally. As it turns out, QuickScan cannot tell where the real pixels of the last **data word** of a Bit Map **command** end, and where the "excess" pixels begin, so we must prevent QuickScan from displaying these excess pixels. This can be accomplished with these viewport parameters.

The **viewport origin** identifies the pixel where the real bit-map begins, relative to the **absolute origin**. That pixel is 0 and the **absolute origin** is 0, so the **viewport origin** is 0-0 = 0. The **viewport limit** identifies the pixel where the real bit-map ends, relative to the **absolute origin**, plus 1. That pixel is 239 and the **absolute origin** is 0, so the **viewport limit** is 239-0+1 = 240. The excess pixel region (see the diagram above) from pixel 240 to 255 now is masked since the viewport extends only between pixel 0 and 239. Our desired horizontal dimension of 240 is now achieved.

Next for next.

Display Mode

We are in **lookup table mode** since we have only 1 bit to provide for each pixel. This bit is 0.

Embedded Mask Polarity

We are not using the **embedded mask** function now, so the value of this bit doesn't matter.

First Word

This word holds the Bit Map **instruction** and makes the linear bit-map array possible. When QuickScan is about to load a **line description** from RAM into the **line buffer**, first it will configure the **line buffer** with the relevant parameters listed above, and then it will load this **first word** as the **command word** of the first **instruction** of the **line description**. Only after that will QuickScan begin loading in the rest of the **line description** from RAM. In this example the **first word** contains a Bit Map **instruction command word**, and of course, a Bit Map **command word** is followed by **data words** containing the **pixel data** of the bit-map. These **data words** will be found, in this case, starting with the beginning of the portion of the **line description** in RAM...which is where our linear bit-map array is stored! Could the **data word format** expected by the Bit Map **command word** and the data format of a linear bit-map array be one and the same?

Well, it just so happens, that this is exactly the case. To see this let's look at the Bit Map **command word** and see how it fits together. We specify 1 bit/pixel mode with alignment to the **color word** LSB, or **d_format** 10000 (see page 15). We specify **w_mode** LX (11) because we wish to write the **multiplier** as well as the **index**. We have no offset from the **absolute origin**, so our **r_origin** is 0. Our horizontal dimension is 240 pixels, which rounds up to 8 **data words** each line at 1 bit/pixel, so our **dw_count** is 8. We do not have **embedded masks**, so the **e_mode** bit is 0. This is the last **instruction** for this **line description** (it is the only **instruction**) so the **end_line** bit is 1.

So, starting with the first line of the object what happens? The object is dispatched at line 0, and the **line buffer** is configured in accordance with the **dispatch table entry** parameters. Then, the **first word**, the Bit Map **command word** detailed in the preceding paragraph, is taken and executed. QuickScan prepares the **line buffer** for a bit-map and expects 8 **data words** to be fed in to describe the bit-map. The **start address** points to the first of these **data words**, indeed the first word of data for our linear bit-map array, and it and the following 7 words are loaded in to make up the

first displayed line for the object (note that the last 16 pixels are masked). Well, so far so good. Those 8 words corresponded to the first displayed line of the linear bit-map array.

On the second line, QuickScan again configures the **line buffer**, and again executes the same **first word**, and again expects 8 words of bit-map data. Only this time, the **start address** parameter is pointing to the 9th **data word**. It was automatically incremented by the value in the **line length** parameter: 8. So, it loads in **data words** 9 through16 (assuming we numbered them from 1), which then provides the data for the second displayed line of the object. Well, that's fine because the 9th through 16th words of the linear bit-map array happen to correspond exactly to the second line of the bit-map.

I think you can see how this process continues, displaying each successive line, sucking in each successive line of bit-map data until the end line of the object is reached, and the last line of data is loaded in. All the time the very same Bit Map **instruction** in the **first word** is used, and all we have stored in RAM is a nice, neat, convenient, linear bit-map array.

Bus Access

Since we must get to RAM to load the bit-map in, we must allow QuickScan to access the RAM bus. **Bus_access** is 1.

Now that we have our object completely defined, all we have to do is "turn it on." This is simply accomplished by taking our just prepared **dispatch table entry** and placing it as the first entry of 4 words in the ODT. We must also, of course, set up the CLUT with our two colors, black and white, in the first 2 CLUT entries, but I shall leave that explanation until the section on Multiplier Applications.

And, so, if you flip back a few pages to the diagram we started with, you can see the end result.

Let's get an idea how much execution time this example takes and how many CPU bus cycles it consumes: As far as execution time goes, we have one object, it is a 1 bit/pixel, bus_access, bit-map with 8 **data words** per line. Since this object is the first object on the line, it qualifies for the minimum **object dispatch overhead** of 320ns. Furthermore, any object following this one will also have minimum **object dispatch overhead** (see section 6.2). The Bit Map **command word** is in the first

word so its execution time is included in the **object dispatch overhead**, and we have 8 1 bit/pixel **data words**, so we use 80*8=640ns to load the **pixel data** (see section 5.2.3) for each line. Notice that no **line description** crosses a row boundary although the **object description** takes up some 5 rows (this is due to the fact that 256 (the words in a row) is a multiple of 8, our **line length**), so we have no row boundary overhead. Thus, we have a total object execution time of 320ns+640ns = 960ns, or just under 1 microsecond. To put that in perspective, we have 31.778 µsec available on each line for object execution (see section 5.1), so QuickScan is fast enough to display ($31.778 \times 10^{-6} + 960 \times 10^{-9} = 33.102$) 33 objects just like this on each line if we wanted it to (all qualify for minimum **object dispatch overhead**).

As far as stolen CPU bus cycles, we have a fixed overhead per 60Hz frame for the Configuration Data, the CLUT, the ODT, and RAM refresh of 1578 cycles, out of an available 59286 cycles, leaving us a remaining 57690 (see section 6.4). 59286 cycles is 100% efficiency: the CPU can access memory with no wait states whenever it wants to, but because of RAM refresh, 97% efficiency is about the best we achieve in practice. Let's see how much our object cuts into that figure. As noted about, the object is bus_access, and it has no row boundary crossings. Therefore, its total bus overhead is one **object dispatch** per displayed line. Each **object dispatch** takes 1.4 CPU bus cycles, and there are 160 lines, so we have 1.4*160 = 224 CPU cycles stolen. Adding that with the fixed overhead of 1578 cycles we have 1802 total cycles stolen, or the CPU is still running at about 97% efficiency! We haven't decreased performance by even 1 whole percentage point. If, as suggested in the previous paragraph, we put up 33 such objects at once, we'd have a total of 7392 CPU cycles stolen plus fixed overhead giving us still about 85% efficiency.

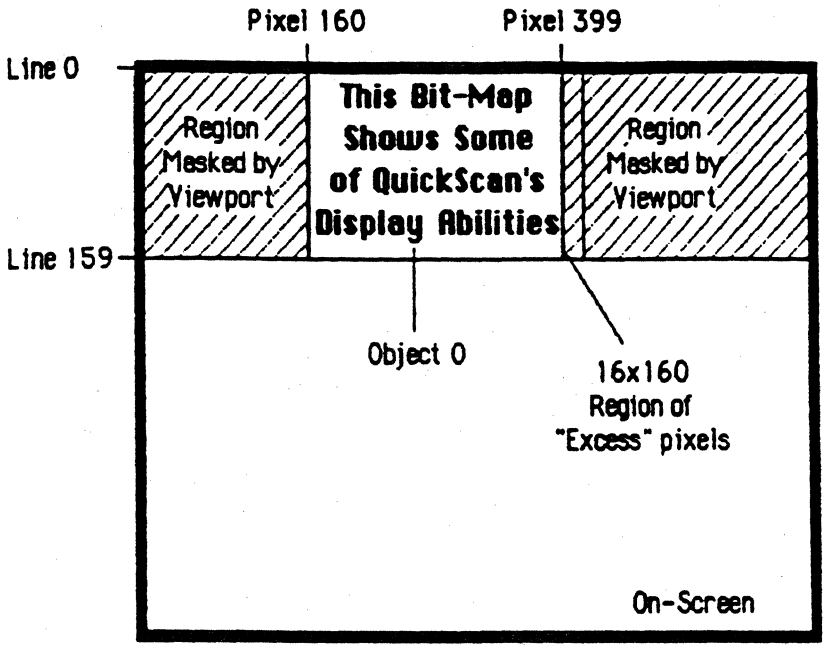
Now, seriously, you ought to be impressed. <u>There is no other display processor I have heard of which comes near to these performance figures</u>. <u>None</u> can put up 33 independent objects on one line, <u>none</u> can put more than a few large bit-maps such as the one in this example on one line, and <u>none</u> can put up half so many objects with such high resolution without bringing the CPU to its knees with cycle stealing. As you'll see as we work through more examples, QuickScan's performance is extraordinary.

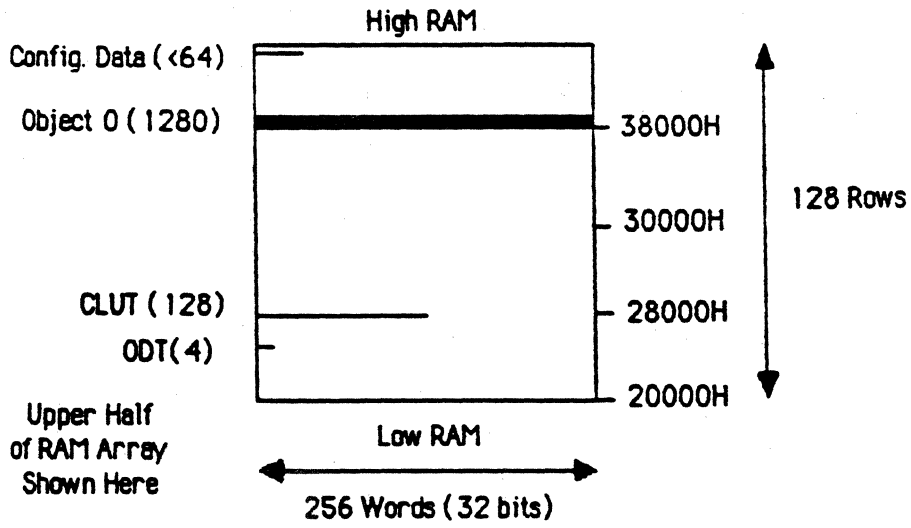
toot! toot!

7.1.2 Horizontal Positioning

Now that we've defined our object, let's consider what it takes to

manipulate it. A fundamental manipulation is positioning the object in display space. Positioning is divided into two separate steps with QuickScan, horizontal and vertical. Let's look at horizontal first. If we wanted to take our example object and reposition it 160 pixels to the right it would look like this:





Note: RAM array proportions are realistic: one line (——) is one row thick.

Notice that the memory map is identical to that of the object in its

original position. We don't have to move object descriptions in order to move objects. But, clearly something must be changed so QuickScan knows to move the object. That something is the **absolute origin** parameter in the **dispatch table entry**.

Whereas the **absolute origin** was set to 0 in section 7.1.1, it is set to 160 here. Now, the horizontal positioning within the **object description** is all referenced to 160 rather than to 0 and everything accordingly shifts 160 pixels to the right.

Notice that the viewport defined by the **viewport origin** and **viewport limit** has shifted along with the rest of the object, so the excess pixels are still appropriately masked. This is because these parameters are referenced to the **absolute origin** and are now offset by 160 as well. Notice, however, that we now have a region to the left of the object which is masked. It doesn't affect us in this example because nothing can be written to the left of the **absolute origin** anyway, but it comes into play in an example below.

If we actually moved this object from its original position to this new position as shown here, note that we could effect the change at any time, yet the display transition would occur between frames. That is to say, if QuickScan happens to halfway through displaying this object when the 68020 changes its **absolute origin** parameter, the rest of the object in that frame will still be drawn with the old **absolute origin** parameter. With many display processors, parameter changes take effect immediately, and consequently displayed objects may be changed partway through a single frame with an unsightly display aberration as a result. With QuickScan you are guaranteed coherence within each frame, regardless of when a parameter is changed. There is, however, a slight related restriction which I'll point out below.

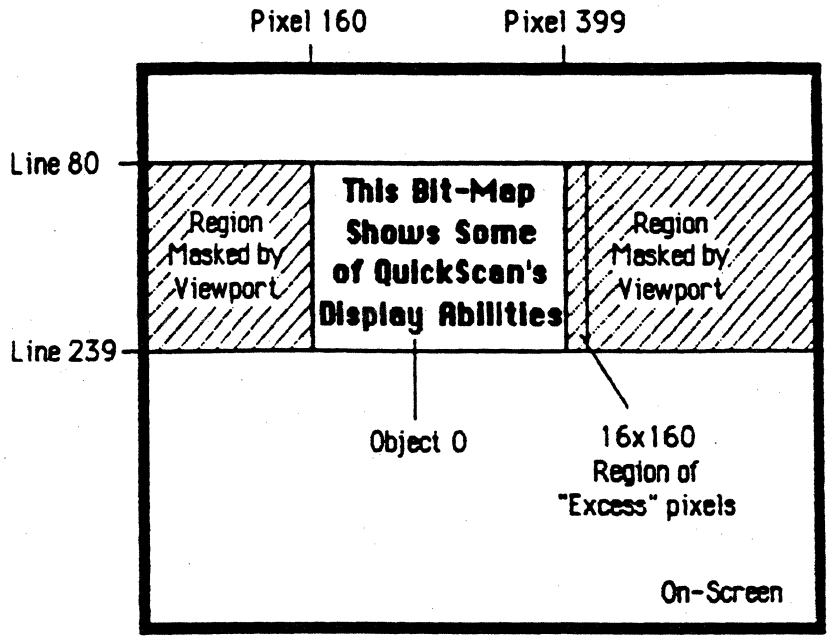
why?

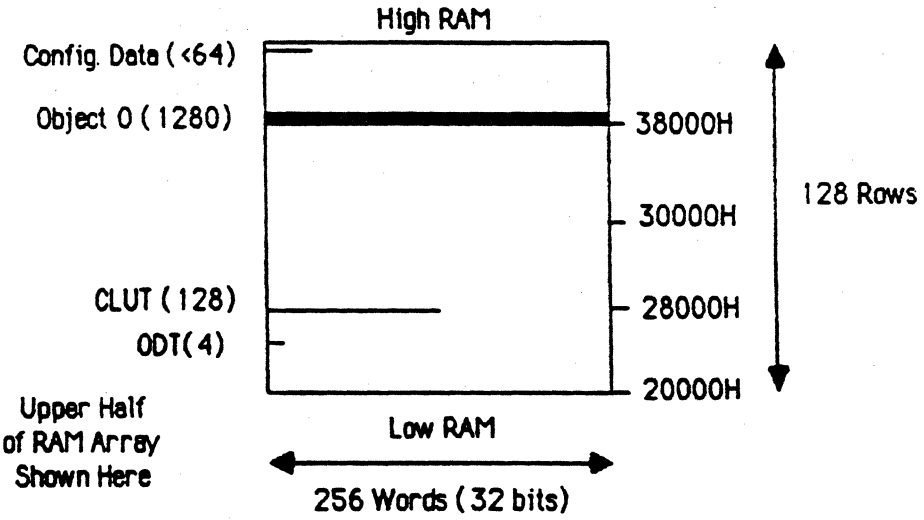
Since the memory layout and access characteristics are the same as that of the example in section 7.1.1, the execution time and CPU efficiency are the same.

7.1.3. Vertical Positioning

To reposition the object vertically, we need only change the **start line** parameter. If we wanted the object's first line to be line 80, then we'd simply change the **start line** parameter to 80 from its current value

of 0. QuickScan would then load the first **line description** at line 80, and each successive **line description** would be loaded with each successive line. The resulting image would look like this:





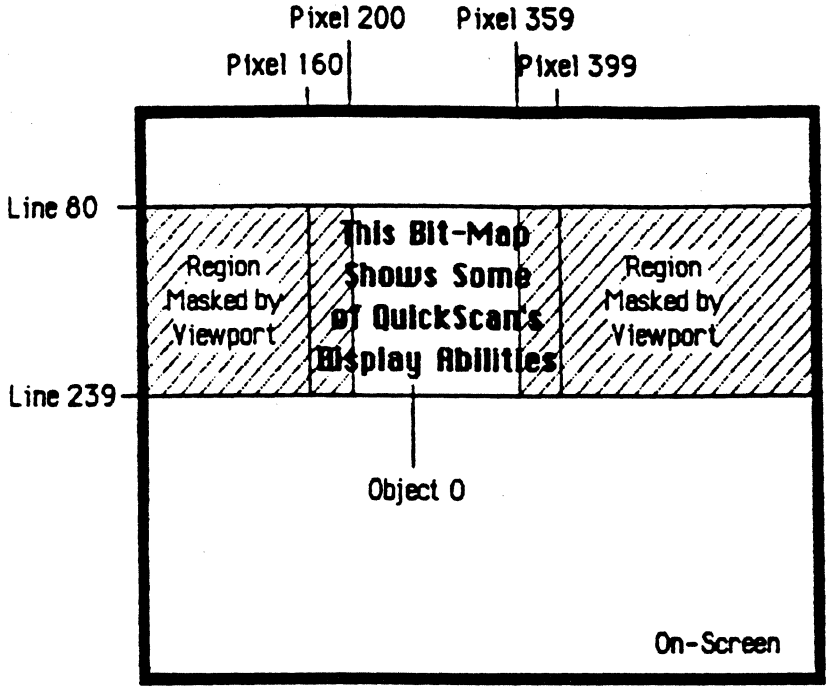
Note: RAM array proportions are realistic: one line (——) is onerow thick.

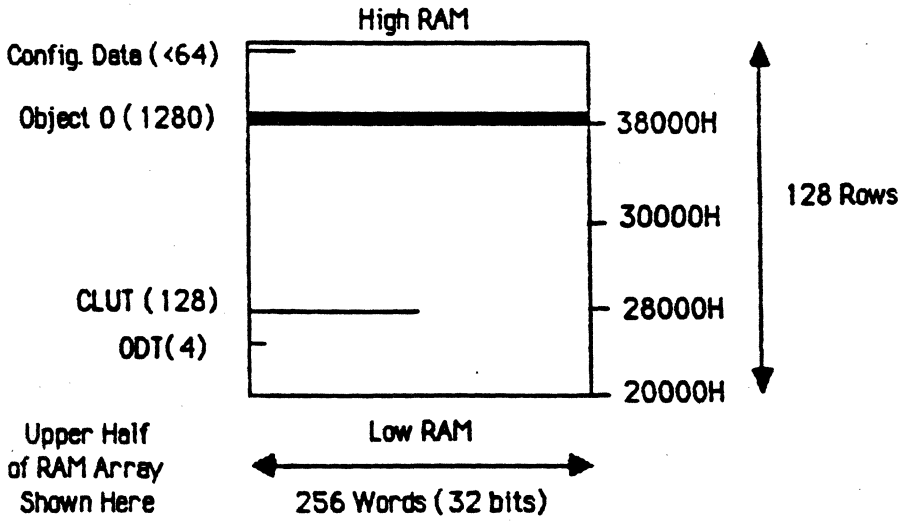
Notice that the memory layout remains exactly the same. Notice also that the previous horizontal positioning is not at all affected by this vertical change.

As with the horizontal change, no matter when the **start line** parameter is changed, the vertical shift will occur cleanly between frames. Also, the execution time and the CPU efficiency remain the same.

7.1.4 Horizontal Viewports

The QuickScan viewport mechanism can be used for more than just masking excess pixels. Consider the following display:





Note: RAM array proportions are realistic: one line (——) is onerow thick.

Here we are diliberately masking off some of the real pixels of the bit-map. This is logically what happens when a Mac window is sized down horizontally so that it is smaller horizontally than bit-map that it "holds", and you use the horizontal elevator to view different parts of the bit-map.

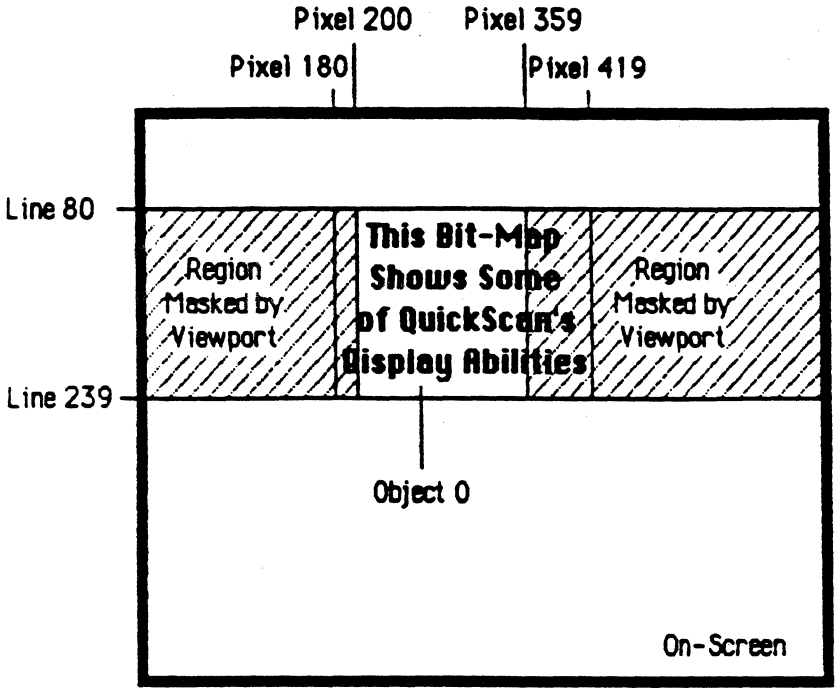
Notice that once again, the memory layout is unchanged. The whole effect is controlled by the **dispatch table entries**, **viewport origin**, and **viewport limit**. As I alluded to before, the left mask region would have some use, and indeed it does. Just as we saw the right mask region masking off the excess pixels, we now have the left mask region masking off some real pixels. Furthermore, the right mask region has been brought a bit to the left to mask some real pixels as well as the excess pixels. The viewport position and size is controlled just as you might expect: the **viewport origin** points to the pixels on the left edge of the viewport, relative to the **absolute origin**, and the **viewport limit** points to the pixels on the right edge of the viewport, plus 1 and relative to the **absolute origin**. It this case the **viewport origin** is 200-160=40, and the **viewport limit** is 359-160+1=200.

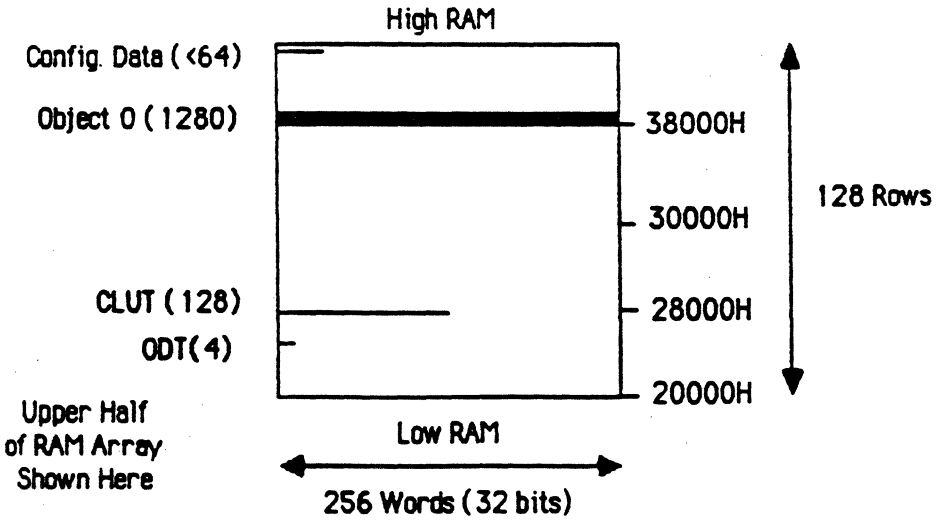
As in changing position, QuickScan guarantees that regardless of when the parameter change occurs, the object change occur between frames. But, it will not guarantee that both parameter changes will be applied before a frame is displayed. This is because of the fact that there is the extremely small possibility (1 chance in 59286) that QuickScan will load the ODT after the first parameter (**viewport origin**) is changed, but before the second parameter (**viewport limit**) is loaded. Then, one frame will be displayed with the new **viewport origin**, but the old **viewport limit**. Now, I realize that in this particular example it is no big deal, but it could be a significant problem given the right circumstances. I am considering incorporating a semaphore mechanism of some sort to hold off the ODT load if multiple parameters are being changed. The other possible solution is to prepare a second ODT in RAM with the changes, then in one write, change the ODT pointer to point to this new table. We'll think up something, but just be aware of this circumstance.

7.1.5 Horizontal Scrolling

If this bit-map were indeed a Mac window, then we would need some way to support the horizontal elevator, or rather we would need to support horizontal scrolling within the horizontal viewport. This effect is easily

achieved by just thinking carefully about what we are doing. We are not moving the viewport, we are moving the object. Hence, all that we have to do is change the **relative origin** of the Bit Map **instruction** in the **first word**, and the bit-map will move without disturbing the viewport. If we change this **relative origin** from 0 to 20, we get the following display:



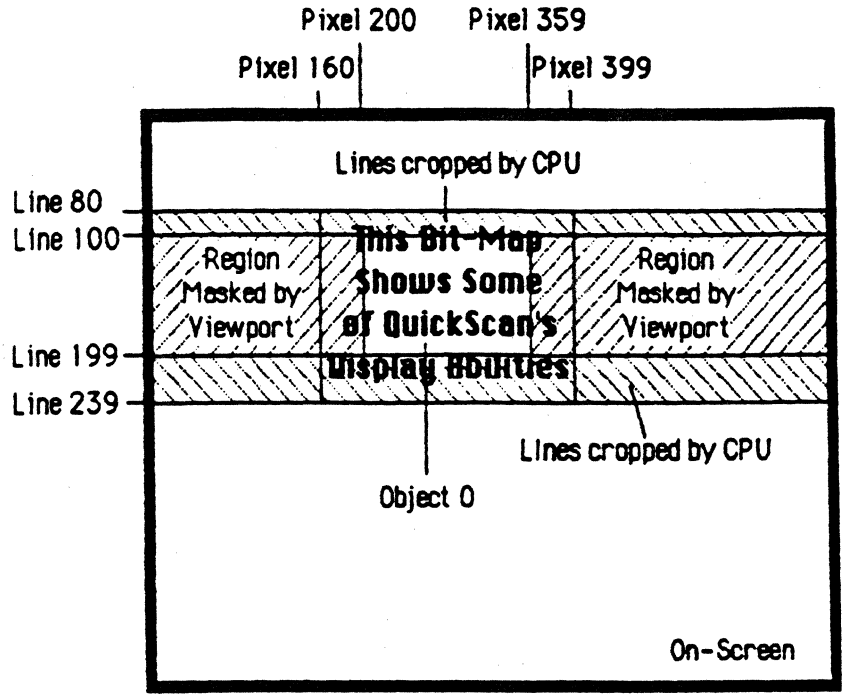


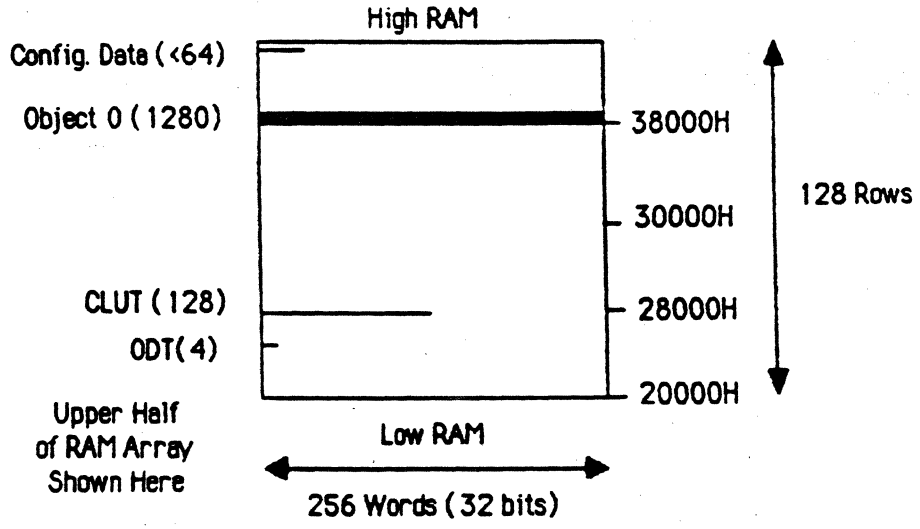
Note: RAM array proportions are realistic: one line (——) is one row thick.

Note that we cannot scroll to the left of the **absolute origin**, so if you anticipate large horizontal scrolls to the left, then you ought to position your **absolute origin** well to the left of the object.

7.1.6 Vertical Viewports

Consider the following diagram:





Note: RAM array proportions are realistic: one line (——) is one row thick.

What is shown here is an object which is masked vertically as well as horizontally. It has a vertical viewport as well as a horizontal one. Unlike horizontal viewports, however, QuickScan does not provide direct support: the vertical viewports must be generated by the 68020.

The way this is achieved is by the 68020 changing the **object description** so that it describes only the lines of the object that we wish QuickScan to display. That is to say, since our vertical viewport in the diagram above extends from line 100 to line 199, then our **object description** will only contain those lines of the object. Then, QuickScan simply will not display those lines "masked" by the viewport and we will get the desired effect.

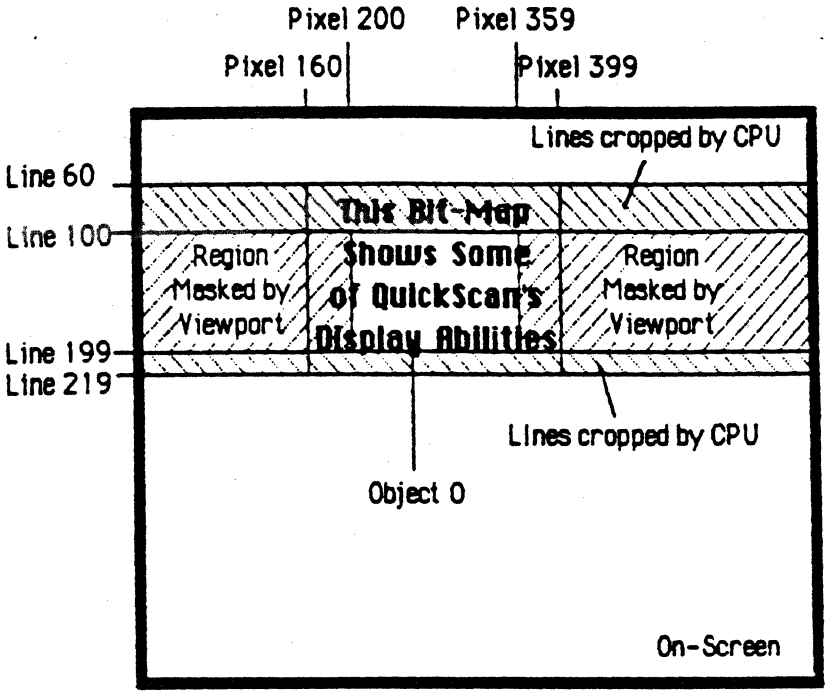
In this example, we see that the visible lines of the object are from its 20th line to its 119th line, since 20 lines from the top and 40 lines from the bottom are masked by the viewport. We start by changing the **start address** parameter to point to the **line description** for the 20th line, since this is where our new object will start. Then, we change the **start line** parameter to line 100, the first line in display space of the new object. And, finally we change the **object height** parameter to 99 to reflect the new height of the object. The result is the displayed region shown in the center of the diagram above.

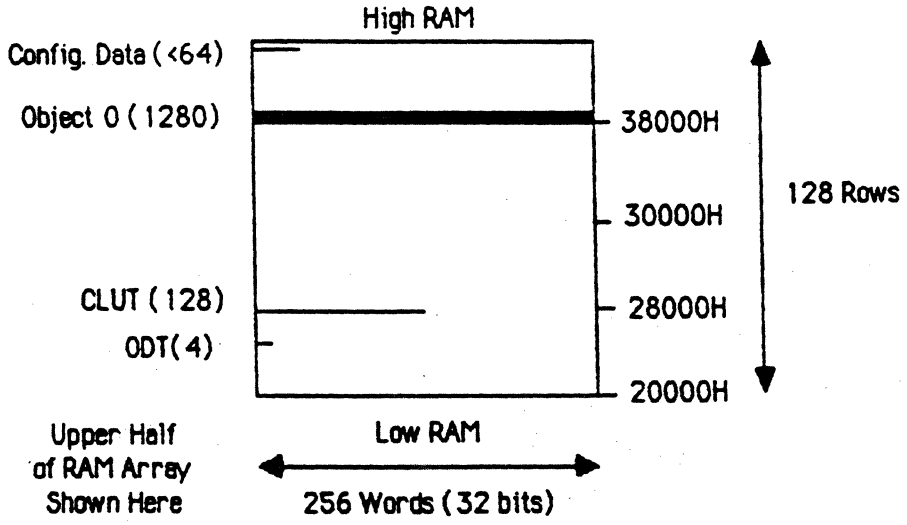
There a few fine points worthy of note. First of all we have the same problem of the small possibility of multiple parameter changes being partially complete when the ODT is loaded, and the resulting display having a minor aberration as we discused in section 7.1.4. Second of all, notice that we have not changed the RAM utilization of the object even though we are only using part of the **object description** . You could, of course, use this RAM for something else if you knew that the vertical viewport would never be changed and that the object would never be scrolled vertically. But, if this is not true, as you shall see in the next example, you ought to leave the rest of the **object description** intact. And, finally, notice that the CPU efficiency increases slightly with a vertical viewport, although the horizontal execution time remains the same. The CPU efficiency is a function of the lines of an object displayed, and with 60 less lines displayed we have consequently less CPU cycles stolen. The horizontal execution time is still the same because those lines which are displayed take the same amount of time to load as they did

before.

7.1.7 Vertical Scrolling

Just as the horizontal elevators in the Mac display caused horizontal scrolling, the vertical elevators cause vertical scrolling. The effect of a vertical scroll 20 lines up is shown here:



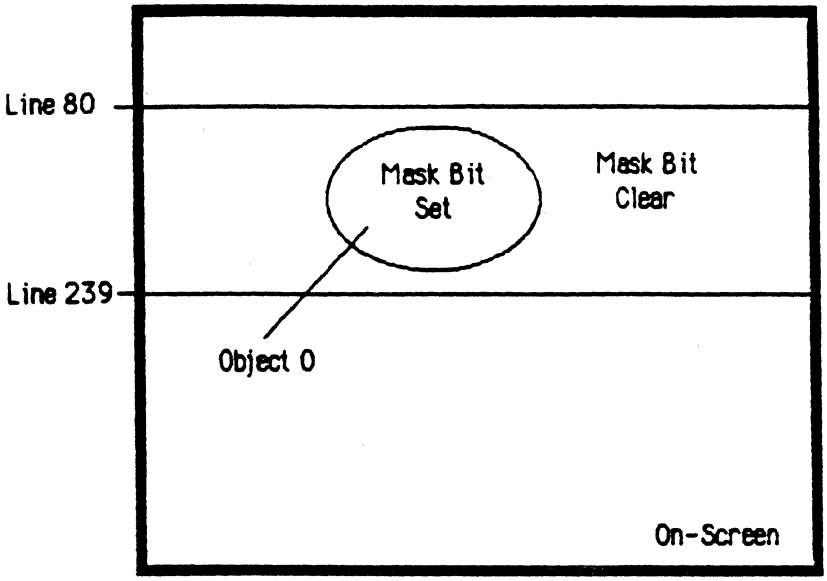


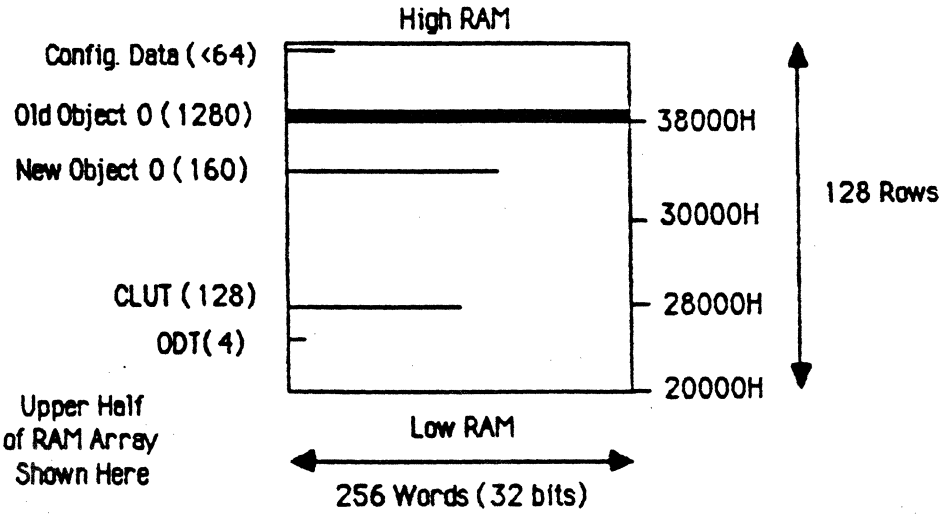
Note: RAM array proportions are realistic: one line (——) is onerow thick.

Just as a horizontal scroll entailed moving the object and holding the viewport constant, a vertical scroll entails the same procedure. So, we position the object vertically at the desired new position, starting at line 60. Then, we build a new vertical viewport just as we did before, except this one starts at the 40th line of the object and ends at the 199th line.

7.1.8. Arbitrarily Shaped Viewports

Consider the following diagram:





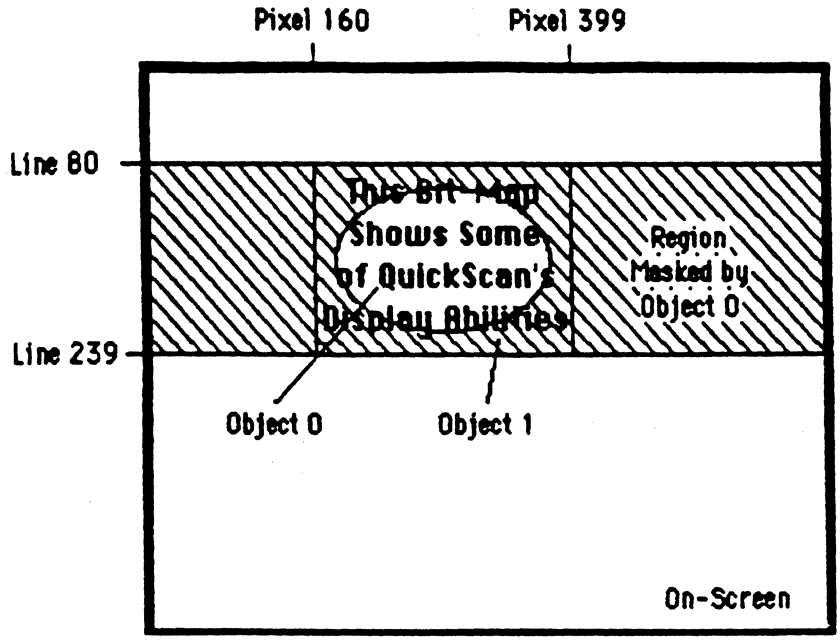
Note: RAM array proportions are realistic: one line (——) is onerow thick.

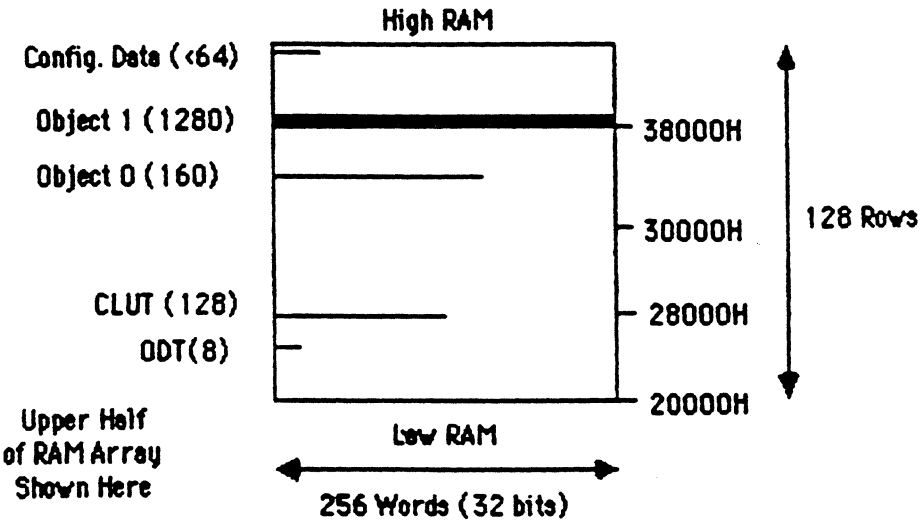
Sometimes we want a viewport which is not rectangular at all. For this application we have a mechanism for arbitrarily shaped viewports. The way it works is you define a 1 bit/pixel object that you wish to use as your mask. This object (which I shall call object 0) must be directly behind (i.e. at the next lower priority) than the object to which you wish to apply a viewport (which I shall call object 1). Then, you specify the **write mode** of object 0 to be M so that it writes to the **mask bit** of the **pixel storage cell**. Where you wish object 1 to be masked, write 0 to the **mask bit**, and where you wish it to show through, write 1. Then, in the **dispatch table entry** of the object 1, set its **viewport limit** to 0. This disables the automatic viewport mechanism from clobbering your custom viewport when object 1 is dispatched.

Object 0 was created in the following way: I used its automatic viewport to mask all pixels on the screen (see first paragraph on page 39). Then, I specified a single Run **instruction** on each line to clear the **mask bits** from the left to the right side of the ellipse for that line. Note that each line's run is different so I couldn't use the **first word** for the Run **instruction**, but rather specified a NOp for the **first word** and put the Run as the first (and only) word of each **line description** in RAM. For those lines above and below the ellipse, I specified a NOp for that word.

Thus, the **object description** requires 160 words, 1 word for each line in RAM. As the first object in each line, object 0 will be displatched with minimal **object dispatch overhead**. The Bit Map **instruction** takes 80ns to execute, and since object 0's **line description** in RAM is exactly 1 word long, object 1 will also be dispatched with minimal **object dispatch overhead** (see section 6.2) as well. So, the total execution time for the 2 objects is 320ns+80ns+320ns+640ns=1360ns.

The resulting display is shown below:





Note: RAM array proportions are realistic: one line (——) is one row thick.

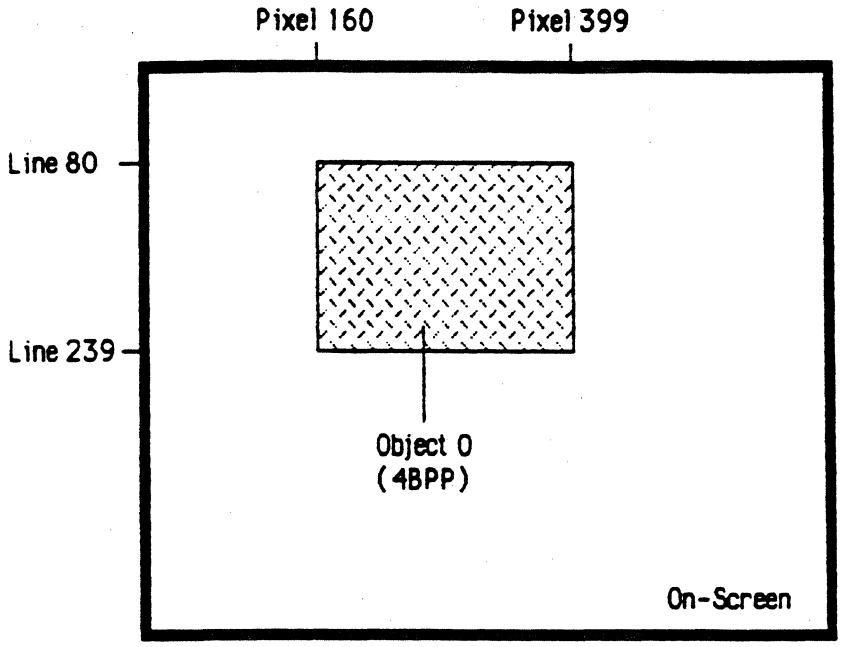
7.1.9. Embedded Masks

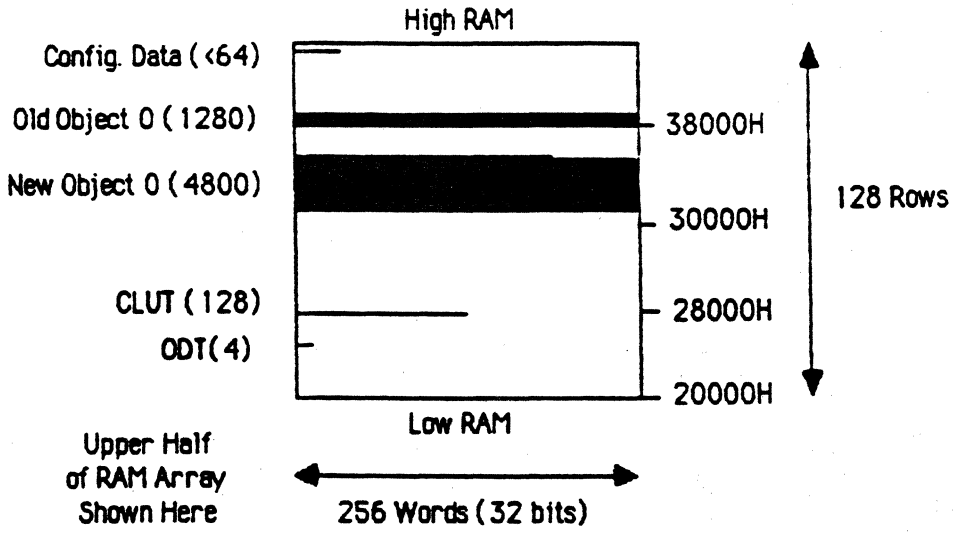
We might wish to overlay a background object with our text bit-map object and have the background show through between the letters. We could achieve this by loading down the background object, then by loading a custom mask object which corresponds to the text's pattern, and finally by loading the text object on top of the mask. But, there is a simpler way: **embedded masks**.

The text object in this example is a 1 bit/pixel bit-map, and it so

happens that if we were going to make a custom mask, we need a 1 bit/pixel bit-map with exactly the same pattern. Using this fact, we can combine the bit-map write and the masking operation with the same text bit-map and save ourselves an **object description**.

To see this, let's first make our background object. This object is 240 by 160 and 4 bits/pixel. It is shown below:





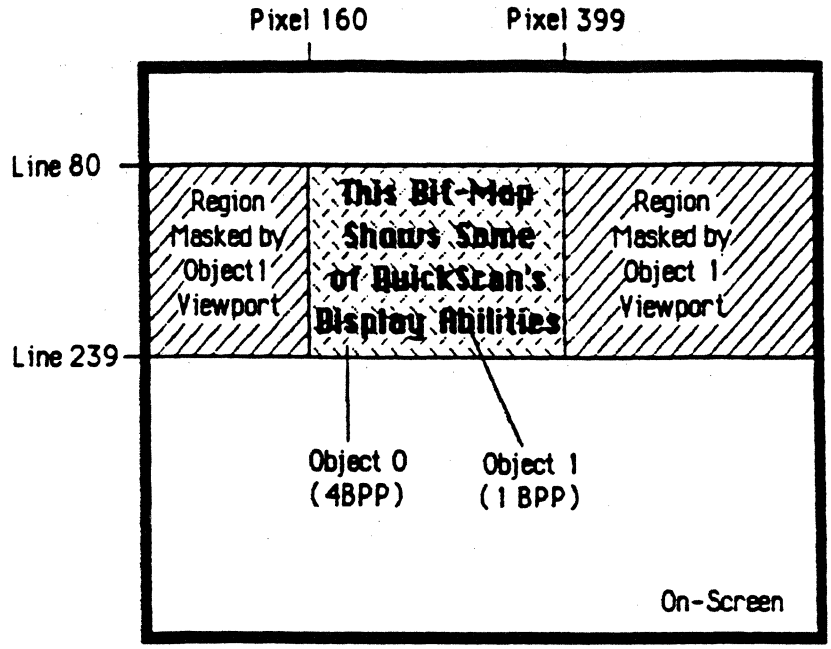
Note: RAM array proportions are realistic: one line (——) is one row thick.

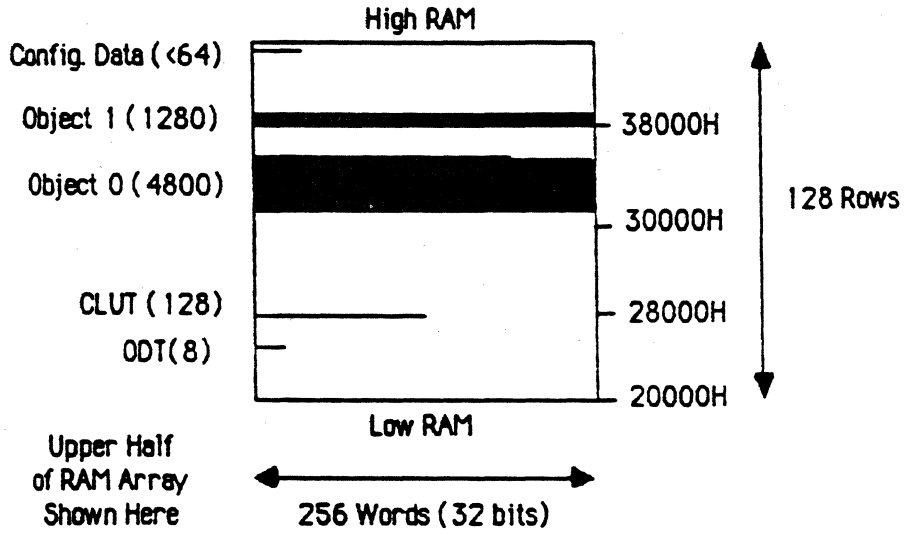
Notice that is has no horizontal mask. This is so because at 4 bits/pixel with a horizontal dimension of 240 we have exactly 30 words per line with no excess pixels. I've disabled the horizontal viewport for convenience. Notice also that we might like the 16 colors mapped by this bit map to be separate from the 2 colors of the text bit-map. To do this we need only change the lower byte of the **constant word** so that when it is combined with the 4 bits of the pixel data the resulting **index** ends up to point to a convenient place in the CLUT. This object shall be object 0.

Now, using the text bit-map from the previous examples, there is very little we have to do to activate the **embedded mask** function. First of all, we must make it so the white background masks (doesn't write) and the black letters don't mask (do write). This is determined by the **e_polarity** bit in the **dispatch table entry**. Let's say that black is 1 and white is 0, then we want 1 to permit writes, so we set **e_polarity** to 1 (see section 6.1.7). Next, we have to change the Bit Map **instruction** in the **first word** so that it is in **embedded mask** mode by setting the **e_mode** bit to 1. And, that's it.

Notice that the fact that we are using **embedded masks** does not obviate the need to have a horizontal viewport to mask off the excess bits of this object. This masking function works with the **mask bit** in the **pixel storage cell** and is independent of the **embedded mask** function. If either or both masks are inhibiting writes at a given pixel, then the write will be inhibited (see section 3.3).

Well, after all is said and done, the resulting display is shown below:





Note: RAM array proportions are realistic: one line (—) is onerow thick.

And, so this completes the section on rectangular bit-map applications. Using the examples shown here and the information in the preceding chapters, you should be able to set up your own rectangular bit-maps, customized for your own particular display needs.

7.2 Runs and Complex Objects

This section shows examples of special case objects whose object descriptions can be specified in ways which economize memory, time, or both. It is important that you understand that all objects shown in this section can be specified using the rectangular bit-maps discussed in the previous section with precisely that same resulting displayed image. But, these special case objects occur commonly enough and the savings are substantial enough that I feel it is worthwhile to give QuickScan special capabilities to support them. Note that the QuickScan mechanisms directly used here are indirectly used in generating rectangular bit-maps, so there is really no additional hardware cost directly attributable to supporting these objects.

All of the special case objects considered in this section are largely made up of runs, and I refer to such objects as *run-class* objects. The main capability that really makes considering run-class objects worthwhile is that of the *fully parallel* run. While a few display processors that I know of have supported runs (although none have yet made it to market), all of them implemented runs by iteratively writing the pixels that make up the run in a line buffer. That is to say, if you specified a run that was 400 pixels long, then the display processor would go and write 400 pixels, one after another, or at best would write the pixels in groups of 4, 8, or 16. QuickScan implements runs by having all pixels that make up the run written simultaneously to the line buffer. So, if 400 pixels are specified in a run, 400 pixels will be written at once. Or, in hardware terminology, we'd say that the runs are written fully in parallel.

The key advantage of the fully parallel run capability is in "getting the jump" on *spatial complexity*. To understand this concept we have to make our way through a little mathematics. You computer-types out there are familiar with use of the term *computational complexity* in regard to iterative algorithms like sorts and searches. We might say that the complexity factor identifies the facets, or *dimensions* of an algorithm so that we can compare the algorithm's efficiency with that of others. For example, an $O(n^2)$ (read "order n-squared") algorithm is less efficient than an O(n) ("order n") algorithm because we can expect for every n operands submitted to each algorithm the algorithm will go through n^2 iterations in the former case and n iterations in the latter.

Spatial complexity, as I use it here, is an analogous concept which identifies the dimensions of an object in regard to the amount of memory necessary to represent the object per the object's size. Thus, an object of O(n) spatial complexity would require twice as much memory for its representation if it were made twice as large, but an object of $O(n^2)$ complexity would require 4 times as much memory for the same doubling of size. Consider the following example objects: A point is O(1), a simple line is O(n), and a bit-map is $O(n^2)$. We can derive these numbers analogously to deriving computational complexity numbers: by changing the size of each object (like changing the number of operands submitted to the algorithm), and seeing how much memory it takes to represent the object, proportional to the change in its size (like seeing the number of iterations of the algorithm, proportional to the change in the number of operands).

A point is represented by 1 pixel, and as it has no dimensions, scaling it by a scale factor n still results in the same size of 1 pixel. So, the memory representation increases proportional to n^0. A point is O(1).

If we have a minimum width line x pixels in length, it can be represented by approximately x pixels. If we scale the line by scale factor n, then it will now be about nx pixels long. So, the memory representation increases proportional to n^1. A line is O(n).

If we have a rectangular bit-map h by v (horizontal by vertical) pixels in size, it can be represented by h*v pixels. If we scale the bit-map by scale factor n, then it will now be about nh*nv pixels in size. So, the memory representation increases proportional to n^2. A rectangular bit-map is $O(n^2)$. Coming to the same conclusion about non-rectangular bit-maps is a little more tricky, but I'm sure you can see intuitively that the result is the same.

Consequently, given two objects of the same size, one represented by lines, and the other represented by bit-maps (e.g. a 3-D wire-frame model vs. a 3-D solid model), we can expect that as we increase the size of the objects, the memory required to represent the line object will increase linearly, and the memory required to represent the bit-map object will increase exponentially. For small objects, the exponential growth is not that different from the linear growth, especially considering we normally have several lines to symbolize the region represented by a single bit map, and there is a fixed overhead for each line in any practical implementation.

But, for large objects, the exponential growth far outpaces the linear growth, and we soon find ourselves needing huge amounts of memory to represent such large bit-maps (thank goodness it's only n-squared!).

Then, forgetting the cost of all of the memory to hold large bit-maps, consider the overhead in manipulating such large bit-maps. Whether the objects are software-based or hardware-based, the exponential growth quickly outpaces our hardware, and we find that interactivity is shot to hell. Notice how you don't move Mac windows, you move their *outlines*. The Mac operating system deals with the exponential explosion by dropping the window object from an $O(n^2)$ bit-map representation to an $O(n)$ line representation when you need interactivity in its manipulation. The manipulation complete, it redraws and gives you back the $O(n^2)$ bit-map representation required for the object to be visually useful.

Anyone who has worked with interactive animation systems is cognizant of the property of "inertia" associated with lugging around large bit-maps. Notice that any people who do commercial 3-D animation (like Pacific Data) always run through sequences with "wire-frames" to get the motion right, then render the final solid objects off-line, letting their computer munge away, computing the big bit-maps. They can manage the $O(n)$ complexity of lines (less the light models, too) in real time, but not the $O(n^2)$ complexity of bit-maps. Notice that video games with bit-map objects either have a very few, simple, large bit-maps (Pole Position, Karate Champ), or have lots of little "sprite"-sized bit maps (Galaxian, Defender, Dig Dug). They may have many large, complex objects made of lines (Star Wars, Battlezone), but you never see a video game with many large, complex bit-maps; there is just no way to handle them in real time.

Just as we endeavor to reduce computational complexity in algorithm development so as to increase program execution speed, we endeavor to reduce spatial complexity in object representation so as to increase object manipulation speed. As we have seen in the video game world, this applies not just to software manipulations, but also to display processor manipulations. And, as fast as QuickScan runs, it too can be brought to its knees by large $O(n^2)$ object representations. It is especially vulnerable to objects with large horizontal dimensions, since its fundamental speed limitation is how many pixels for one line it can load in one line's time. For many bit-map objects, there is simply nothing that can be done - we have to face the fact that they are $O(n^2)$ and live with it. But, wouldn't it

be nice if we could find a useful class of bit-map objects that could be represented by some lower order of spatial complexity...

Well, there just so happens to be a large class of useful bit-map objects which can be represented largely or entirely by runs. These are the **run-class** objects which I introduced at the beginning of this section, and as we shall see, they are O(n). Objects of this class are formally characterized by having a low frequency of horizontal color modulation relative to their horizontal size, which means the number of pixels in each horizontal line is much greater than the number of color changes. Objects which fall into this class include backgrounds, cartoons, bar and pie charts, certain types of 3-D models, certain CAD/CAM objects, and many others. These objects can be specified efficiently in terms of a few horizontal runs because only 1 run is needed per color change and thus the number of runs in a line is much smaller than the number of pixels in a line.

So, a **run-class** object can be represented in memory by about r*v runs (r is the average number of runs per line, v is the number of lines). If we scale the object by a scale factor n, then we find that representation in memory has changed to r*nv, because *at any scale the object has the same number of runs horizontally*, but the scaling factor increases the number of lines. Thus, the memory representation increases proportional to n^1, and the object is O(n).

Virtually every graphics display system I have seen (including the SGI IRIS) ultimately treats **run-class** objects as $O(n^2)$ bit-maps, solving the exponential complexity explosion by throwing fast, expensive processing muscle at it. Even if they store the objects in terms of runs and therefore enjoy O(n) complexity in their memory consumption, they iteratively write out each pixel of each run to a line buffer, effectively expanding the object back into an $O(n^2)$ bit-map as far as manipulation speed goes. QuickScan solves the problem with brains rather than brawn and instead of writing out the h individual pixels of a run iteratively at high speeds it simply writes all h pixels at once at a reasonable speed, maintaining O(n) complexity both in memory consumption, and in manipulation overhead.

The result is that as objects get larger, the processing of the objects increases linearly with QuickScan, whereas with everyone else's graphics display system the processing increases exponentially. Furthermore,

QuickScan's linear growth is limited to the vertical dimension where it has plenty of time, whereas their growth is in both the horizontal and vertical dimensions. So, all else being equal, anything that they can do with wire-frame models, we can do with solid models (they're O(n), we're O(n)). If they can manipulate one run-class object in real-time that is n by n, we can manipulate many (approaching n) such objects at once (they're $O(n^2)$, we're $O(n^2)$).

Now, I'm sure you can appreciate that we are gaining a phenomenal advantage over conventional graphics displays by having fully parallel runs. When it comes to run-class objects, THE OTHERS CANNOT KEEP UP. I don't care if they have a CRAY-X/MP hooked up to an ultrafast frame buffer. We have 640 processing elements working at once. They have one. Current technology cannot iteratively write 640 pixels as fast as the 80ns it takes us to parallel write one 640-pixel run, at any cost. In fact, it doesn't even come close. We have the opportunity here to chart new territory in real time computer graphics - and we're talking about a consumer product! Just think about the awesome interactive applications that can come out of this capability. It really blows me away.

Of course, this capability does not directly help us in speeding up non-run-class bit-map objects, but remember, QuickScan still is an extraordinarily fast bit-map display processor. Its efficient handling of run-class objects augments this bit-map capability at the programmer's discretion, and indeed, an individual object can very well be part runs and part bit-maps, and still close enough to a pure run object to be interactively manipulated. (Such objects are called *complex objects*, and I show examples of such objects in the forthcoming subsections.)

So, as you read the following subsections and consider the worth of the parallel run capability, remember: this is really new technology in computer graphics. Everyone's been talking about applying large-scale parallelism to computer graphics for years - it's the only direction left for more speed - but no one's ever been able to do it in a commercial product. If this thing flies, we'll be leading the way into a new era.

7.2.1 Backgrounds

One application area in which runs immediately show their worth is that of the generation of backgrounds. Backgrounds that are all of one color that would otherwise be represented by a large 1 bit/pixel bit-map,

now can be drawn with a single run per each line. Large backgrounds with static objects (e.g. trees, mountains, clouds, sky) can be specified with a handful of runs per each line, requiring orders of magnitude less memory and line buffer write time than a comparable bit-map representation. In fact, backgrounds even larger than the screen can be efficiently stored and manipulated to give the illusion of the screen being a viewport into another world.

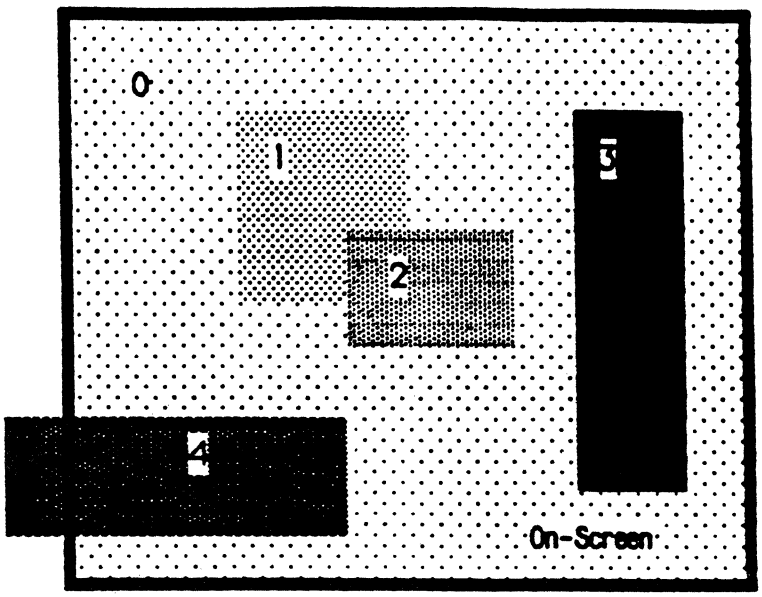
QuickScan is particularly optimized to generate rectangular, single-color backgrounds. It can generate such backgrounds without using any RAM, without stealing any CPU cycles, and taking only 320ns to execute for each line of the background, regardless of the background's size. (Indeed, this type of background is handled so efficiently, that it actually qualifies to be of O(1) complexity in memory consumption.) The way we specify such a background is very straightforward:

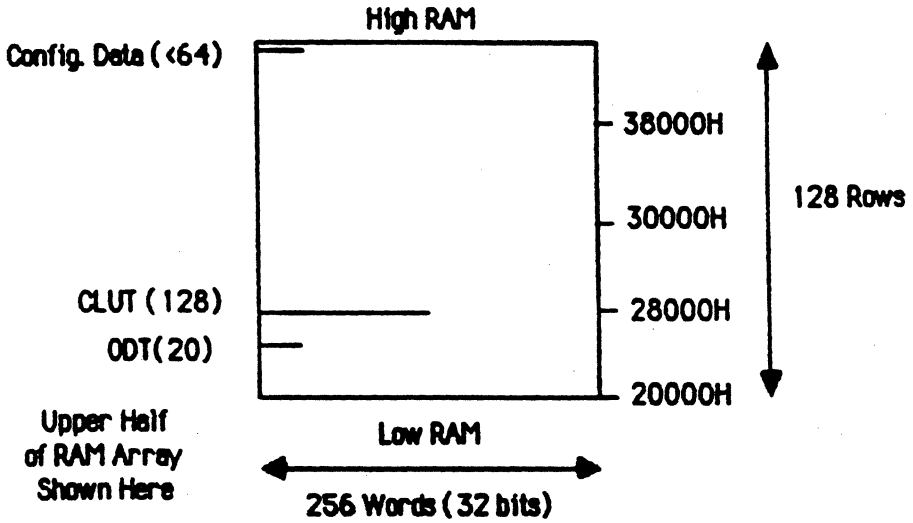
You make a **dispatch table entry** at the priority at which you want the background.

Load **start line** with the first line of background; **object height** with its height-1; **absolute origin** to the background's left border; **bus_access** to 0; **viewport origin** and **limit** both to 0; **constant word** and **display mode** as you wish; and **start address**, **e_polarity**, **line_mode**, and **line_length** to any value.

Load the **first word** with a Run **instruction**, setting **r_origin** to 0; **r_limit** to the horizontal dimension of the background; **end_line** to 1; and **data_7**, **w_mode**, and **d_align** as you wish.

And that's it. On each line of the object, the one Run **instruction** in the **first word** will execute, generating a run from the left side of the background to the right, and that's it. You can choose the color and the display mode. Since it's a no **bus_access** object, you are guaranteed that its **dispatch overhead** is minimal (see section 6.2). An example of 5 such backgrounds is shown below (each pattern represents a single color). Note that there is no space in RAM allocated to each object at all, except of course, for the 4 words in the **dispatch table entry**.



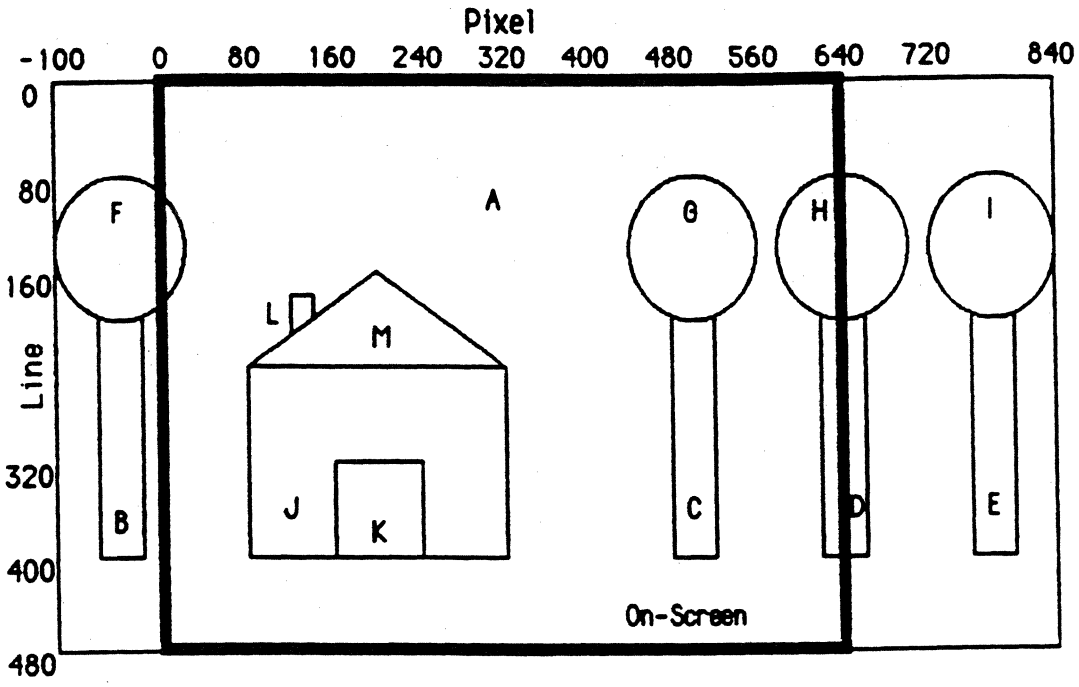


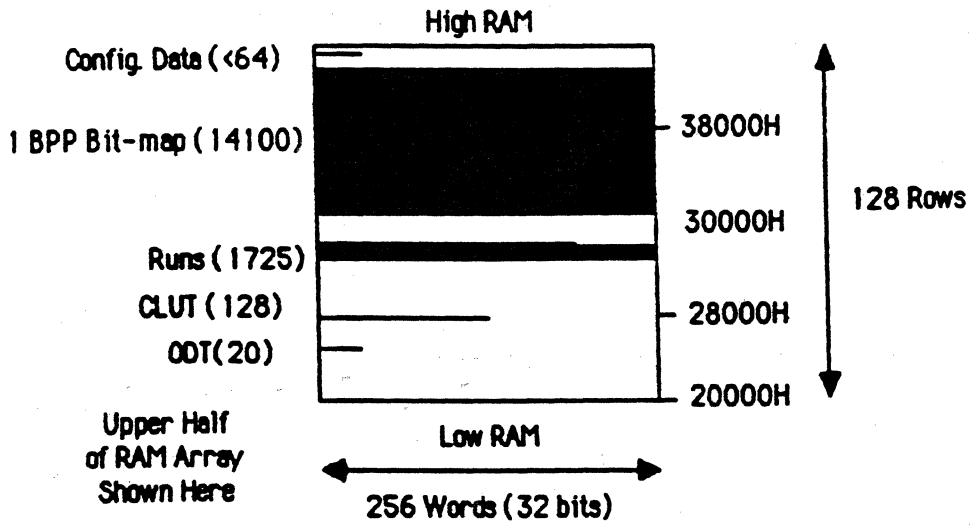
Note: RAM array proportions are realistic: one line (——) is one row thick.

Generating backgrounds more complex than just a single color, however, is a little more involved. Since we would then have various shapes in the background, we couldn't rely on each line having just the same single run. Indeed, we couldn't even guarantee that each line would have the same number of runs!

Such background objects are usually made up of a collection of

individual primitive objects. These primitive objects are called *subobjects* because each could be an object in its own right. An object which contains 2 or more subobjects is called a *complex object*, and a complex object's object description is made up of the union of its subobjects' object descriptions. A complex object (a forest scene) is shown below, with each subobject identified with a letter:





Note: RAM array proportions are realistic: one line (—) is one row thick.

A subobject may be made up of bit-maps, runs, or both, and there may be any number of subobjects in an object. In the forest scene complex object shown in the above diagram, there are 13 subobjects, each a solid region of one color represented by runs. Subobjects may also overlap, and in fact, in the above diagram subobject A is a simple rectangle - the complex region which we see in the diagram for subobject A results from the overlaps of the subobjects in front of A.

Note that partitioning a complex object into subobjects serves us only as a conceptual tool to help us find a way to represent an object efficiently. QuickScan understands an object only in terms of what it is told to display by its dispatch table entry and line descriptions; it is unaware of how the object has been partitioned. Thus, the criterion we use to partition an object into subobjects is completely arbitrary, and we can define this criterion however it is convenient. The criterion I used in this example was to isolate a subobject wherever there was an individual region of color, but it could just as well have been to isolate the house as a single subobject and each tree as a single subobject. As we shall see in a moment, my choice was informed and by choosing the former criterion I saved a little more memory than I would have with the latter. But, there may very well be an even more efficient criterion to partition this complex object that I haven't thought of.

To generate the object description for the forest scene, we first order each of the subobjects by *subpriority*, background to foreground. Subpriority is to an object as priority is to display space: it indicates who is in front of who. I assigned a letter to each of the subobjects in order of its priority such that A is the background-most and M is the foreground-most.

Next, for each subobject we generate an object description, its line descriptions referenced to the single absolute origin of the complex object. Since the left border of the complex object is at pixel -100, we might as well set its absolute origin to -100. And, since each subobject in this complex object is a contiguous region of a one color, each subobject line description is a single Run instruction.

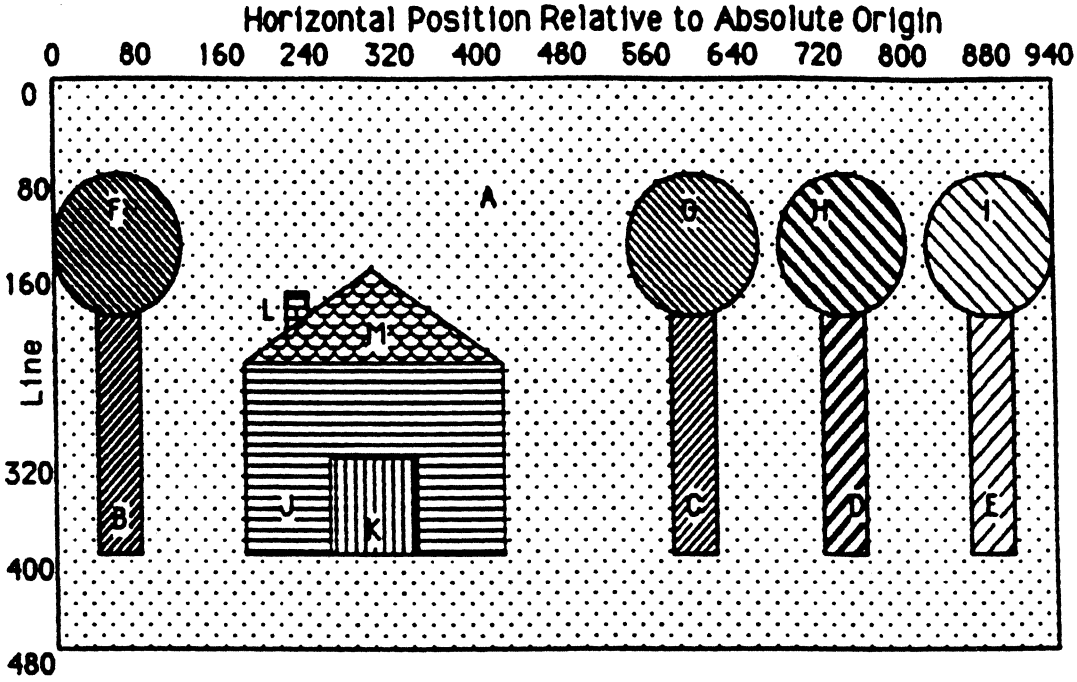
Subobjects A, B, C, D, E, J, K, and L are all rectangles, so for each one's line descriptions, we specify the same Run instruction (starting

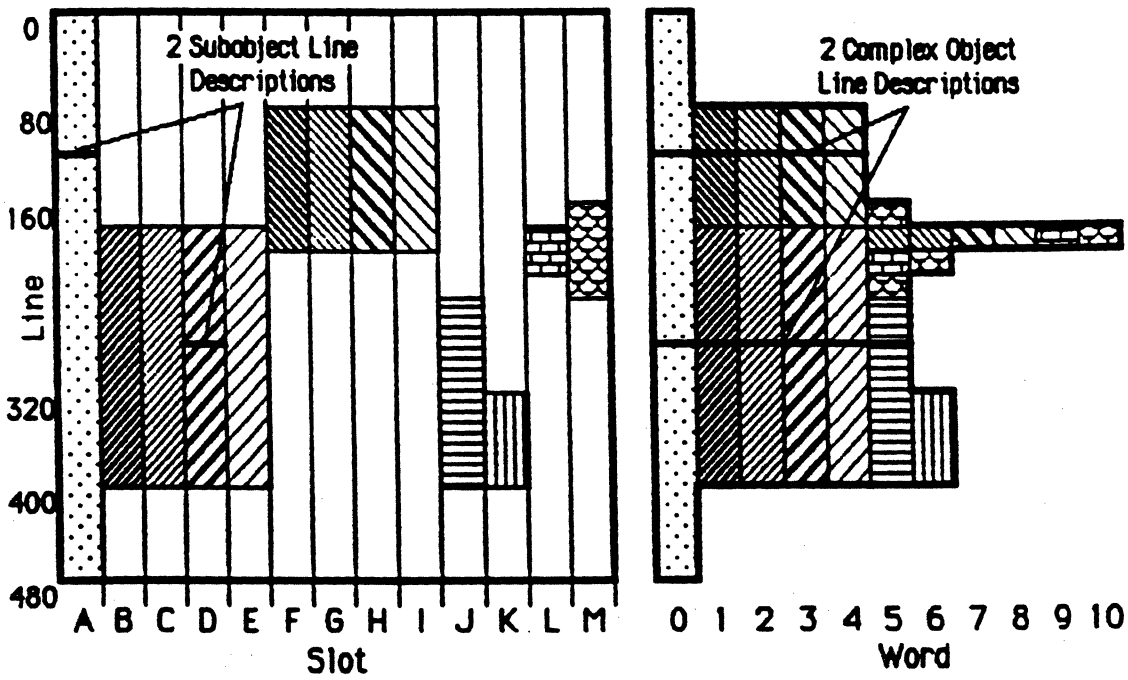
at the rectangle's left edge and ending at its right edge). For example, **Subobject** B is 40 pixels wide, 220 lines tall, and has its left edge at pixel -60. It is described by 220 Run **instructions** , each with the **relative origin** set to 40 (-60 - (-100)) and the **relative limit** set to 80 (-21 - (-100) + 1).

Subobjects F, G, H, and I are all circles, so these take a little more work to describe. We first observe that a circle is vertically symmetric across its center, so when we figure out the set of runs for the **line descriptions** of the top half of the circle, we need only reverse the order of the set to get the **line descriptions** for the bottom half. To determine the top half's set of runs, you can figure out the left and right edge of the circle on each line by using some simple geometry, and then set up a Run **instruction** for each line with the **relative origin** at the left edge and the **relative limit** at the right. So, quite unlike the Run **instructions** for the rectangular **subobjects** , all of the Run **instructions** in each half of the circle have different **relative origins** and different **relative limits** , and must be computed individually for each line.

Subobject M is a triangle, and as with the circle **subobjects** , you need to apply a little geometry to determine the left and right edges of each line, then use that information for the **relative origin** and **relative limit** of the Run **instructions** for its **line descriptions** .

Now, to assemble these various **subobject's object descriptions** into the one **complex object's object description** for the entire forest scene, we have to interleave the various **subobject line descriptions** , line-by-line, with the lowest **subpriority subobject's line description** on each line first, and the highest **subpriority subobject's line description** last. This may be a little difficult envision, so on the next page you'll find a diagram which shows the interleaving process in two steps. Above, you'll see the forest scene, this time with each **subobject** identified with a pattern. Then, on the lower left, you'll see a diagram of all of the **subobjects'** individual **object descriptions** , interleaved with each **object description** restricted to a slot corresponding to the **subobject's subpriority** . To see how this works, compare the 480 lines of this diagram to the 480 lines of the forest scene. Notice that the vertical size and position of the patterned bar representing the **object description** for each **subobject** corresponds with the vertical size and position of the **subobject** itself in the forest scene.





Subobject Object Descriptions (Slotted)

Subobject Object Descriptions (Packed)

This is because the **object description** of each **subobject** only exists on those lines where the **subobject** exists. Thus, each line of a slot (see

line numbering to the left) holds the **line description** corresponding to the same line of the slot's **subobject** in the forest scene (two sample **subobject line descriptions** are highlighted in the diagram).

Since each slot corresponds to a **subpriority** level, the **line descriptions** on each line are in proper order for interleaving, left to right, into a **line description** of the **complex object** - you just have to eliminate the empty slots. The diagram on the lower right shows what we get when we eliminate these empty slots, and pack everything to the left (you can use a straight edge held horizontally to compare the two diagrams). This is an exact representation of how the interleaved **subobject line descriptions** make up the **line descriptions** for the **complex object**. If you scan left to right across a given line in this diagram, the **subobject line descriptions** you'll cross will exactly make up, in that order, the **line description** for the **complex object** for that line of the forest scene (2 sample **complex object line descriptions** are highlighted in the diagram). And, if you put all of the **complex object line descriptions** from line 0 to line 479 one after another in RAM, you'll have the **object description** for the full **complex object**. Thus, we have the **complex object's object description** formed from the union of the **subobjects' object descriptions**.

For example, at line 0, the **line description** of the **complex object** is made up of just **subobject** A's **line description** since no other **subobjects** are on that line, but about 80 lines down, we find that the **complex object's line description** is made up of **subobject** A's **line description** followed by F's, G's, H's, and I's. At around line 160 the **complex object's line description** gets very long, being made up of every **subobject's line description** except for J's and K's. Then by line 479, once again the only **subobject** on the line is A and the **complex object's line description** is just made up of A's **line description**.

Since each **subobject line description** in the forest scene is just a single Run **instruction** (which is a **single-word instruction**), the width of each of the patterned bars in the diagram on the right is one word. If you count the number of bars horizontally on any line, you'll find out the number of **instructions**, hence the number of words, for the **complex object's line description** on that line. Notice that, unlike the previous examples I've shown you, these **line descriptions** are of variable length, and **variable length line mode** must be selected in the

dispatch table entry for the complex object.

QuickScan has no way of determining by itself where each line description ends, so the last instruction on each line of the complex object must have its end_line bit set to let QuickScan know. Of course, the last instruction on each line may belong to any subobject, so you must inspect the object description line-by-line and set the end_line bit in whichever subobject's Run instruction it is appropriate. Note that end_line bit is not set at the end of each individual subobject's line description unless it happens to be the last subobject line description in the resulting complex object's line description. The end_line bit is the only way QuickScan has of finding the divisions between line descriptions in an object description, so it is truly unaware of subobject line description interleaving, or as I stated before, the way we choose to partition an object into subobjects is purely a conceptual tool for us humans, and QuickScan is unaware of it.

Notice that subpriority is handled by QuickScan very simply by just overwriting as each subobject line description is loaded into the line buffer. The lower subpriority subobjects are written to the line buffer first (since they are first in the complex object line descriptions), and they are overwritten by the higher subpriority subobjects that overlap them. So, in the forest scene we get a concave curve at the top of the tree trunks and an angle at the base of the chimney even though each of these subobjects is a simple rectangle. And although our very background subobject, A, appears to be of an extremely complex shape, it also is just a simple rectangle. Since the subobjects are specified by runs, it costs us nothing to waste the portion of a subobject that is covered up by another subobject, so we might as well describe these background-most subobjects in whatever way is convenient.

QuickScan object descriptions have the first word of each line description stored in common for all lines of the object in the dispatch table entry. So, if every line description of an object description starts with the same instruction command word, then we can put that word in the first word and thereby avoid having to store it individually in RAM for every line of the object description. Can we use this feature with our forest scene complex object? Well, looking at the packed diagram and the forest scene above, we see that on every line, the first word is indeed the same: it is the single word of a subobject A's Run

instruction ! On every line of the complex object we have subobject A generated by a Run instruction with its relative origin at 0 and its relative limit at 940. So, by putting this instruction in the first word, we can directly shave 480 words (1 word for each line) off the complex object's RAM consumption. Although this may seem coincidental, it really isn't. Complex objects commonly have a rectangular "backdrop" upon which all of the foreground detail is overlapped. This characteristic is one of the motivating influences in QuickScan's design for the inclusion of the first word as a parameter in the dispatch table entry.

So, we've "put away" subobject A, but how much RAM will the rest of this object consume? The figure is listed in the initial forest scene diagram's memory map, 6900 bytes. Not too bad for a 13 color object that is 940 by 480! For comparison's sake the RAM comsumption of an equivalently sized 1 bit/pixel bit-map is shown in the memory map as well. Despite this bit-map's consumption of 56.4K bytes, we only get 2 colors to choose from! If we wanted to have a bit-map with all 13 colors, then we'd need 225.6K bytes. Note also, that we could, with the same 6900 bytes of memory consumed, have each run of a subobject have a different color. So, you might put a horizon in subobject A, and perhaps stripes on the wall and roof of the house (subobjects J and M) to make it look like a cabin. Then, we'd have well over 16 colors in the complex object and the cost of an equivalent bit-map would be 451.2K Bytes. I think you get the point.

Let's now consider the execution time for each line of the forest scene. Since the forest scene would probably be the lowest priority object displayed (since it is a background), then its dispatch overhead will be minimal, 320ns. Since the lines are variable length, some will take longer than others to execute. To get a worst case figure, let's look at the longest lines in the object, those around line 160, which each have 11 Run instructions . Now, since subobject A's Run instruction is in the first word, its execution time is included in the dispatch overhead. For the other 10 Run instructions , the execution time is 80ns apiece, for a total of 800ns. Thus, the worst-case execution time for any line of the forest scene is 320ns + 800ns = 1120ns. Now, it's not quite fair to leave it at that because if you look in section 6.2 on page 41, you'll find that we add 240ns to the dispatch overhead of the object next higher in priority (unless its a no bus_access object) because our last

instruction on every line of this object is a Run instruction . It's really fair to consider this extra dispatch overhead as part of our execution time, so then our worst case execution time is really 1120ns + 240ns = 1360ns, with the next higher priority object dispatching with no overhead. Also, because this is a variable length object, we have to be very careful in planning its placement in RAM if we want to ensure that no line description crosses a row boundary. If we don't plan for this case, then we should also include in the worst case the execution time added in case of a row boundary crossing, 560ns (see section 6.3). So, now our absolute, most horribly worst case execution time is 1360ns + 560ns = 1920ns. Since we have about 32 µsec total in each line, that means this forest scene, in the very worst case, takes up 6% of the available time for writing line descriptions to the line buffer. If we guarantee there are no row crossings, then it is 4.25%.

Bear in mind that only a few of the lines of the forest scene take anywhere this amount of time to execute since most are very short. Notice also, that if added more subobjects to the forest scene to make the picture more detailed and interesting, the execution time would not increase by much. This is because most of the execution time for this particular object comes from the fixed overhead of dispatching and row crossing, a penalty we pay once. Of the 1920ns listed for the very worst case, 320ns+240ns+560ns = 1120ns, is fixed overhead. If, for example, our worst case line of the forest scene had 22 runs instead of 11, our very worst case execution time would only increase to 1920ns+(22-11)*80ns = 2800ns, or 8.75% instead of 6% of the total line time.

Now, let's take a look at the CPU cycles stolen. We steal 1.4 CPU cycles for each line of the object, or 480*1.4 = 672 cycles. Let's assume that some line descriptions are going to cross row boundaries. Since the object description is 6900 bytes long, it is 6900÷4 = 1725 words long. There are 256 words in a row, so there are 1725÷256 = 6.7 rows in the object description . In the very worst case, it will cross 7 row boundaries total, resulting in 7*1.4 = 10 CPU cycles. So, in very worst case, the forest scene will steal a total of 672+10 = 682 CPU cycles total. Now, figuring that with the fixed overhead of 1578 cycles (see section 7.1.1) out of 59286 total available cycles, and we have the CPU running with 59286-1578-682=57026 CPU cycles or at about 96% efficiency (compared to 97% ideal efficiency). There's virtually no loss in CPU performance attributable to this object.

We I first introduced the concept of **complex objects** I stated that it was more efficient to partition the forest scene into its color regions instead of into its conceptual objects, the house and the trees. This comes from the fact that we are using **subobject** overlaps to make complex regions. For example, if wanted to represent the front wall and door of the house without overlaps, then we'd need to specify a run for the wall to the left of the door, a run for the door, and a run for the wall to the right of the door. Using overlaps, we specify a run for the wall, and then a run for the door on top of the wall. We get our left wall-door-right wall at the cost of 2 runs instead of 3. Similarly, with the lolly-pop trees, the ball on top makes a small concavity into the rectangle trunk. With overlaps we just need 1 run for the trunk and 1 run for the ball over the trunk. Without overlaps we need a little run to the left of the ball, a run for the ball, then a little run to the right of the ball. Thus, by partitioning this object into its color regions instead of its conceptual objects we save memory and execution time.

While this partition criterion works well for this particular **complex object**, it may not work as well for other **complex objects**. You just have to look closely at what you want to display, and then try a few ways of partitioning it. Just like anything else, with a little practice you can get to the point where you can eyeball it and immediately know how to deal with it. I am confident that we can make a paint program for an authoring system which automatically generates reasonably optimal **complex object** partition criteria, so QuickScan users don't have to concern themselves overly much with this sort of decision making.

The previous examples have all used the Run **instruction** for generating runs. If you flip back to section 5.2, you'll notice that there are two other run generating **instructions**, Screen Run and Sequential Runs. Screen Run is independent from object dependencies in that it always generates a run from the left edge of the screen to the right edge, regardless of the current **absolute origin**. This function plays a crucial role in QuickScan's internal control functions, but for the most part is not very useful from the user's point of view. Sequential Runs, however, is extremely useful in **complex objects** where there are several adjacent regions of color in a line. In a suitable **complex object** it uses about half the amount of memory as an equivalent number of Run **instructions**, and it plays a central role in generating the objects in the next sections.

7.2.2. Masks

Runs find another application in the generation of large masks. One of the most common uses of masks is for the purpose of implementing viewports. Inside the viewport the **mask bit** of all **pixel storage cells** is set to 1, thereby allowing them to be written by the forthcoming object, and outside the viewport the **mask bit** of all **pixel storage cells** is set to 0, thereby inhibiting them from being written by the forthcoming object. Viewports can be very large, in theory even larger than the entire screen, and it is very expensive, in RAM and in time, to generate them with even a 1 bit/pixel bit-map. But, is an $O(n^2)$ bit-map necessary? If you think about it, viewports are large contiguous areas of "color," the color being the **mask bit** state. They qualify beautifully as **run-class** objects, and can be generated in O(n) with runs.

In fact, the QuickScan automatic horizontal viewport mechanism works in just this way. If you specify a horizontal viewport for an object in its **dispatch table entry** (see sections 6.1.6 and 7.1.4), the way QuickScan actually generates the viewport is as follows: First, it generates a Run Screen **instruction** , clearing the **mask bit** of every **pixel storage cell** in the **line buffer**. Second, it generates a Run **instruction** , using the **viewport origin** for its **relative origin** and the **viewport limit** for its **relative limit** , setting the **mask bit** of every **pixel storage cell** between the **viewport origin** and the **viewport limit**. Because of the parallel run capability, this mechanism is guaranteed to take exactly 160ns to execute, regardless of the size of the horizontal viewport.

Arbitrarily shaped viewports can be easily specified as well, and an example of one is given in section 7.1.8.

There are also applications where a **complex object** needs a transparent region within it for which runs can be used to generate a mask. If, for example, you wanted to display a large wheel-shaped space station, you might want to use runs to mask out the regions between the spokes of the wheel so that when you draw the space station with bit-maps, you won't cover up these openings.

7.2.3. Cartoons

Representing cartoons efficiently and animating them in real-time is perhaps the most exciting application for fully parallel runs. Since

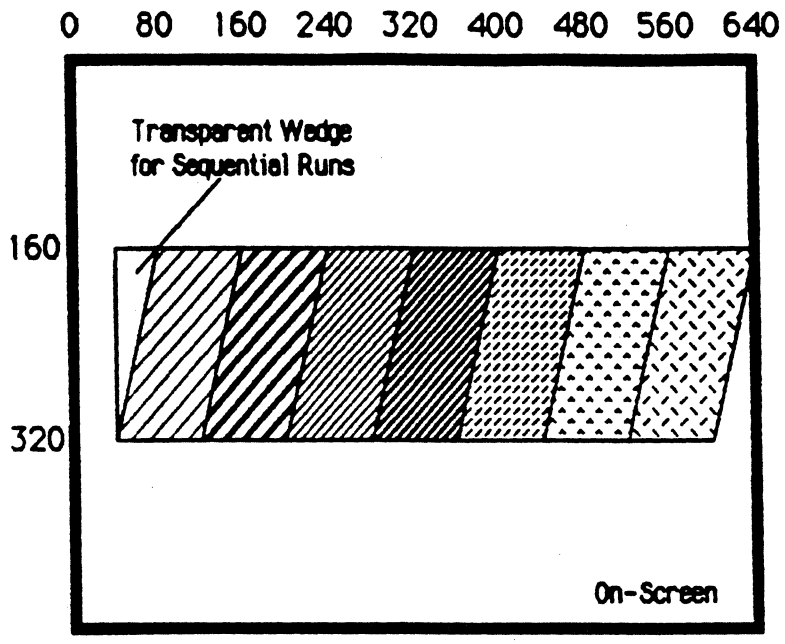
cartoon characters are made up of large, contiguous regions of color they usually meet the criteria for run-class objects, and compress readily from $O(n^2)$ bit-maps into O(n) run-generated objects. As we can store these cartoons characters efficiently, we can store many frames of each character at once, and then, by switching between these frames rapidly, we can get animation. In fact by storing a great deal of frames we can actually store enough for a number of possible animation sequences, thereby allowing *interactive animation*, so the user of the system can control a cartoon character like a puppet.

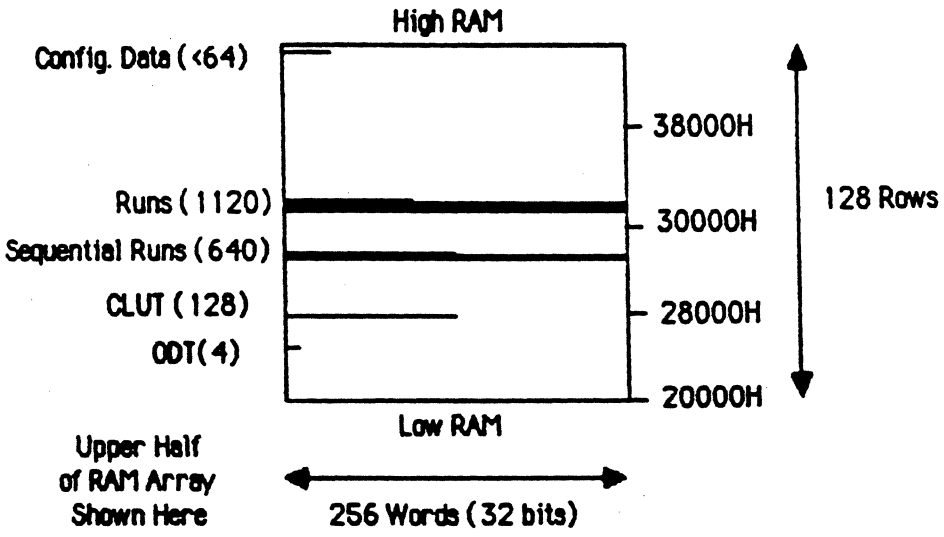
Now, these ideas are not new; people have had dreams of animation machines since the dawn of computer graphics. Indeed, you can find scaled-down versions of these ideas implemented currently in video game and home computers. These systems are just too simple and too crude to be interesting or very useful, and as such, they have not received much notice.

Because of QuickScan's speed and its fully parallel runs, it is able to animate several Disney-quality cartoon objects in real-time. And, with reasonable data compression in the CD ROM, we can supply the data for such animation continuously. This section will explain how an aliased cartoon (i.e. one with "jaggies") can be displayed by QuickScan, and section 7.3.3 will explain how an anti-aliased (i.e. smoothed) cartoon can be displayed.

Before we get into the actual cartoons, we need to get a better understanding of the Sequential Runs **instruction** , since it is used extensively in cartoon representation. The Sequential Runs **instruction** generates a sequence of adjacent runs, left-to-right, starting from its **relative origin** (see section 5.2.5). In has certain advantages over the Run **instruction** , and certain disadvantages. Its key advantages are that it **stores** 2 runs to a 32-bit word, it provides full **data format** flexibility in **writing** to the **pixel storage cells**, and it can permit low **dispatch overhead** for the next higher **priority** object. Its key disadvantages are that the runs are limited to 256 pixels or less, there is the overhead of 1 word and 80ns for each run sequence, and that run sequence always has an even number of runs (if there's an odd number needed, the last run is made null). Both **instructions** take 80ns per run (although Sequential Runs has the additional overhead of 80ns per sequence), so the real issue is how much RAM we can save.

The following diagram shows an object efficiently represented by Sequential Runs:





Note: RAM array proportions are realistic: one line (——) is one row thick.

The seven slanted bars (each pattern represents a solid color) can be just as well represented with Run instructions as with Sequential Runs instructions . Let's see how we'd do this in each case.

Using the Run instruction , on each line we'd have to specify a Run instruction for each bar. Since the first bar is slanted, the relative origin of the first run on each line is different. Thus, we cannot put the first Run instruction into the first word and must instead waste the first word with a NOp instruction . With 7 Run instructions per line, we have 7 words per line description , and with a total of 160 lines, we have 7*160 = 1120 words to store this object. Its execution time, assuming minimal dispatch overhead for itself is 320ns+7*80ns= 880ns, but since the last instruction on every line is a Run instruction , we have to add 240ns of additional overhead to the next higher priority object's dispatch overhead (see section 6.2), and it is fair to consider 880ns+240ns = 1120ns as the total execution time.

Using the Sequential Runs instruction , we'd have to specify just one Sequential Runs instruction for each line. Now, it would be to our great advantage if we could make use of the first word. The problem with using the first word for this object is the relative origin of the first run is different on every line due to the slant. Notice, however, that the command word of the Sequential Runs instruction does not contain the actual run data for the instruction : this is specified in the subsequent data words (see section 5.2.5). Notice also, that we can specify a "transparent run," a run which spans a number of pixels but is masked, as one of the runs in the sequence. So, utilizing this information, we can place the command word in the first word with its relative origin set to the very leftmost point of the first slanted bar. Then, we can specify a transparent run on each line to make up the difference between that relative origin and the actual position of the left edge of the first slanted bar for that line (I drew a wedge in the diagram showing the area spanned by these transparent runs). Thus, we can get the desired image, yet also make use of the first word.

Since the command word for each line is contained in the first word, we need only store the data words. Since we have 8 runs on each line (counting the transparent run), we need 4 data words for each line. There are 160 lines, so we need 4*160 = 640 words to store this object, only 57% of the RAM needed to store the Run instructions . Assuming minimal dispatch overhead for itself, the exection time for each line of this object is 320ns + 4*160ns = 960ns. Since the last instruction on each line is a Sequential Runs instruction with 4 data words , then we

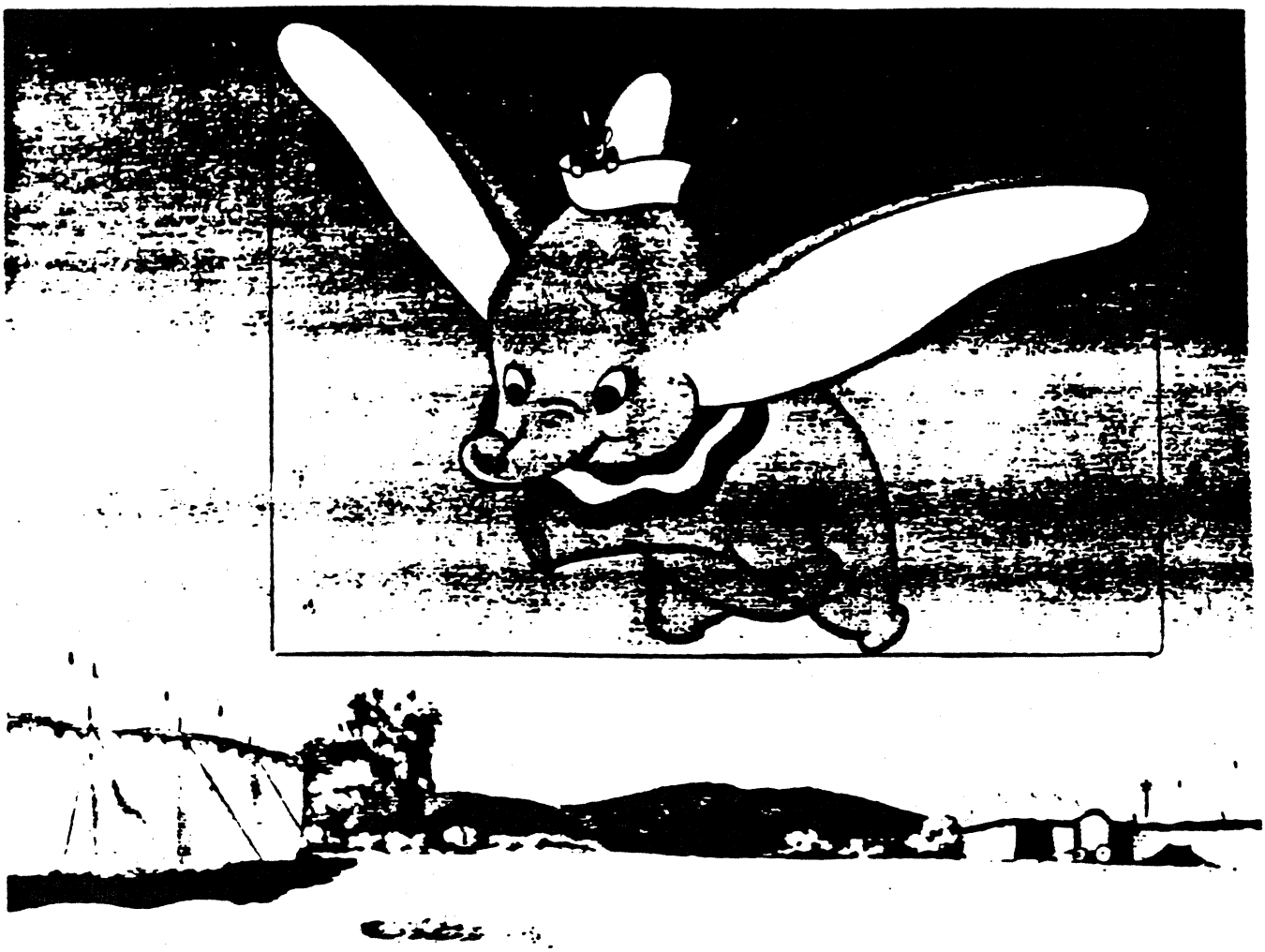
know that the next higher **priority** object will be dispatched with minimal overhead (see section 6.2), so the 960ns is truly the total execution time. Thus, by using Sequential Runs, we have slightly shorter execution time than with the Run **instruction** .

Note that despite this particular example, many **run-class** objects are represented more efficiently by using Run **instructions** than by using Sequential Runs **instructions** . The forest scene example of section 7.2.1 is a case in point. If you try to use Sequential Runs to represent this object, you find that the individual regions of color are for the most part separated from each other, so you end up wasting one transparent run getting from one color region to the next. So, effectively, you spend 2 runs to get 1 displayed run, and you lose the memory savings over the Run **instruction** . Also, some of the color regions are longer than 256 pixels (like **subobject** A), so we need up to 4 runs one after another just to span the whole region. Needless to say, Sequential Runs are not suitable for representing this object.

Okay, let's consider an example of a cartoon object. On the following page you'll find a frame from the Disney feature, *Dumbo*. In this example, I represent just Dumbo, himself, less the mouse in his hat. The rectangular outline I've drawn around Dumbo is the smallest rectangle we can make around the object, and is the region necessary for a comparable bit-map representation.

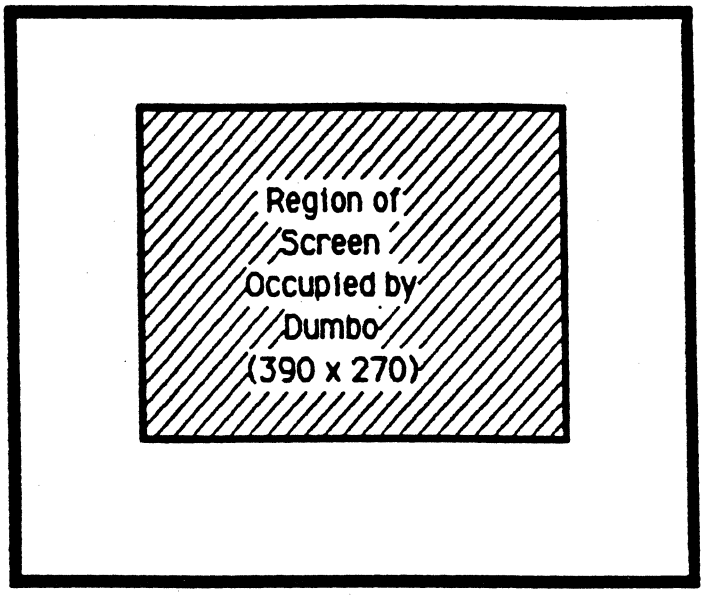
You'll notice that, unlike the other **run-class** objects we've considered previoiusly, Dumbo is composed of more than just large regions of color. He also has black lines which serve to both border these regions and provide additional details. We could represent these black lines by very short runs if we'd like, but it is more efficient in this case to break Dumbo into 2 **subobjects** , a color region **subobject** , and a black lines **subobject** , with the black lines overlapping the color regions. Then, we can efficiently represent the color regions with Sequential Runs, and represent the black lines efficiently with 1 bit/pixel bit-map (using embedded masks - see section 7.1.9).

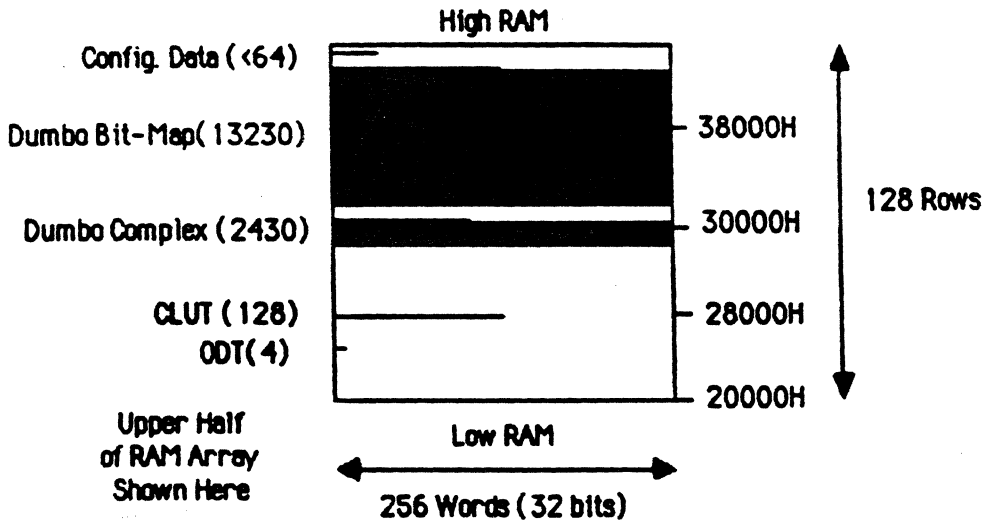
If we follow this approach, we end up with about 750 words for the color region **subobject's object description** and about 1680 words for



A Frame from Disney's *Dumbo*.

the bit-map subobject's for a total of around 2430 words, or 9720 bytes. If we were instead to specify Dumbo with a large 4 bits/pixel bit-map, he would require 13230 words, or some 52.9K bytes. So, even with a detailed, Disney-quality object, we are using only 18% of the memory we need with a bit-map. See the diagram and memory map below:





Note: RAM array proportions are realistic: one line (——) is one row thick.

There are few characteristics about the Dumbo image worthy of note.

First, this object is effectively masked in all regions surrounding Dumbo. That is to say, if we had a object of lower **priority** than Dumbo, then we'd be able to see this object in all of the little crevices around Dumbo (e.g. between his ears and his head, between his legs), just as we would expect if we had built a custom mask for exactly Dumbo's shape. Second, Dumbo is not anti-aliased, or rather his edges and lines are all going to be jaggy. Since he is so large in this particular image (he takes up 34% of the whole On-Screen area), these jaggies won't be all that noticeable, but nonetheless, it won't be quite Disney quality. This issue will be addressed in section 7.3.3.

Animation of Dumbo can be accomplished by storing the **object descriptions** for his various animation states in different places in RAM and changing the **start address** parameter in the **dispatch table entry** for the Dumbo object to point to the approriate animation state for each new frame of animation. The resulting effect is we'll see Dumbo smoothly flapping his ears and soaring around, and we'll be appropriately seeing the background around Dumbo's exact outline at all times. If this sounds like no big deal since that is what you'd expect to see, bear in mind that sustaining such animation in real-time with a smoothly shaped object this large cannot be done by any but the most expensive graphics display systems available. With QuickScan it's child's play.

If we wanted to animate Dumbo with very good quality animation, we'd need to sustain a rate of about 15 frames/sec. Assuming that each frame has roughly the same amount of data, then we would need 9360*15 = 140400 bytes of data per second. Even if we applied no additional compression of Dumbo's representation than what we have already done with the run and bit-map **subobjects** (and we certainly could compress it significantly more), the CD ROM could sustain this data rate with enough time left over for some simple branching. So Dumbo's flying around with excellent quality animation under a child's interactive control can be a reality with QuickScan.

And, if we did compress cartoon character representations further on the CD ROM, and then expanded them back upon reading them off, we could have several independent objects being animated in real time simultaneously. We'd implement this by loading up a few frames of one object at once, then jumping to another track on the CD ROM and loading up a few frames of another object, jumping and loading a few frames of

another object, and so on until we'd loaded frames for all the objects. Then, we'd jump back to the first object to load its next few frames, and continue through the cycle again. Meanwhile, we'd sequence through the frames that had been loaded by changing the start address of each object at each animation state to point to the appropriate frame, and the objects would animate smoothly, each independent of the others.

What's extraordinary about this capability is that each object, within limits, is in its own time continuum. What I mean by this, is the various objects on the screen do not have to be synchronized with each other in time. So, if Dumbo is flapping his ears to fly, and there is Mickey Mouse on the ground waving his hand, the two actions of flapping and waving don't have to be in sync. In fact, if Mickey wanted to stop waving his hand, and walk away, he could do so without Dumbo's motion being affected. The possible applications for multi-object interactive animation are just amazing.

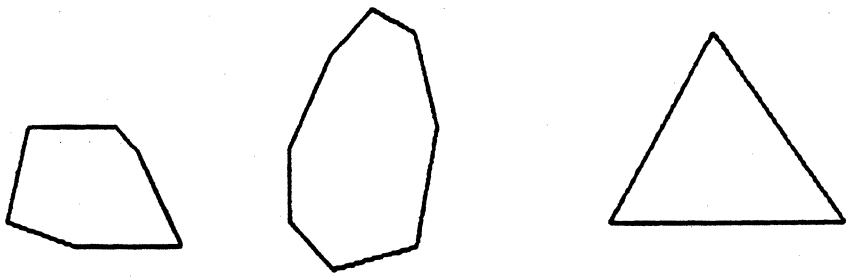
You may be wondering where I dug up the estimates for the amount of RAM needed to represent Dumbo. My method was to xerox the cartoon image onto a piece of 1/8" square graph paper. Then I defined each square of the graph paper to be a 10 by 10 pixel block, and proceeded to count the number of runs and the number of words of bit-map needed to render one line out of every 10 lines of the image with an accuracy of 10 pixels horizontally, being conservative in any rounding off. I then multiplied my results times 10, working from the assumption that the other 9 lines in each 10 line group required roughly the same representation as the line I measured. I am confident that the precise memory requirements will be somewhat less than my estimates, because not being able to work with each line individually, I had to take a detail occurring in one line of the object and pay for its representation in 10 lines to be sure it was accounted for.

I won't belabor you with the details of the breakdown of the object description , but the worst case line description has 8 sequential runs, and 6 words of 1 bit/pixel bit-map. The Sequential Runs instruction is in the first word, as in the previous example, so we have 80ns for each run, 80ns for the Bit Map instruction , and 80ns for each bit-map data word, for a total of 8*80ns+80ns+6*80ns = 1200ns. Assuming minimal dispatch overhead , we have 1200ns+320ns = 1520ns worst case execution time (since the line description ends with 6 1 bit/pixel

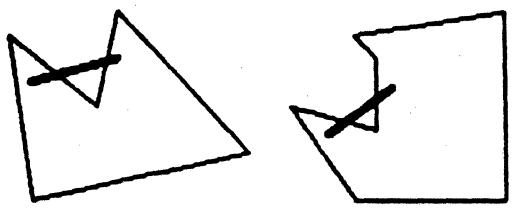
bit-map **data words** there is no additional **dispatch overhead** for the next higher priority object). Out of the available 32µsec, Dumbo takes up 4.75%. The CPU overhead is minimal. So, we could very well have 20 cartoon characters of Dumbo's size and complexity flying around on the screen at once. If you flip back a couple of pages and take another look at just how big Dumbo is, notice that no one has ever before seen such a capability in real-time computer graphics. It's only possible because of the fully parallel runs. I really think that kids (of all ages) are going to go wild.

7.2.4. 3-D Polygon Modeling

Fully parallel runs are extremely useful in efficiently representing filled polygon regions. Since a filled polygon is a single color region, it is the quintessential **run-class** object and can be readily and deterministically converted into a set of runs. In fact, there is a large and useful class of polygons which can be represented by a single run for each line. And, within that set are the convex polygons (polygons for which no 2 interior points exist with a segment between them that crosses into the exterior) which in any orientation can be represented with exactly one run for each line of the polygon's height. Examples of convex and non-convex polygons are shown below.

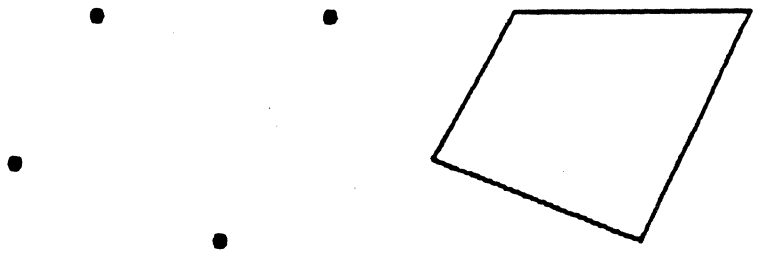


Convex Polygons



Non-convex Polygons
(with segment disproving convexity)

The convex polygon subset is of great interest in 3-D modeling applications. Given the vertices of a convex polygon, we can compute the lines between these vertices that form the perimeter of the polygon. Then, by scanning the polygon from top to bottom, line-by-line, we can easily generate a run for each line extending from the leftmost perimeter of the polygon to the rightmost perimeter. (Since the polygon is convex, we know that one run for each line will be necessary and sufficient.) The runs for all of the lines, once submitted to QuickScan as an **object description** , will generate an image of the convex polygon specified by the vertices. This process is known as *scan-converting* and diagram of the process is shown below:



Start with Vertices Compute Lines of Perimeter

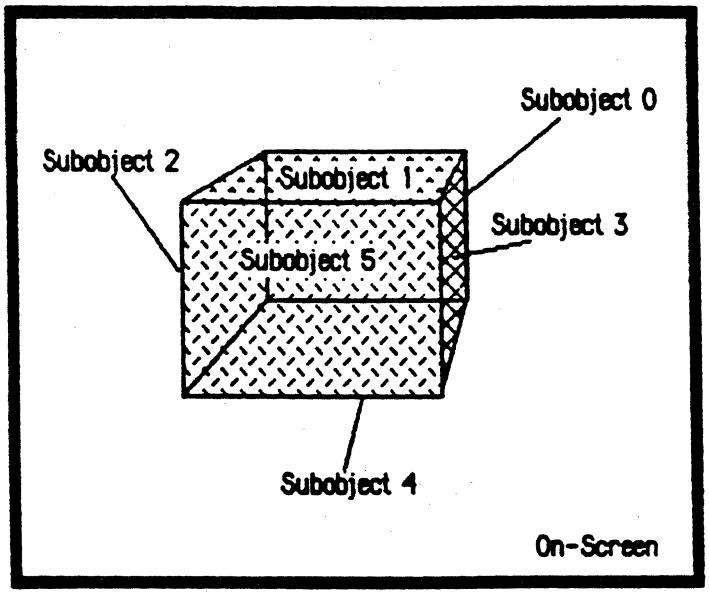
Compute a Run for Each Line
From the Left Edge to the Right Edge

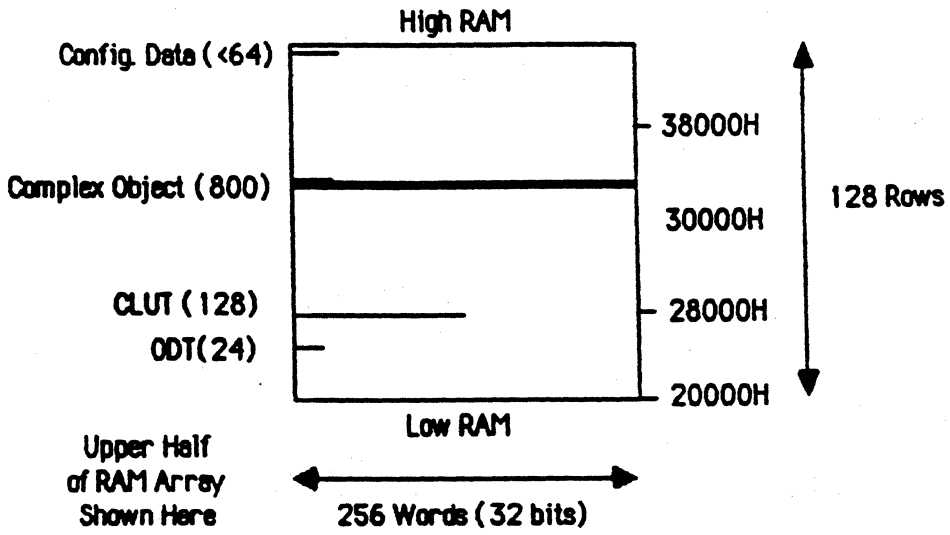
Scan-converting a Convex Polygon

If now we perform 3-D coordinate transformations (translation, scaling, rotation, or perspective) on these vertices, we will compute a new set of vertices reflecting the transformed position of the polygon. By scan-converting these vertices we will deterministically generate a new set of runs describing the transformed polygon, and QuickScan will display an image of the transformed polygon.

A polyhedron is a solid object which has a polygon for each face.

Examples of polyhedra are cubes, boxes, and pyramids, but they can, of course, be very complex. If the faces of a polyhedron are convex, they we can easily generate the polyhedron from the union of convex polygons. In effect, the polyhedron is a **complex object**, and each polygon face is a **subobject**. A rectangular solid polyhedron composed with 6 **subobjects** for its faces is shown below with a pattern identifying each visible face (the hidden lines showing through would not be visible in the real display):





Note: RAM array proportions are realistic: one line (——) is one row thick.

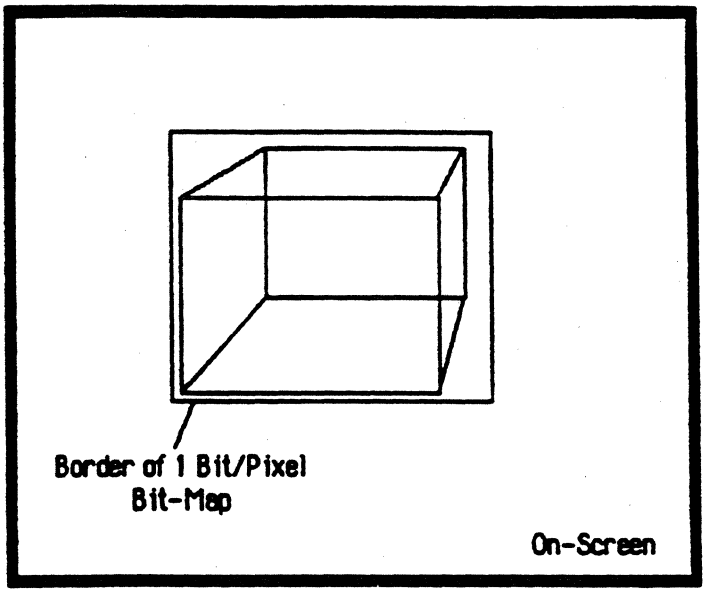
To generate this polyhedron **complex object**, each of the polygon **subobjects** was scan-converted and assigned a **subpriority** based on its z-coordinate (z is perpendicular to this page of paper). Then, the individual **subobject's object descriptions** were interleaved (just as we did in section 7.2.1), and that was it.

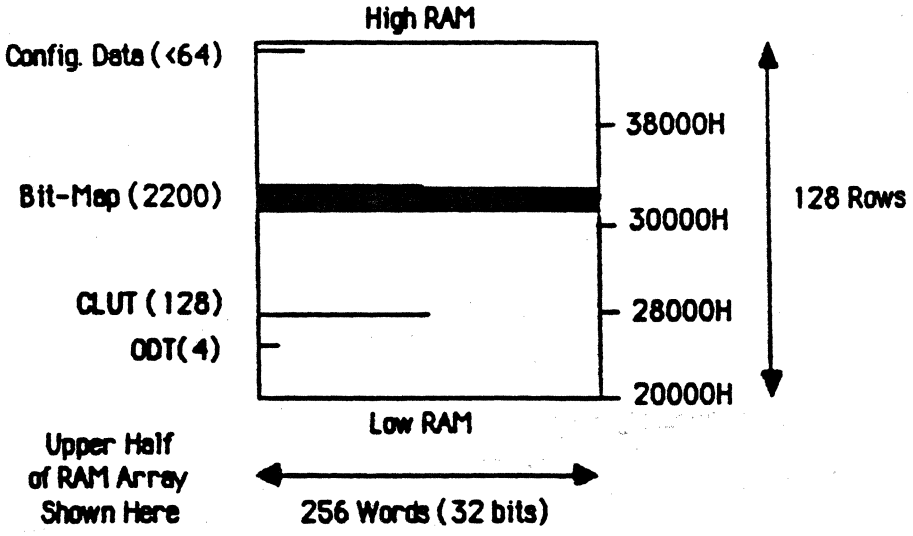
If we wish to apply a 3-D transformation to this polyhedron we just apply the transformation to the vertices of each of **subobjects**, scan-convert them, and interleave them again. The only extra work we have to do beyond what we had to do to transform the polygons individually is to determine the correct **subpriorities**. Without going into the details, this means at worst one more vertex transformation per polygon and a sort of the computed z-coordinates.

If this seems simple and straightforward, then you're right. It is. So why don't we see more 3-D polygon graphics displayed on personal computers and video games? Well, for one thing, computing a great many vertices can take a fair amount of time, and as objects get complex, computers without special hardware slow to a crawl. But, certainly simple objects like cubes and pyramids don't require much computational effort. Why don't we see cubes spinning around in space for neat effects in video games? Ah, now we're getting to the crux of the problem. It's not limits in computational speed which is the first roadblock. It's limits in display speed - large objects just take too long to draw to the screen.

We <u>do</u> see examples of real-time 3-D graphics. The Battlezone and Star Wars video games are two excellent 3-D games, but they have no solid objects: every polygon is represented with outlines. Although they can do the coordinate transformations in real-time, they can't update the frame buffer fast enough. I have a demo disk for the Macintosh (come and see me if you want a copy) which has a 3-D image of a Macintosh Computer tumbling in space. The Mac has no trouble keeping up the coordinate transformations in real-time, and since it is just drawing the outlines of the shape, it has no trouble keeping up with the display update. Pacific Data Images told us that when they wanted to run through the motion of a 3-D scene in real-time, they just generated the outlines of the objects. They found themselves in the same situation as the Atari video game programmer and the Macintosh programmer: just can't update the display of those solid polygons fast enough.

Well, fortunately for us, with QuickScan we don't have to update the display; it does it for us. And, it can keep up because it generates runs fully in parallel. In fact, it takes just as long to generate the solid faced polyhedron above as it takes to display the outlined polyhedron shown below (except for the **subpriority** sort):





Note: RAM array proportions are realistic: one line (——) is one row thick.

Instead of drawing a left and right pixel for the lines on each edge of

the polygons for this diagram's display, we just specified a run with a left and right limit for the first diagram's display. In fact, the run is really easier than the outline because we don't have to worry about setting one bit in the middle of a 32-bit word for the one pixel of the line.

We already know that we have the computational muscle to do the coordinate transforms and outline drawing (seriously, I'd love to show you the Mac disk), so with QuickScan we can definitely manipulate solid 3-D polygons and polyhedra in real-time. Throw in a floating point co-processor, and we'll really cruise!

One of the beautiful things about these fully parallel runs is that each run takes the same amount of time to draw, 80ns, regardless of how long it is. So, no matter how big the polygons get (if for example, we get very close to them), it will take the same amount of time to display them: 80ns/polygon. We have a fixed and deterministic execution time for each polygon with QuickScan. Period. The horrendous problem of determining what you have time to display after the polygons have been transformed with which you have to wrestle in any other graphics system environment, is trivial with QuickScan.

The other nice thing is that the hidden surfaces are automatically removed by the prioritization of the **subobjects** . You don't have to be the least bit concerned about which part of which polygon is obscured and which isn't. You can forget about those backfacing algorithms that try to reduce the amount of updating required by identifying completely hidden polygons. It doesn't matter anymore, QuickScan takes care of it all.

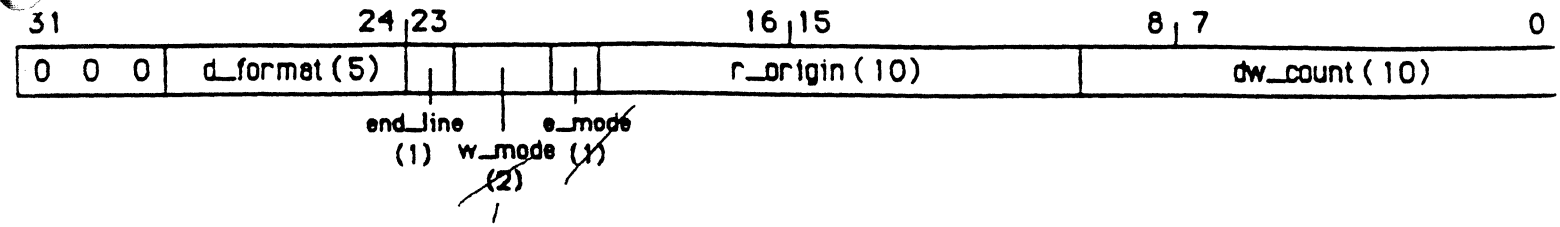
But, of course there has to be limit of how many polygons QuickScan can put up. This limit is of course dependent on **dispatch overhead** , row crossings, and other factors, but to get a rough idea, we just need to divide the time to execute a Run **instruction** , 80ns, into the total line time 32µsec: 32µsec÷80ns = 400. So, if you wanted to, you could put up an object made up of almost 400 filled convex polygons, regardless of their size or shape. And, if you got the computational power (or special hardware - see Appendix B) to transform the vertices, you could manipulate this gargantuan object in real-time. Pretty awesome.

Now, it would be nice if we could apply a lighting model and shade the faces of the polyhedra realistically. I'll show you how in section 7.3.2.0

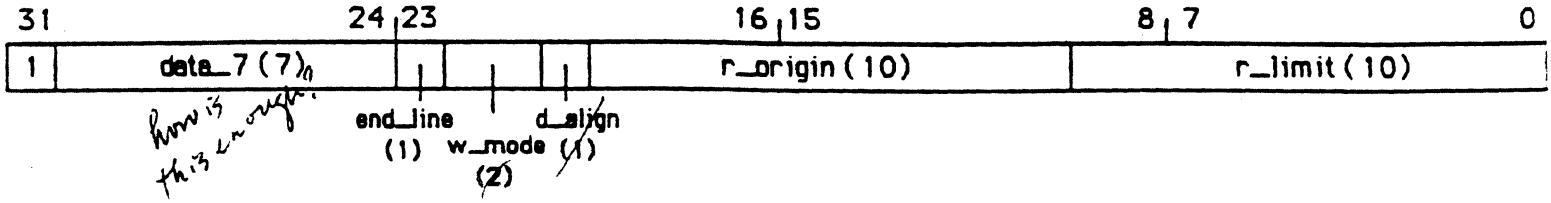
A.1. Command Word Format

SGP 3/4/85

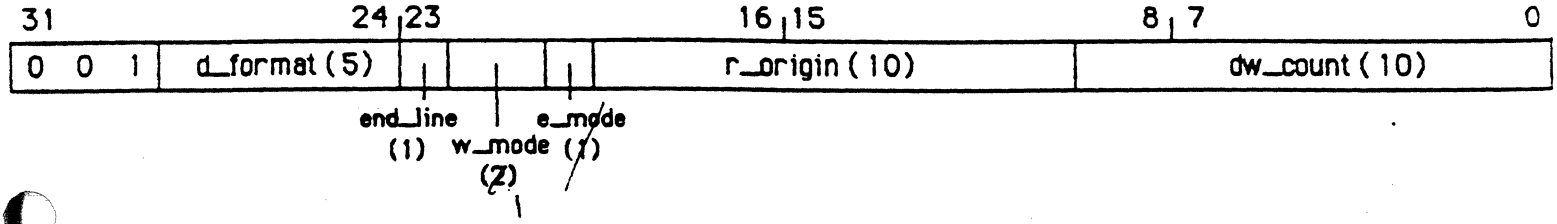
Bit Map (BMap)



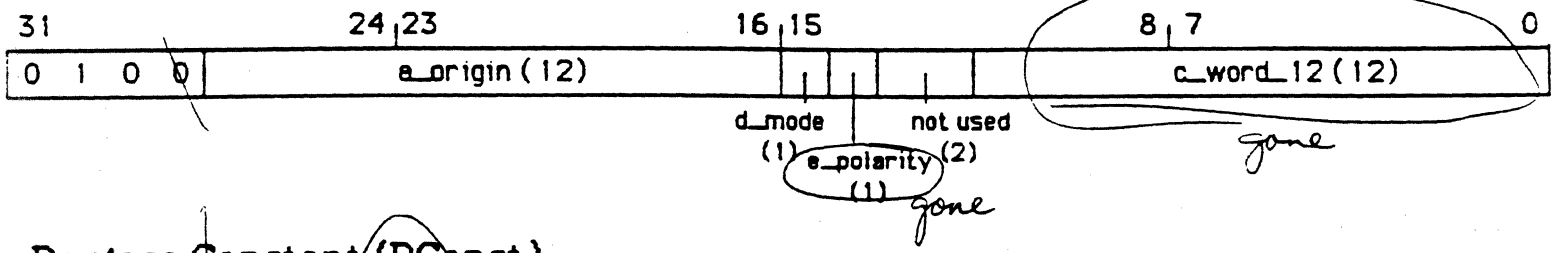
Run (Run)



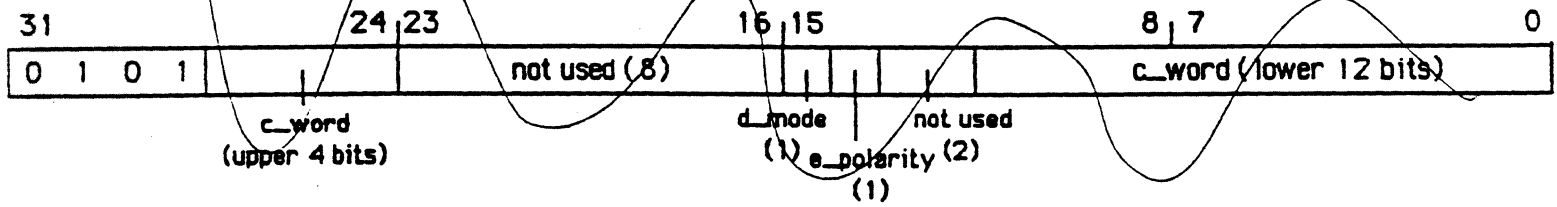
Sequential Runs (SRuns)



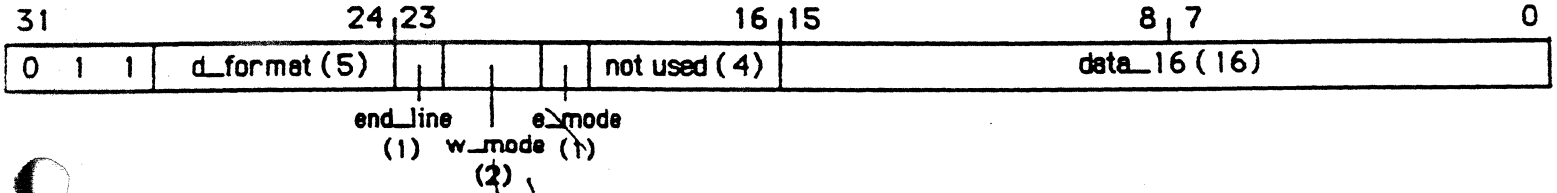
Context Switch (CSwitch)



Replace Constant (RConst)



Run Screen (RScreen)

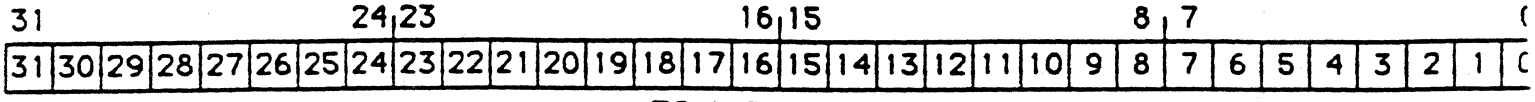


A.2. Data Word Format

SGP 3/4/85

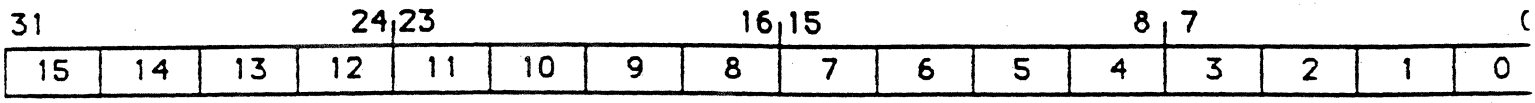
Bit Map Data Word Formats

1 Bit/Pixel



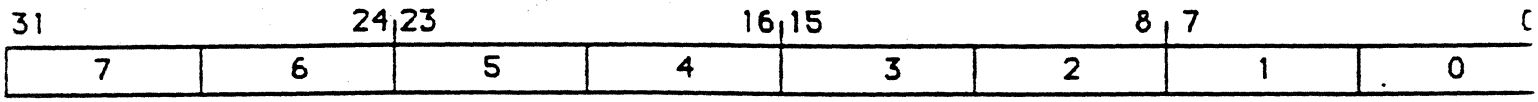
32 1-Bit Pixels

2 Bits/Pixel



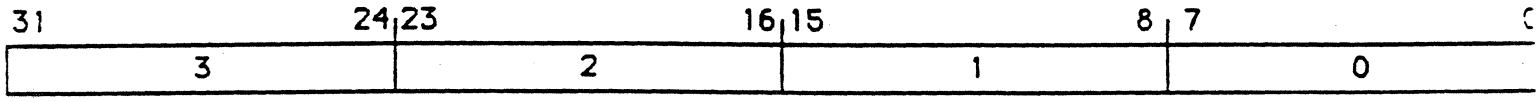
16 2-Bit Pixels

4 Bits/Pixel



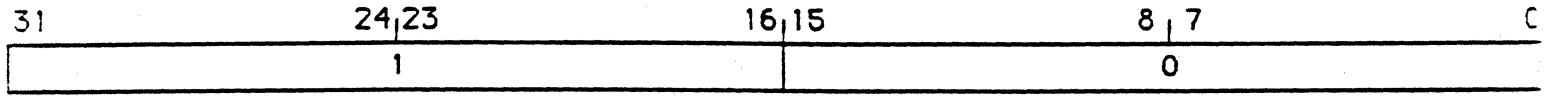
8 4-Bit Pixels

8 Bits/Pixel



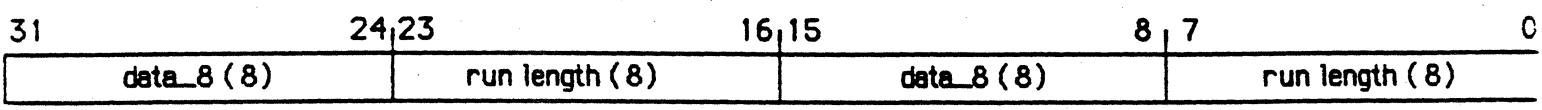
4 8-Bit Pixels

16 Bits/Pixel



2 16-Bit Pixels

Sequential Runs Data Word Format

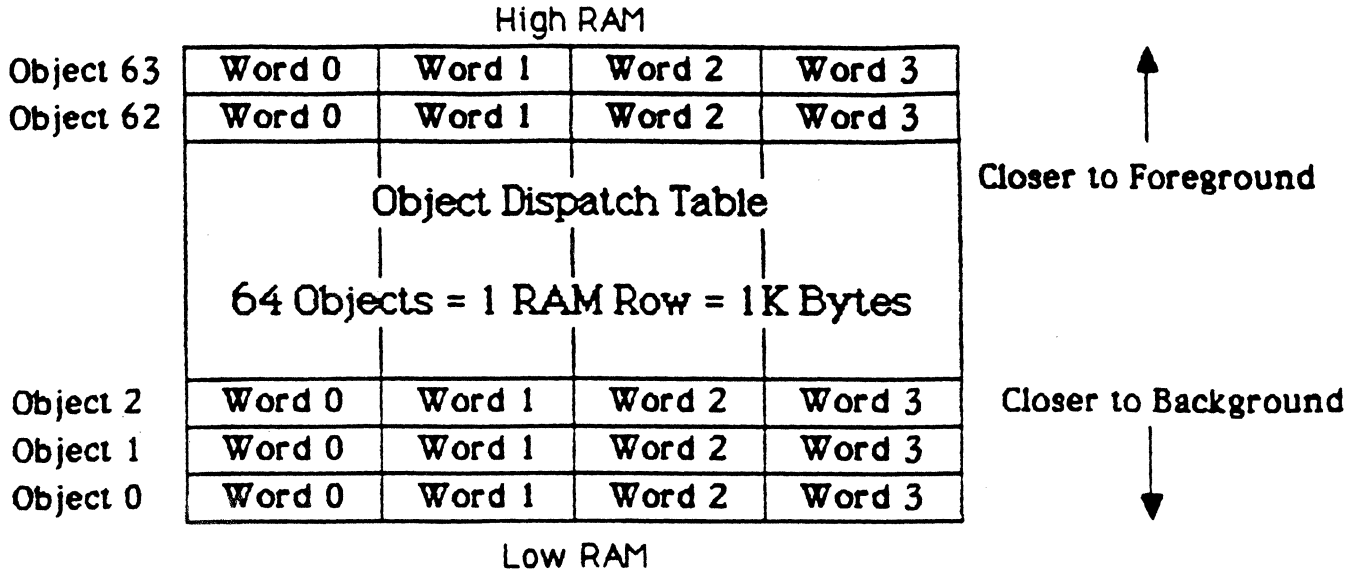


second run — 2 Short Runs — first run

A.3. Dispatch Table Word Format

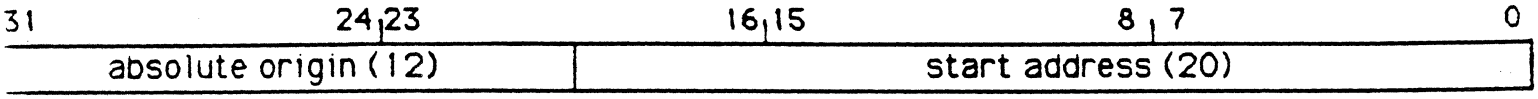
Dispatch Table Format

SGP 3/4/85

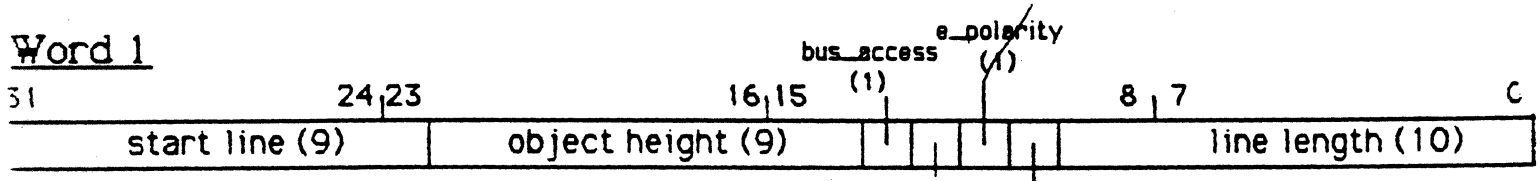


Dispatch Table Entry Format

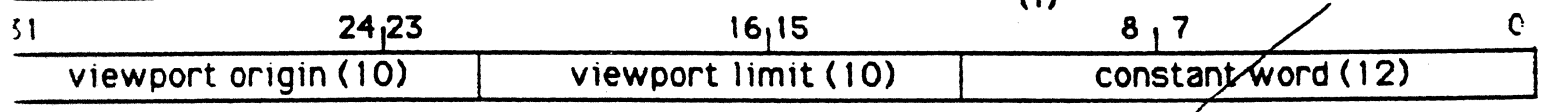
Word 0



Word 1



Word 2



Word 3