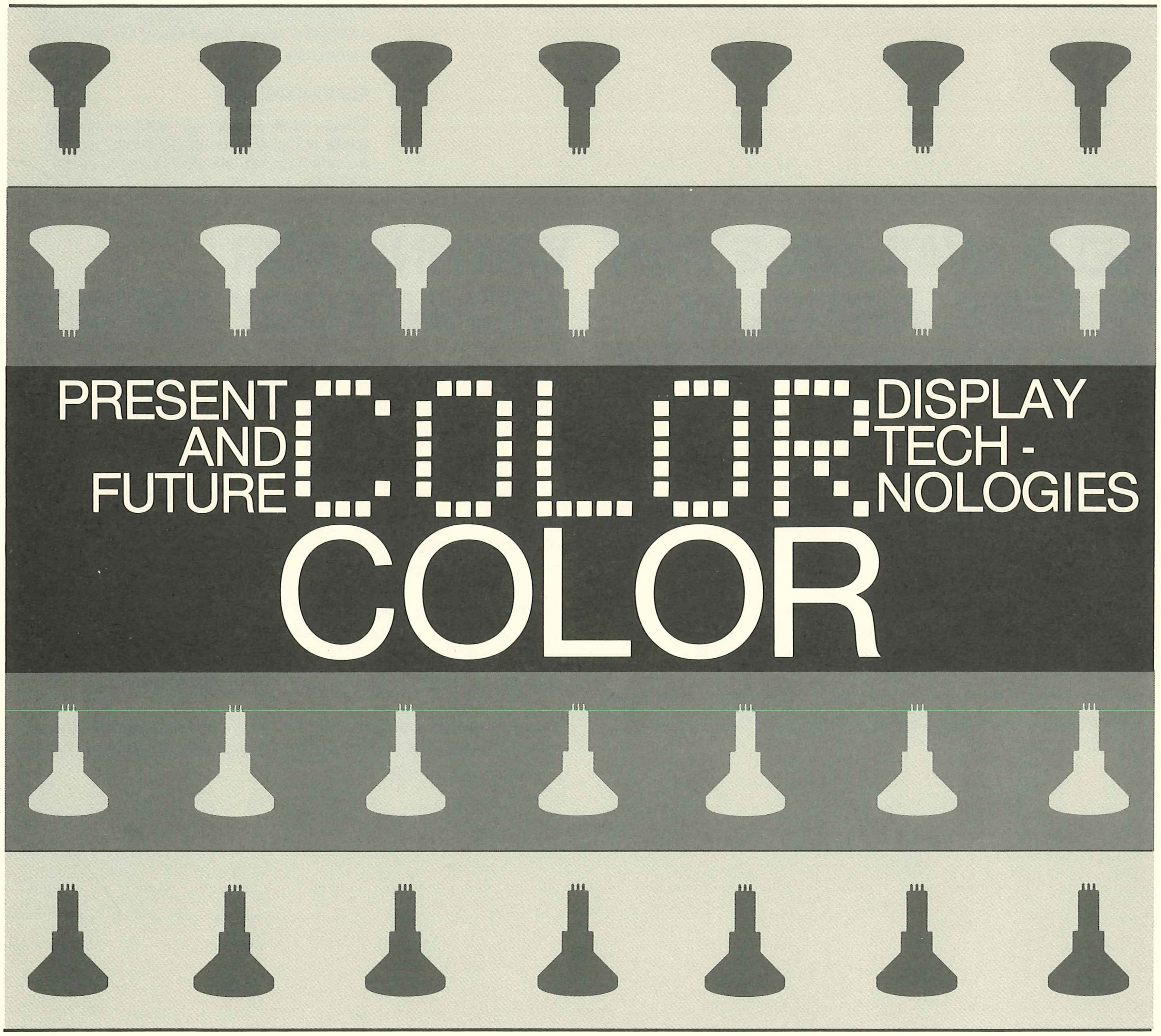
TEK INFORMATION FOR TECHNOLOGISTS AUGUST 1983

TECHNOLOGY report

COMPANY CONFIDENTIAL



Tektronix®
COMMITTED TO EXCELLENCE

CONTENTS

Volume 5, No. 4, August 1983. Managing editor: Art Andersen, ext. MR-8934, d.s. 53-077. Cover: Joe Yoder; Graphic illustrator: Jackie Miner. Composition editor: Jean Bunker. Published for the benefit of the Tektronix engineering and scientific community.

Why TR?

Technology Report serves two purposes. Long-range, it promotes the flow of technical information among the diverse segments of the Tektronix engineering and scientific community. Short-range, it publicizes current events (new services available and notice of achievements by members of the technical community).

Contributing to TR

Do you have an article or paper to contribute or an announcement to make? Contact the editor on ext. MR-8934 or write to d.s. 53-077.

HELP AVAILABLE FOR PAPERS, ARTICLES, AND PRESENTATIONS

If you're preparing a paper for publication or presentation outside Tektronix, the Technology Communications Support (TCS) group of Corporate Marketing Communications can make your job easier. TCS can provide editorial help with outlines, abstracts, and manuscripts; prepare artwork for illustrations; and format material to journal or conference requirements. They can also help you "storyboard" your talk, and then produce professional, attractive slides to go with it. In addition, they interface with Patents and Trademarks to obtain confidentiality reviews and to assure all necessary patent and copyright protection.

For more information, or for whatever assistance you may need, contact Eleanor McElwee, ext. 642-8924. □

WRITING FOR *TECHNOLOGY REPORT*

Technology Report can effectively convey ideas, innovations, services, and background information to the Tektronix technological community.

How long does it take to see an article appear in print? That is a function of many things (the completeness of the input, the review cycle, and the timeliness of the content). But the minimum is six weeks for simple announcements and as much as 14 weeks for major technical articles.

The most important step for the contributor is to put the message on paper so we will have something to work with. Don't worry about organization, spelling, and grammar. The editors will take care of that when we put the article into shape for you.

Do you have an article to contribute or an announcement to make? Contact the editor, Art Andersen, 642-8934 (Merlo Road) or write to d.s. 53-077. □

PRESENT AND FUTURE COLOR DISPLAY TECHNOLOGIES FOR GRAPHICS

John J. McCormick is the manager of Display Research, part of Tek Labs. John joined Tektronix in 1965. He has been involved with oscilloscope designs, A/D converters, and display technology including liquid crystals and electroluminescence. Earlier, he was a member of the technical staff at RCA Laboratories where he worked in quantum electronics. John received an MS in electrical engineering from Princeton University in 1965 and a BS in electrical engineering from the University of Kansas in 1962.

Computer graphic displays are increasingly using color to improve productivity in complex applications. But, to obtain color, the user typically must give up something else, such as resolution. This article describes the commonly used methods for generating color graphic displays, all of which are based on the cathode-ray tube (CRT). The article also identifies the more important characteristics of those color displays. After assessing these characteristics, it looks briefly at what some new approaches still in the laboratory might do to improve color displays. John presented much of this information at the Conference on Computer Graphics Applications for Management and Productivity (CAMP '83) in Berlin, Germany, March 1983.

It is the CRT that determines what characteristics a color display will have. All other factors, such as color display systems, are subordinate to the strengths endowed and the weaknesses imposed by the physics and economics of the cathode ray tube.

Three types of color CRTs are used for graphics displays today. The shadow-mask dominates, but the penetron and the direct-view storage tube with color write through enjoy special positions. (Throughout this article, "display" is used to mean the CRT and associated hardware.)

Let's look at the tube that Tektronix has never used – the penetron.

The Penetron Tube

The penetron or penetration tube resembles a monochrome CRT except that the phosphor is actually two phosphors, in separate layers. Each layer requires a different electron-beam energy to activate it. In one method, shown in figure 1, the red phosphor acts as an energy barrier that the beam must penetrate to reach the green phosphor – hence the name *penetron.*

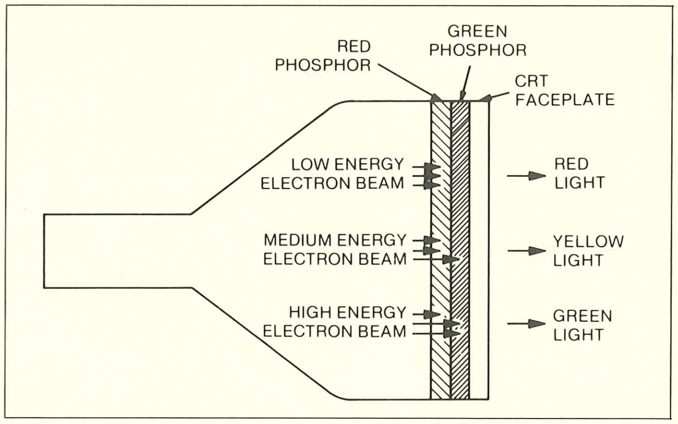


Figure 1. An example of a penetration-phosphor CRT. By changing beam energy, one or the other (or both) phosphors are activated.

Although it is theoretically possible for such a tube to have three phosphors, all practical tubes have only two, usually red and green. A low-energy (6 kV) electron beam excites only the outer red phosphor and is stopped by the dead layer. A high-energy beam (12 kV) penetrates the dead layer and excites the green phosphor.

Although the different beam energies can be obtained in several ways, two are most common. In one method, the target potential is switched. In the other, two electron guns are employed, each at a different potential, and then current is simply turned on in one or the other gun. In either method, each color requires a different deflection voltage or current to deflect the beam to a particular point on the screen. This requirement must be taken into account in the system electronics.

Color Write Through

The second type of color tube is a Tektronix storage CRT that utilizes the penetration effect. This direct-view storage tube (DVST), shown in figure 2, uses "color write through" (CWT) to provide color.

The tube consists of a writing gun that operates at a large negative potential (6 kV) with respect to the target, an array of low-energy (several hundred volts) flood guns, and a special phosphor target.

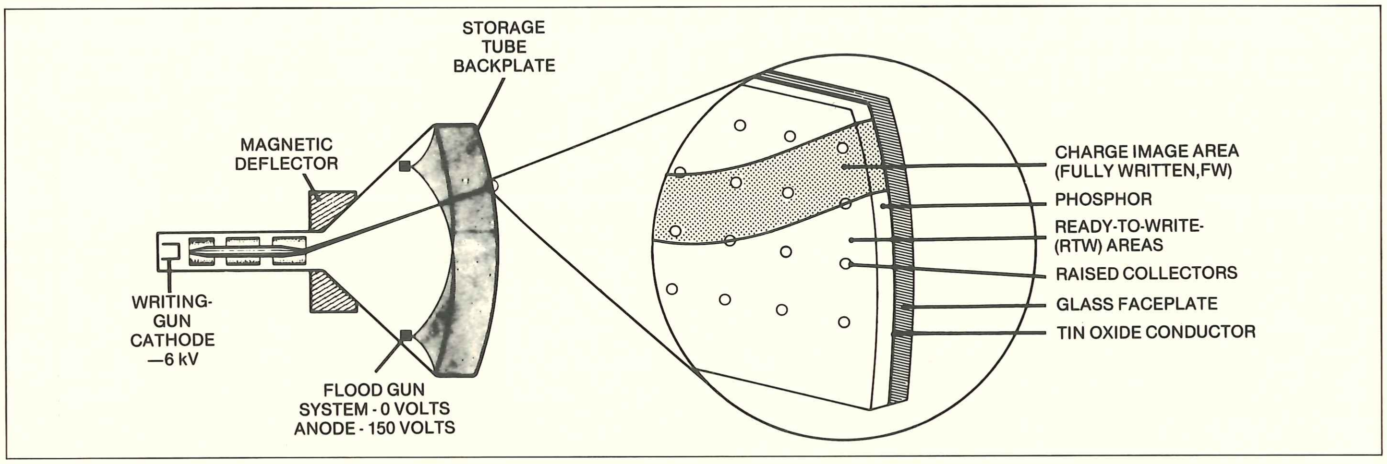

STORAGE TUBE BACKPLATE
MAGNETIC DEFLECTOR
WRITING-GUN CATHODE —6 kV
FLOOD GUN SYSTEM - 0 VOLTS ANODE - 150 VOLTS
CHARGE IMAGE AREA (FULLY WRITTEN,FW)
PHOSPHOR
READY-TO-WRITE-(RTW) AREAS
RAISED COLLECTORS
GLASS FACEPLATE
TIN OXIDE CONDUCTOR

Figure 2. The direct-view storage CRT with color write-through uses the penetration technique and the two-gun structure standard in DVST. A second color is obtained by employing two phosphors instead of just one.

The phosphor is separated from a transparent conductor by an insulating layer pierced with an array of conductive dots. As in the penetron, the phosphor is actually two phosphors. In this case, small particles of each phosphor are mixed together. The normal green phosphor for storage is mixed with red phosphor particles surrounded by a dead layer.

The normal storage operation of the DVST[1] is unchanged by the special phosphor structure.

The flood guns at ground potential continuously flood the entire phosphor target with electrons, maintaining it near ground potential through the action of secondary electron emission. When the DVST is in the storage mode, the writing gun scans the target and "writes" by leaving a charge on the phosphor dielectric. Because the high-energy beam produces a secondary electron emission greater than unity, the written areas charge to a moderate positive potential (about 300 V) with respect to the flood-gun cathode. Therefore, when the flood electrons strike the written target areas, they cause the phosphor to luminesce. The unwritten areas, which are maintained near ground potential by the flood electrons (which charge the screen negatively) do not luminesce.

To prepare the screen for new information, the written areas are erased by pulsing the conductive backplate and resetting the phosphor potential to its lower bistable state.

If the writing-beam current is decreased below some threshold value, no information is stored, but the phosphor luminesces briefly as a result of the writing gun's high energy. This phenomenon is known as "write-through." With the penetration phosphor in place, the writing beam's high energy electrons penetrate the red phosphor's dead layer and excite both phosphors, thus producing a yellowish green trace. This is *color write through.*

Meanwhile, because the flood electrons have much lower energy, they excite only the green phosphor where the image is stored. Only one color is available for stored images, but other colors can be obtained in the non-storage mode or by writing over the stored trace with the write beam.

Shadow-Mask CRT

Overwhelmingly, the color display of choice is the shadow-mask CRT. It is also, in some respects, the most complex.

In the shadow-mask tube, shown in figure 3, three electron guns are used to address either three primary-color phosphor dots or three primary-color stripes. The dots and stripes are grouped in *triads,* so closely spaced that they appear to the eye as one. Color results from a proportional mixture of the luminescence from the individual dots or stripes of the triad. The shadow mask assures that each beam addresses only its assigned color dot or stripe. The beams from the red, green, and blue guns must be angled properly to pass through the mask openings and strike their corresponding phosphor; all other phosphors dots are "shadowed."

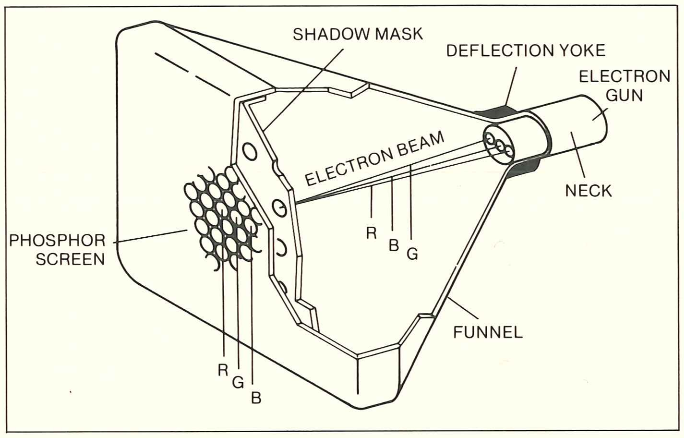


Figure 3. An example of a shadow-mask color CRT. This tube dominates the color graphics market today.

Because the dot pattern permits smaller horizontal spacing between triads, dots are used when maximum resolution is required. The guns are typically configured in a delta for dots and in-line for stripes. Although in-line guns can also be used with phosphor dots, the delta configuration produces smaller spots. In-line guns, on the other hand, require less convergence circuitry.

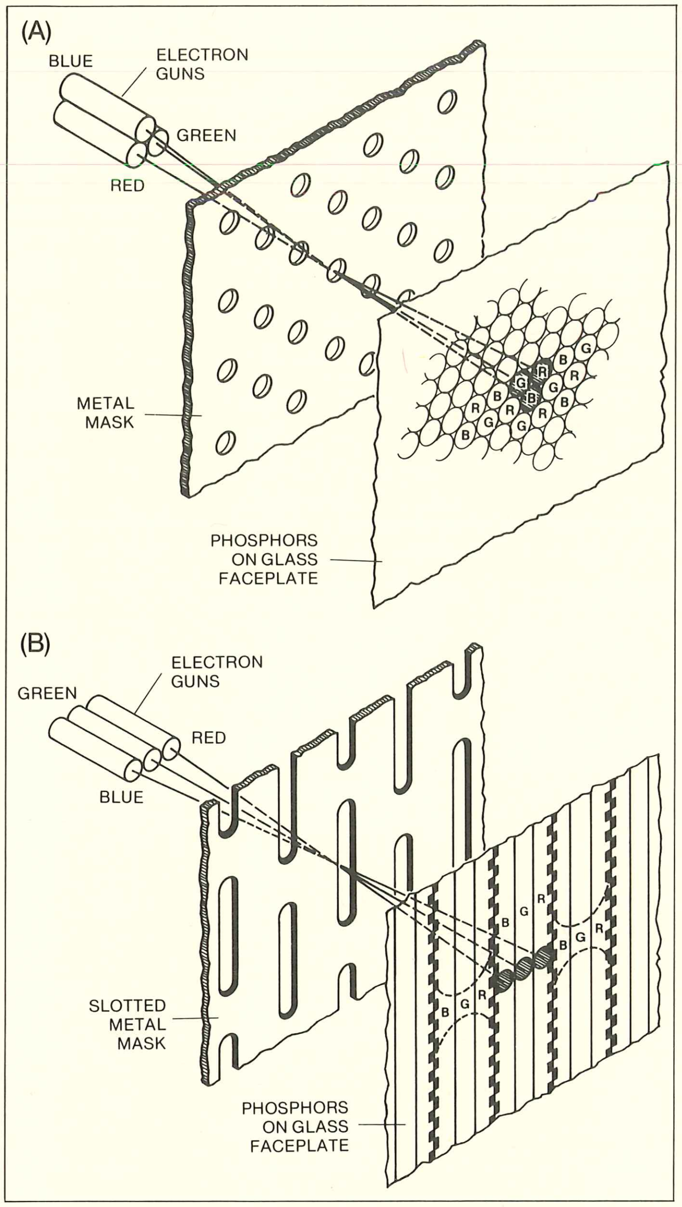


Figure 4. Shadow-mask CRTs use one of two gun/phosphor pattern configurations: (A) Delta gun arrangement with dot-patterned phosphor; (B) In-line gun arrangement with strip-patterned phosphor.

Misconvergence typically occurs because the three beams pass through the deflection yoke at slightly different angles and locations, and are thus deflected to slightly different points on the screen. Correction circuitry is necessary to maintain convergence and thereby assure registration of the three primary colors.

Misconvergence can be avoided by displaying the three primary colors one at a time, and never simultaneously. This technique is used in Tek's Color DAS. The Color DAS displays just three colors: red, green, and yellow. The CRT is a special with red, green, and yellow dots.

Color Display Systems

Three methods are commonly used for displaying color graphics on CRTs: vector storage, refresh vector, and raster refresh. Vector-storage displays must use the direct-view storage tube. In theory, both the refresh-vector and the refresh-raster displays could use either the penetration CRT or the shadow-mask CRT. However, the refresh vector can more easily be implemented with the penetration CRT, while the refresh raster is almost exclusively implemented with the shadow-mask CRT.

Refresh Vectors – A typical refresh-vector display system using a penetration CRT is shown in figure 5. Vectors are drawn by deflecting the beam between the specified end points of the vector. An image formed from a combination of vectors can be rapidly changed by merely changing the vector end points. Since relatively few points are needed to define images consisting mostly of lines, the dynamic capabilities of this type of display are excellent. But since the complete image must be refreshed (repeated) often so that the viewer perceives a constant luminance, deflection speed usually limits the number of vectors that can be drawn before flicker becomes a problem.

Because the penetron requires a different deflection-amplifier gain for each of the two colors, field-sequential operation is generally used in refresh-vector systems. The red information is written in the first field, and then the deflection-amplifier gain is changed before the green information is written in the second field. Because producing a third color by registering (combining) two colors is difficult, a third color (if provided) is usually produced by an intermediate acceleration voltage in a third field.

Vector storage – By storing vectors on the screen, the vector-storage display overcomes the flicker-imposed limit on the number of vectors that is inherent in a refreshed-vector system. A typical vector-storage display system using a DVST with color write-through is shown in figure 6. Although the DVST deflection system is similar to that of the refresh-vector system, deflection speed is not critical – when the DVST is set to the storage mode, green vectors are stored on the screen. Deflection speed, therefore, affects only the time required to draw a complete graphics image. There is no flicker, no matter how many vectors are drawn. The DVST, therefore, is an excellent display for complex, intricate images.

By employing the unique capabilities of the DVST with CWT, images with another color can be added to the display. When the write beam is operated with reduced current to prevent storage, a yellow-orange spot is produced on the screen. This non-stored spot can be deflected to produce refreshed vectors. The number of vectors in this second color, however, is limited by maximum deflection speed and flicker.

A third color can be obtained by writing the refresh vector on top of an identical, stored vector. This mode produces a greenish-yellow. Unlike the penetron, no misregistrations are encountered since the same writing-beam potential is used to write in all modes.

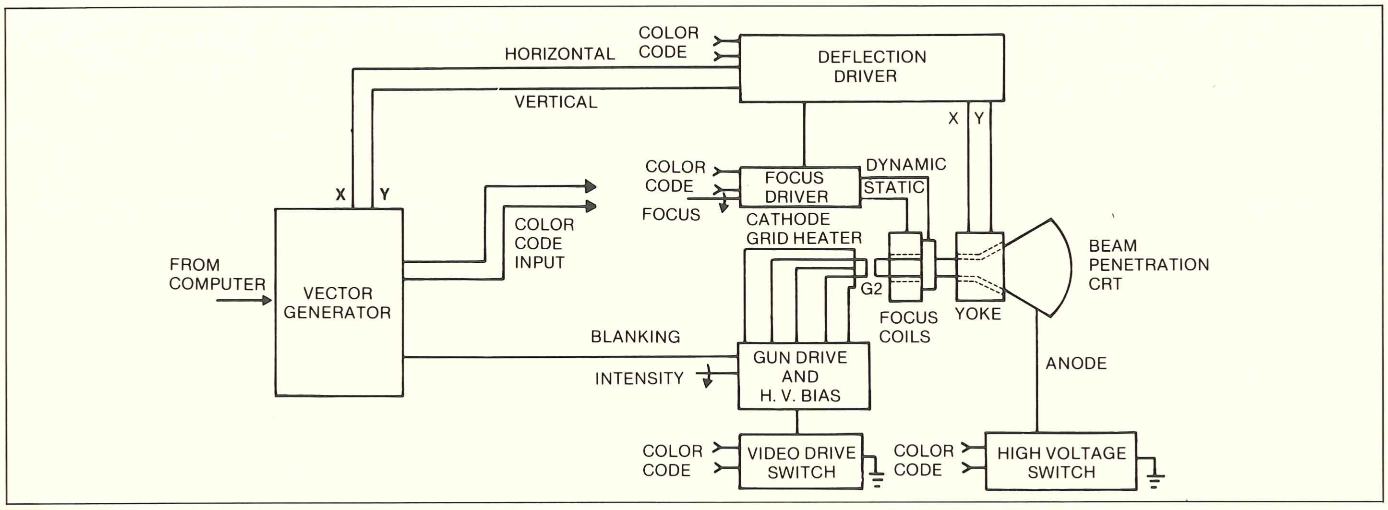


Figure 5. A typical penetration CRT refresh-raster display system. Because the deflection drives need to change for each color, field sequential operation is used.

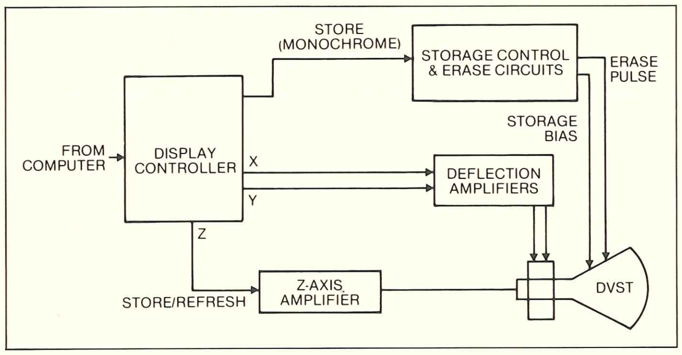


Figure 6. Block diagram of a vector storage display system with color write-through capability.

While a stored image must be erased all at once, a refreshed image can be selectively updated and even provide dynamics – displaying the motions of machinery, for example.

Refresh raster – The refreshed raster is the most common color graphic display. An example of such a system using the shadow-mask CRT is shown in figure 7(a). In this type of display, three beams are deflected together over the phosphor screen in a predetermined raster pattern, as shown in figure 7(b). A bit-map memory determines when each of the three guns receives current, and how much, and thereby how much of each color is produced at each point (pixel) on the screen.

The information in the bit map must be read out repeatedly at a rate fast enough to avoid flicker. On the other hand, the time required to change images on the screen is determined by how fast scan conversion can reload the bit map. The larger the bit map, the slower this process is; thus, raster images with many pixels must trade off speed of interaction and cannot produce dynamic images. As the number of pixels increases, so does the rate at which information is clocked out of the bit map. The deflection speed of the CRT beam and the bandwidth of the CRT video amplifier must increase accordingly and will ultimately limit the number of pixels.

Characteristics of Color Graphic Displays

Two characteristics of color displays are particularly important to graphics users – display quality and information handling.

Display-quality factors are diverse, including optical characteristics such as resolution, edge sharpness, brightness, contrast, and color quality. In addition, temporal and spatial "noise" add such undesirable optical characteristics as flicker, jaggies, and moire patterns.

The second user-important characteristic – information handling – includes factors such as display size, number of vectors or pixels, number of colors, and interactivity.

You must bear in mind that the distinction between the two characteristics is not absolute. The distinctions are technical and arbitrary. The user's perception of image quality is strongly influenced by such "information-handling" factors as size, number of pixels, and the colors available. Because perception is so important, Jerry Murch of IDD is continuing his investigations of user perceptions.

Image-Quality Characteristics

Resolution strongly affects image quality. Because this is so, it is particularly important to precisely understand what resolution is – and is not. In discussions of raster displays, the term resolution is often used incorrectly as synonymous with the number of scan lines (addressability).

Resolution is the display's ability to resolve – that is, separate – two closely spaced points or lines.

Resolution is the essential characteristic that determines image sharpness. Resolution is independent of display size, but smaller displays need higher resolution than large displays to resolve an equal number of lines or pixels.

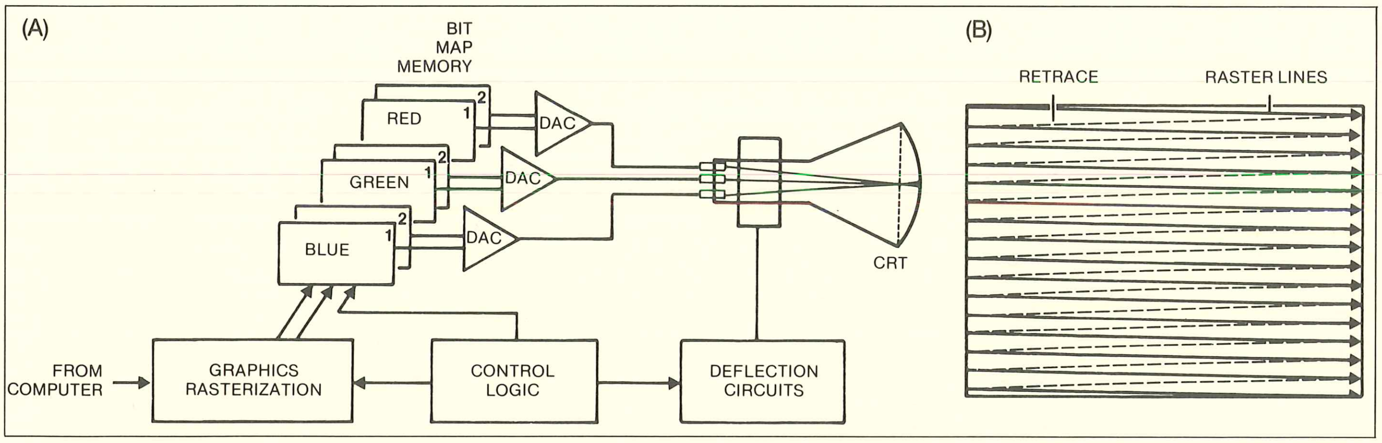


Figure 7. (A) Block diagram for a typical refresh-raster color display system. (B) Raster scanning pattern. Raster lines are written from top to bottom, with retrace a small fraction of the total field time.

Addressability, on the other hand, is the display's ability to position lines or pixels anywhere on the screen. A display's addressability can exceed its resolution; this will not affect the resolution of the display. However, if the addressability is not high enough, the resolution of complex images will suffer, since some image points will either not be presented or they will be misplaced on the screen.

The resolution of vector displays is primarily a function of the electron-beam spot size as vector-generated images consist of lines equal in width to the spot. (The current distribution in an electron beam usually is Gaussian and circularly symmetrical, as shown in figure 8.)

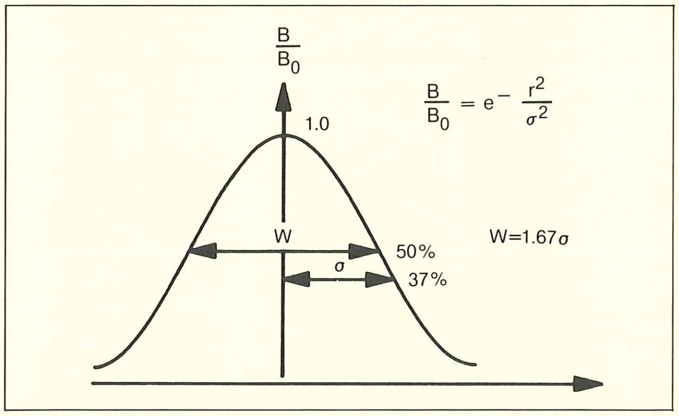


Figure 8. The Gaussian spot profile is the primary factor in image resolution.

Although several techniques are used to specify the resolution of displays, the modulation transfer frequency (MTF) method is the most definitive because it takes into account not only the spot size, but spot shape and the minimum spacing between spots too.

Images can be thought of as being made up of a set of line pairs of different spacings (spatial frequencies) and contrast, much the same as thinking of electrical signals as a set of sine waves of different amplitudes and frequencies.

The MTF is a measure of how well a display passes the different spatial frequencies in an image. MTF can be plotted as contrast versus spatial frequency. Obviously, at spatial frequencies where the MTF is large, there is good resolution. Conversely, the resolution is poor for spatial frequencies where the MTF is small. The profile of the MTF provides information about how sharp the line edges appear, and hence how good the resolution of the display is.

The method generally used to measure CRT resolution is to display a raster of lines, and then shrink the raster until the lines cannot be resolved. At this point, dividing the raster height by the number of lines yields a measure of how close two lines can be placed and still provide an acceptable image. But this method gives only the upper limit on the MTF profile – the shape of the spot must be known before the MTF can be derived. Figure 9 shows the line spacing obtained with this method for a Gaussian spot shape and an idealized spot shape, which plots as a rectangle. In practice, the spot is somewhere between rectangular and Gaussian. (Whatever the shape of the current-distribution plot, the perceived CRT spot is circular.)

In the case of the DVST, the target threshold for storage shapes the Gaussian distribution of the writing beam current into a more rectangular spot. This change produces a sharper line edge than refresh-vector displays can achieve. The result is a crisp edge and the *appearance* of very high resolution for stored-line-type images.

In both refresh-vector displays and vector-storage displays, resolution is improved by trading off brightness for a smaller spot because spot size is usually the limiting factor and lower beam currents lead to a smaller spot. However, it is possible to make these displays exceed the eye's resolution at normal viewing distances.

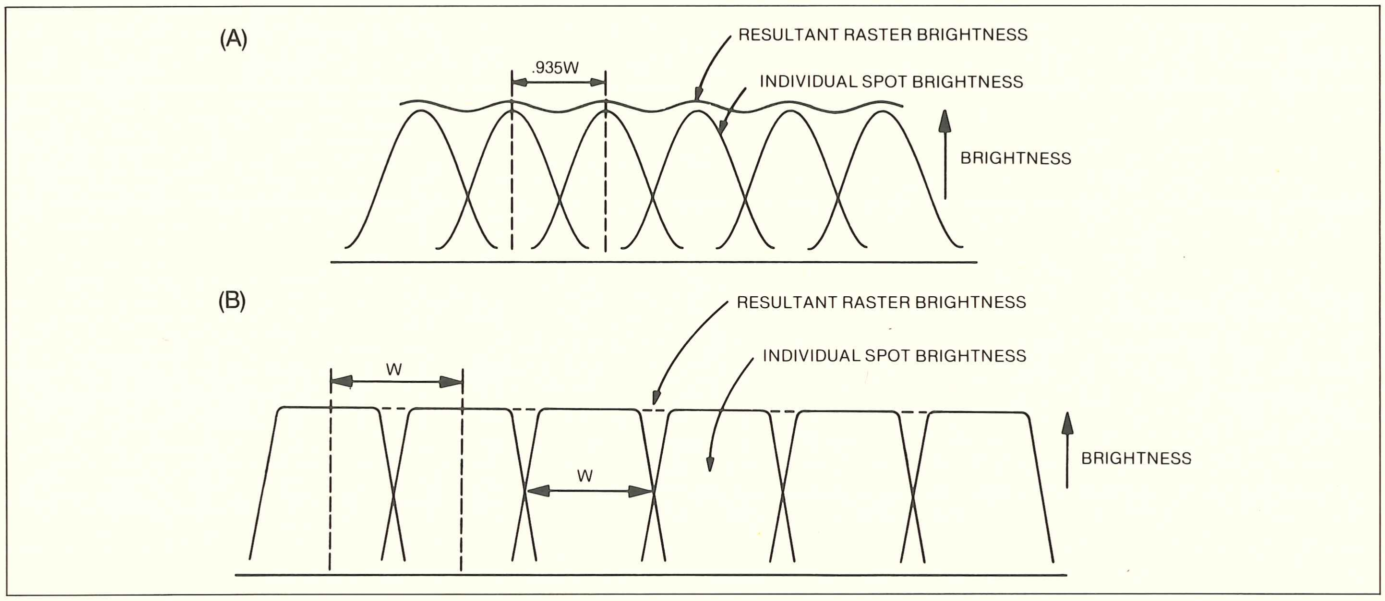

(A)
.935W
RESULTANT RASTER BRIGHTNESS
INDIVIDUAL SPOT BRIGHTNESS
BRIGHTNESS
(B)
W
RESULTANT RASTER BRIGHTNESS
INDIVIDUAL SPOT BRIGHTNESS
W
BRIGHTNESS

Figure 9. Scan-line spacing determined by the "shrinking-raster" method for (A) a Gaussian beam profile, (B) an idealized rectangular beam profile.

In shadow-mask CRTs, the resolution of color raster displays is determined not only by the electron-beam spot size, but also by the pitch of the shadow mask, the number of scanning lines, and the bandwidth of the video circuit.

In raster displays, the resolution in the vertical axis is a function of the spot size, the number of scan lines, and the spacing of the shadow-mask holes. The limiting effects of raster-line spacing and shadow-mask holes follow the Shannon sampling theorem. The scanning line-pair frequency must be at least twice that of the highest spatial frequency in the image to be presented; 2.5 to 3 times is optimum.[2]

If the raster line spacing is smaller than the shadow-mask pitch, then the mask pitch is the limiter. Conversely, if the spot size is larger than the shadow-mask pitch, then spot size will be the primary factor limiting resolution. Ultimately, the spot size limits the resolution of shadow-mask CRTs; the spot size is essentially determined by the shadow-mask pitch and is set so as to avoid image pattern noise caused by interference effects between the raster lines and the periodic pattern of the phosphor. A Gaussian spot size (full width at half maximum) of between 1.2 and 1.5 times the shadow-mask pitch is usually required. When the optimum spacing (for contrast) between raster lines is used, the number of resolvable scanning lines can be determined as will be seen later.

Resolution in the horizontal axis is determined not only by the spot size but also by the bandwidth of the video amplifiers, since the time between adjacent pixels is given by

$$t(\text{pixel}) = \frac{T(\text{frame})}{Nv\ Nh}$$

where T(frame) is the "active" frame time, Nv is the number of scan lines and Nh is the number of pixels per horizontal line. As the number of pixels (Nv × Nh) increases, so does the bandwidth required of the video circuits. To reduce the bandwidth required, some raster displays use a 30-Hz interlaced raster; however, this approach produces more flicker than a 60-Hz non-interlaced display.

Addressability – Good addressability is necessary for (but does not guarantee) high resolution and many vectors or pixels. Addressability is the ability to position a line or pixel to a given place on the screen. The vector displays have inherently high addressability, limited only by digital-to-analog (D/A) converters and noise. For example, a vector display capable of resolving 1000 lines may be able to position vector end points on a 4000 × 4000 grid. Generally, to assure a smooth line, the D/A converters limit addressability to about four times the resolution.

Raster displays are more limited in addressability due to the fixed pattern of the beam. The size of the bit map limits addressability because the number of pixels is limited. Useful bit-map size is determined by the number of scan lines and the bandwidth of the Z-axis video amplifier. The number of scan lines limits the addressability in the vertical dimension; the video bandwidth limits it horizontally. Increasing the number of pixels in either the vertical or horizontal dimension requires a corresponding increase in the pixel clock rate and a faster bit-map memory system.

Increasing the number of scan lines and pixels per line increases the addressability of the display system, but does not increase image resolution unless the spot size is reduced accordingly and the video bandwidth increased. Alternatively, addressability can be increased by increasing the display size rather than reducing the spot size; however, the video bandwidth still must be increased since bandwidth depends on pixels per line and not display size.

Image artifacts – While vector displays present relatively smooth lines, raster displays produce artifacts in addition to the desired image. These artifacts are unwanted images caused by the sampling effects of the raster pattern. This sampling produces high spatial frequencies on the display that do not exist in the intended image. If the viewer sees the higher-frequency artifacts (called aliasing), they appear as noise on any edges oriented other than vertically or horizontally. It is aliasing that makes a sloping line look like a staircase instead of a smooth line. This staircase is often called a "jaggie."

Moire – Moire is an artifact, produced by either the interference between the frequencies of image lines or the interference of the raster lines with the sampling frequency of the shadow mask. When the spacing of raster lines is close to the spacing of the mask, then brightness varies periodically across what should be a uniformly colored field unless the spot size exceeds the shadow-mask spacing by a sufficient margin. Generally, the spot size chosen is from 1.2 to 1.5 times the shadow-mask pitch, with the raster spacing about equal to the spot width. Resolution depends upon spot width, which is related to the mask pitch. Therefore, the shadow-mask pitch is the primary fundamental limitation to the resolution of a shadow-mask display.

Color Quality Characteristics

The quality of color includes brightness, contrast, purity, and convergence.

Brightness – The brightness (B) of a CRT is determined according to the following relationship:

$$B = \frac{k \propto IV}{\pi A}$$

where k is the attentuation factor due to the glass faceplate and the shadow mask, $\propto$ is the phosphor efficiency, I is the time average of the beam current, V is the electron-beam accelerating voltage, and A is the scanned or written area.

The brightness of the penetration CRT in a refresh-vector system can be quite high, limited only by the beam current and the current saturation of the phosphor. (The efficiency $\propto$ decreases with high beam current.)

The DVST with color write through, on the other hand, has limited brightness in both the storage and refresh mode. The stored-image brightness is limited because only the low-voltage flood electrons excite the phosphor. The color refresh image brightness is limited, although it is produced by the high-voltage writing beam, because the writing beam current must be kept low to prevent storage.

The brightness of a color raster display, although quite good for low-resolution CRTs, is limited by shadow mask interception of about 80 percent of the beam current. The fact that there are three beams partially compensates for this loss.

Contrast – Two mechanisms limit the contrast of CRT displays: intrinsic and extrinsic. Intrinsic contrast is the contrast of the written parts of the image relative to the unwritten parts as measured in a dark room. The intrinsic contrast, Ci, can be defined as the ratio:

$$Ci = \frac{Bw}{Bunw}$$

where Bw and Bunw represent the brightness of the written and unwritten parts of the screen, measured in a darkened room. This contrast is quite high for both the refreshed-vector and raster color displays. The DVST has low intrinsic contrast because the unwritten areas of the target receive some excitation from the flood guns.

The other type of contrast – extrinsic contrast (Cex) relates better to the "real-world." It is given by:

$$Cex = \frac{Bw + R}{Bunw + R}$$

where R is the reflected and scattered ambient light off the phosphor and screen surface.

Because all three types of color displays reflect and scatter about the same amount of room light, their contrasts are primarily determined by their brightness. Extrinsic contrast can be improved by placing a filter in front of the display screen. Such a filter attenuates emitted light once while reflected light must make a "double pass" and is therefore attenuated twice.

More effective are selective filters designed to absorb room light while transmitting the light emitted from the display. The CRT itself can have either an "anti-glare" coating applied to the front surface or a matte finish to prevent specular reflections.

Color Purity and *Convergence* – Color purity generally refers to the uniformity of color over a large area of the screen. Purity is a measure of whether or not the primary colors are spectrally pure.

Purity is not much of a problem with the penetron and the DVST/CWT, but it can be a problem in the shadow-mask CRT. To produce pure color, each of the three beams should excite its entire phosphor dot – and only its dot – when the beams pass through the shadow-mask hole properly.

In a shadow-mask CRT, if some electrons meant for the red dot impinge upon the green dot, then the red is not pure. Slight changes in the shadow-mask position due to manufacturing tolerances or thermal distortions can cause one or more beams to miss parts of their respective color dots. The resulting differential loss at one triad causes the purity problem.

Convergence, on the other hand, is a measure of whether or not each primary color image is in perfect registration with the other two primary color images. Convergence usually differs from place to place on the display screen.

Misconvergence in penetration refresh-vector displays occurs because the electron beam operates at different energies; registration requires different deflection currents. Such deflection-factor matching is difficult because of hysteresis in the deflection yoke and the random nature of the deflection currents, among other things. Convergence is not a problem with the DVST/CWT displays due to both the limited nature of their color and because all colors are written by the same beam.

Misconvergence in shadow-mask CRT raster displays is caused by the fact that the three color guns pass through the deflection yoke differently and are thus exposed to slightly different deflection fields. The error is a non-linear function of the deflection current, and is difficult to eliminate. A compromise is usually accepted. Both static and dynamic corrections are used. These must be periodically adjusted because of circuit drifts and other factors.

Several graphics terminals provide the user with a keyboard convergence adjustment, but until convergence adjustment is automated, or its need eliminated, the user will continue to be inconvenienced by this annoying characteristic of shadow-mask CRT displays. (The color display of the 4115 uses a feedback method to automatically adjust convergence.)

While convergence accuracy on the order of 1/4 pixel is desirable for good image quality, most displays suffer from more than one pixel worth of convergence error somewhere on screen. As the resolution of the shadow mask is increased, the problem of misconvergence becomes worse and elaborate correction circuits and procedures will be required.

Image Information Handling Characteristics

Size – The maximum size for graphic displays using penetration CRTs, DVST, and shadow-mask CRTs is about 25 inches (635 mm) diagonal. The penetron and DVST can also be made quite small (6 inches, or 152 mm, diagonal) and, by having a small spot, still provide many vectors. The number of vectors in a refresh-vector system is not limited by the resolution, but by the deflection speed required to write vectors at a flicker-free rate. To display many refreshed vectors, the deflection system must have very high bandwidth (at the expense of power). The DVST avoids the need for high power by storing many vectors, but faces the same trade off for refresh color write-through vectors.

The shadow-mask CRT, however, encounters serious trade-offs when attempting to make the size either quite small or quite large. Since the shadow-mask pitch sets the spot size, which then determines the resolution, smaller CRTs must have a smaller mask pitch to present the same number of resolvable raster lines as the larger CRTs. There appears to be an ultimate limit to how small the mask pitch can be made: about 0.2 mm for CRTs 13 inches (330 mm) diagonal or smaller.

As the size of the CRT goes up, it is difficult to maintain a small pitch in the mask; practical, cost-effective 19-inch (483-mm) diagonal CRTs probably will not have a shadow-mask pitch smaller than 0.25 mm, although some Japanese manufacturers claim that 0.21 mm is possible. Currently, 0.3-mm-pitch shadow-mask CRTs are commonly used for high-resolution 19-inch (483-mm) graphics displays.

The most resolvable pixels that can be displayed is determined by translating the mask pitch into the required spot size – about 1.2 times the mask pitch – then dividing the line spacing (approximately equal to the spot size) into the "usable" vertical and horizontal dimensions of the CRT. Figure 10 shows how the number of pixels varies with the size of the CRT. It can be seen that a shadow-mask CRT of any size is not likely to be capable of displaying more than about 1150 × 1500 resolvable pixels.

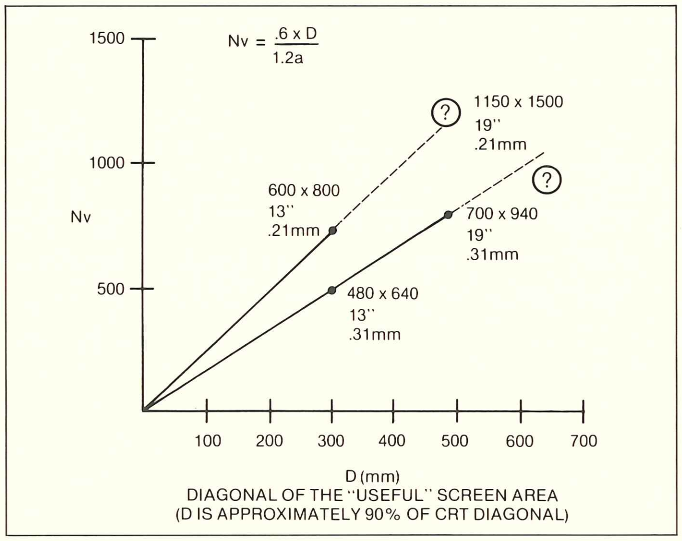


Figure 10. Number of resolvable pixels in shadow-mask displays as a function of screen size. Displays beyond 1000x1400 resolvable pixels are unlikely.

Number of colors – The penetron CRT is limited to at most three distinguishable colors; the DVST with CWT also is limited to three colors – but both are typically used as two-color tubes. Only the shadow-mask CRT offers a full range of colors. Figure 11 shows the color gamut of each of the three types on the CIE color diagram. The color gamut of the shadow-mask CRT is pretty well determined by the availability of phosphors for the three primary colors (red, green, and blue). These color gamuts will be reduced in the presence of high ambient lighting.

Summary and Future Possibilities

Of the three types of color CRTs used for graphics, the DVST and shadow-mask raster dominate.

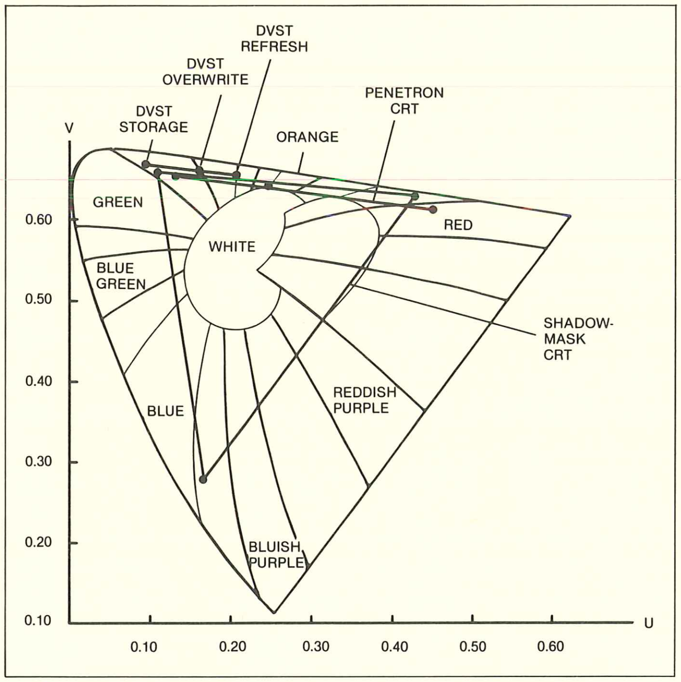


Figure 11. The color gamut for penetron, DVST, and shadow-mask CRT displays. All but the shadow-mask CRT produce *limited color* displays.

Although the penetron refreshed-vector display is found in some specialized graphic displays – such as avionics displays – it isn't used much in computer graphics because of its cost and severe limitations. The penetron can present only a few colors, and it is inherently difficult to converge.

The DVST with color write through has been found to be very useful in displaying complex images in which color is needed only to highlight areas of the display.

The shadow-mask raster display is by far the most prevalent type of color display in computer graphics applications. Although higher resolution and better convergence are desired, the primary industry emphasis seems to be on better visual ergonomics and better interactivity. In raster displays, it is the frame buffer that limits the interactivity, that is the updating and changing of images.

In the future, a few new technologies might significantly improve resolution, convergence, and size. Let's look briefly at these next.

Limited Color Displays

Two alternatives to the penetron look very promising. These should allow highly interactive, high resolution, multicolor images consisting of refresh vectors (raster displays are also possible).

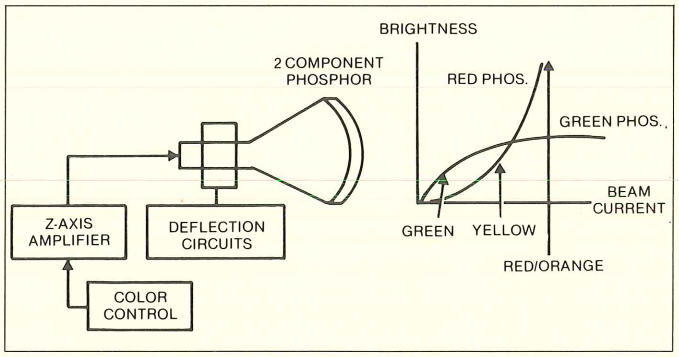


Figure 12. Current-sensitive color CRT. Physically similar to the penetron, this CRT differs by selecting color by current level changes rather than by changing accelerating voltages.

The first alternative, being developed by SONY, is the *current-switched color CRT*[3] shown in figure 12. Like the penetron, this limited color display uses a mixture of two phosphors; each produces a different color. Because the two phosphors differ in current-saturation characteristics, as beam current increases the displayed color changes from red to green. The red phosphor saturates, and all further light comes from the green phosphor. As in the penetron, the spot size can be small; however, since the electron-beam accelerating voltage is identical for both colors, there is no problem with registering colors.

A continuum of colors can obtained by beam-current modulation. This modulation can be performed a point at a time or field-sequentially. However, the current-switched color CRT, like the penetron, produces a a narrow-range of colors. Another shortcoming is that color and brightness are difficult to control independently.

The second alternative to the penetron, being developed here at Tek, also employs a simple, one-gun "monochrome" CRT (see figure 13). It is a field-sequential color display that uses a liquid-crystal color switch[4,5]. A monochrome CRT with an unpatterned multicolored phosphor is placed behind an electrically controlled color filter. By synchronizing the red information written on the CRT during field number one with a color switch set to pass only red light, and likewise the green information in field number two with the switch passing green light, a two-primary field sequential color display can be produced.

Writing information in both fields produces yellow. Varying the electron beam current in each field makes any color combination of the two primaries possible. There are no convergence problems, and the resolution is limited only by the size of the spot.

A limitation is a somewhat limited viewing angle, which makes it most suitable for use by just one or two viewers.

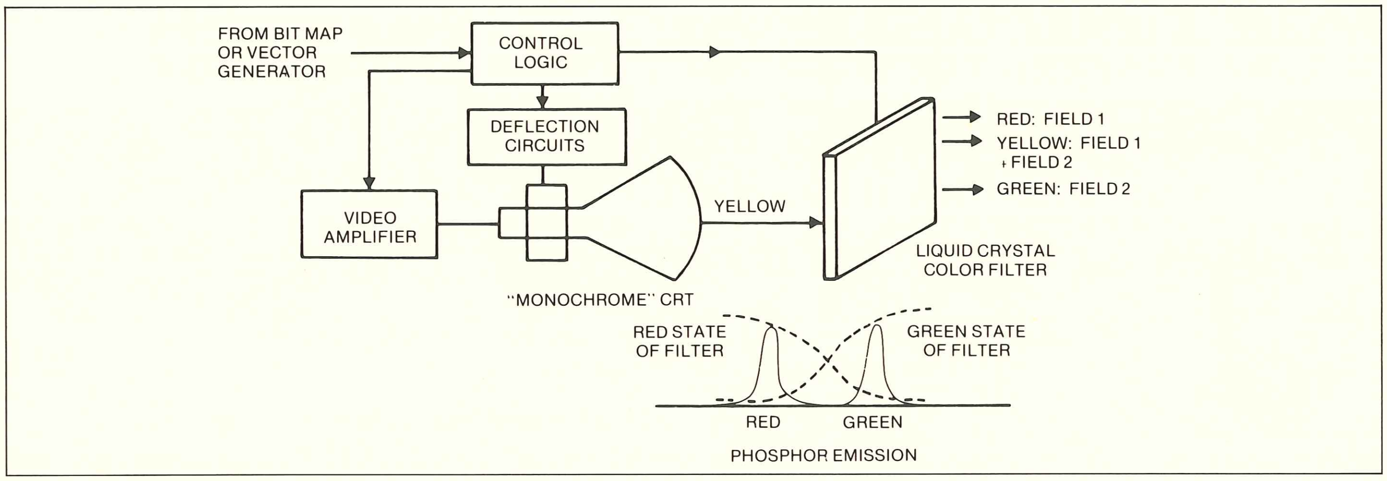


Figure 13. Block diagram of a liquid-crystal-switched field-sequential color display.

Full Color Displays

Optical projection systems are another possibility for producing large, high-resolution color displays. Such systems converge the output from several single-color projection displays. A light valve may be used to gain sufficient brightness. The liquid-crystal light valve[6] (shown) in figure 14 is an example of such a device.

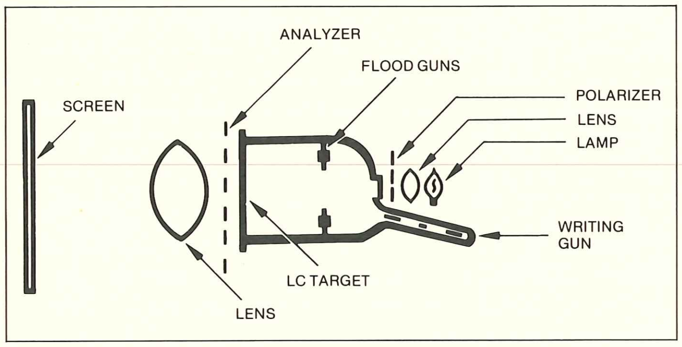


Figure 14. A liquid-crystal light-valve projection system.

The low-current electron beam writes a charge on a liquid-crystal target in the CRT. Polarized light is projected through the light valve and an analyzer to the screen. Three such light valves would provide a full-color display without the size and resolution limits of the shadow-mask CRT.

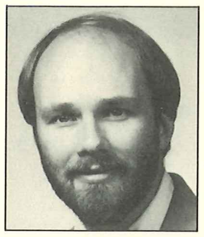
Conclusion

Clearly, the shadow-mask raster display dominates the color graphics market today. The good image and interactivity inherent in this technology have gained it wide acceptance. However, color does not replace resolution in all applications, and the demand for both full color and very high resolution is pushing shadow-mask technology to its limits. Although alternatives providing higher resolution are emerging, it is not yet possible to predict what technology will replace the shadow-mask CRT. It appears, however, that the liquid crystal color shutter display being developed by Tek has the best chance of all the contenders. □

References

1. R.H. Anderson, "A Simplified DVST," *IEEE Transactions on Electron Devices,* Vol. ED-14, No. 12, December 1967, pp. 838-844.
2. P.G.J. Barten, "Optical Performance of CRT Displays," *Proceedings of the First European Display Research Conference,* September 1981, pp. 160-165.
3. A. Ohlcoshi, O. Takeuchi, H. Kusama, K. Kamboyashi, and T. Yudawa, "Current-Sensitive Multi-Color CRT Display," *Record of the 1982 International Display Research Conference,* October 1982, pp. 64-65.
4. Michael G. Clark and Ian A. Shanks, "A Field-Sequential Color CRT Using a Liquid-Crystal Color Switch," *SID Symposium Digest,* May 1982, pp. 172-173.
5. R. Vatne, P. Bos, and P. Johnson, "A New LC/CRT Field-Sequential Color Display," presented at 1983 SID Symposium, May 1983.
6. D. Haven, "Electron-Beam-Addressed Liquid-Crystal Light Valve," *Record of 1982 International Display Research Conference,* October 1982, pp. 72-75.

THE NEW WAVE OF AUTOMATED PROGRAMMING TOOLS



Norm Kerth is a member of the Software Center, part of the Computer Science Group. He joined Tek in 1974, spent time in the Communications Division and Computer Research Laboratory before moving into the Software Center. He was educated at the University of California, Berkeley where he received a BS in electrical engineering and computer science. While studying at Berkeley, he worked in one of Hewlett-Packard's software quality assurance groups. He has lectured at several universities on programming methodologies.

This article presents an overview of automating the design and implementation of software. This material was prepared for Northcon.

The increased number of computers has caused a severe shortage of trained software professionals. The U.S. Department of Defense[1] estimates the gap between demand and supply is between 50,000 to 100,000 software professionals. Furthermore they suggest that the gap will widen to between 860,000 and 1,000,000 software professionals by 1990. See figure 1.

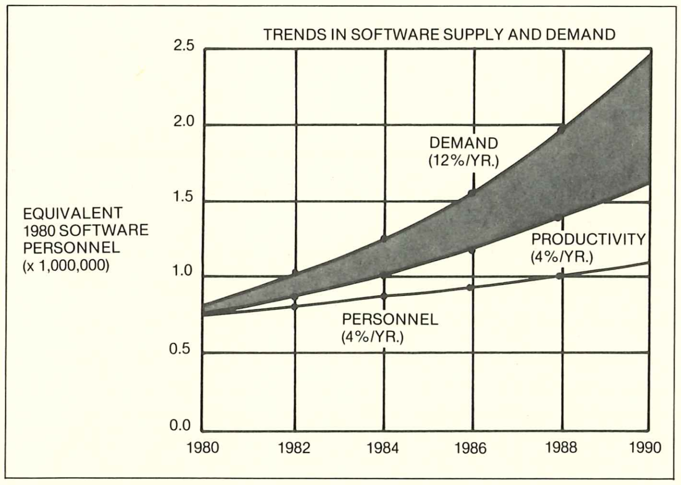


Figure 1.

Clearly, the United States will not have enough software professionals. So the professionals we now have must become more productive. Productivity can be increased by improved management techniques, appropriate software engineering methodologies, better eduction, and by better tools. This article will focus on increasing programmer productivity through new tools.

Over the past 30 years, the computer industry has discovered a number of tools that increase a programmer's productivity. The invention of the compiler provided one of the early productivity increases. More recently the UNIX[2] system, with its many system utilities and its unique way of connecting them together, provides significant time savings for many programming applications.

Each of these discoveries provided a way for programmers to work at a higher level of abstraction, that is, in a more problem-oriented domain where they could ignore implementation details. Future productivity increases will result from even higher levels of abstraction and from automating more of the tedious aspects of programming.

Three Automation Approaches

This article discusses three approaches to automating the design and implementation phases of the software development task: (1) employing prebuilt software components, (2) employing program generators, and (3) directly executing program specifications.

Software Components

Using previously built software components is a simple way to avoid some design and implementation effort. A small industry has sprung up to supply these components for the more popular microprocessors. A software component is a routine designed to solve a common software task. The component has documentation to aid its integration into a particular application, as well as extensive performance data and usually some test cases to show that the program works. Components range from simple math routines such as FPAK[3] to real-time operating systems such as MTK, VRTX, and MTOS[4-6]. Typically, software components are basic functions at the lowest levels of many microprocessor-based products.

There are several difficulties that must be solved before a large market for software components can develop: user differentiation between similar products; assuring that the component works correctly; and determining whether the component meets one's needs (for example, is a real-time operating system fast enough for a certain application?). The software-component industry must establish industry-wide standards for documentation, interfacing, data structuring, and performance assessment. Ideas for solutions to these problems can be found by studying the creation of the integrated-circuit industry.

Some order may be brought to the software-components industry when the new Department of Defense language Ada[7] becomes available. The Ada language has a special mechanism called *packages* that allows a programmer to describe what a software component will do and to describe an interface without showing how the component is implemented. The idea is that the package "encapsulates" the data used along with the routines that modify that data in one unit that is hidden from the user. Programmers can freely use the component but, since they do not know how it is implemented, they cannot abuse the component by exploiting its implementation. Ada also provides a *generic* feature that enhances the software component approach by allowing one algorithm to be used for different types of data.

While employing software components can significantly increase productivity, few programmers have tried it. One blockage is an aspect of the not-invented-here syndrome – programmers shy away from using someone else's code because reading and debugging foreign code is painful. They are also concerned that the code will not exactly fit the specifics of the project or that it will be so general as to be inefficient. (We found that programmers who are the most receptive to using software components are those with an unreasonably short schedule.)

The use of software components has the potential for large productivity increases. To this end, a number of researchers have extended the idea to include tools that not only keep track of previously built software components but that also support software design by encouraging programmers to decompose a program into a small set of subprograms. Decomposition is done in such a manner that, if all the subprograms are finished, then the overall program is completed. This is using the mathematician's strategy of reducing the problem to a *previously solved problem.* If a designer reduces a program into subprograms and those subprograms have already been completed, then the tools notify the designer that further decomposition is unnecessary.

Two systems employing the "previously solved" strategy are USE.IT[8] and SARA[9]. Each of these systems is tightly coupled to a particular design methodology to increase productivity by guiding the designer to work in a disciplined manner. Since these methodologies are based on mathematics, these tools can partially verify that the decomposition of a program into subprograms was done correctly. While these tools will not find all design errors, they nicely complement the human review; the computer-based tools doing the tedious repetitive checking while the person looks for conceptual errors.

Systems like USE.IT and SARA are not as popular as they should be, because the methodologies connected with these tools take quite a bit of practice. Before the value of the system is appreciated, the typical programmer is likely to give up trying to master the methodology. However, as more disciplined methodologies[10] are taught in universities, this problem will be lessened.

Even if a tool is not used to keep track of software components, it can still significantly increase productivity. Early work suggests software components can reduce the software development cycle by up to 50 percent.

Program Generators

The second type of radical productivity increase can come from program generators. A program generator accepts a description of how a program is to behave and then produces the program. A compiler compiler is an example of this idea. The syntax of a language is fed into a compiler compiler, and a syntax parser is produced[11]. Menu generators and database-report generators are other examples of tools that fall into this category[12].

Presently, these tools are used only by a few experts. Database experts use report generators, and compiler designers make good use of compiler compilers. However, these tools can aid all sorts of software development beyond the intended use. For example, a compiler compiler can rapidly generate the human-interface portion of many software applications or generate portions of network protocol software! The applications are limited only by the user's imagination and the knowledge of how to use these tools.

Program generators are best used for a class of problems that are well understood and whose variations can be described by a formal language. Software components are not radically different from program generators; in fact, one can argue that some simple program generators are nothing more than a decision tree that helps a user find the right reusable software component. A better way to view these tools is as a continuum. Advanced software-component systems begin to look like program generators. Likewise, the distinction between advanced program generators and executable program-specification systems is hazy.

Executable Program Specifications

Engineering research

The expert-systems and knowledge-based engineering research in the artificial-intelligence field has lead to impressive demonstrations of executing a user's specification. Specifications are constructed in a rigorous language or in English and a program evaluates the specifications and, where the specification is unclear, asks the user for more information. This process continues until the system can execute the specification. One example of this tool is the PSI Program Synthesis System[13].

The PSI system is composed of two phases: the acquisition phase, which interactively acquires the high-level specification of the program, and the synthesis phase, which produces an optimized program. Figure 2 shows a dialogue between the PSI system and a user who is developing a program named CLASS. The program will input a sample set of items (for example, a list of job qualifications). Then the program repeatedly inputs a trial set (for example, applicant qualifications) and prints "fit" or "didn't fit" accordingly. The program terminates when the word "quit" is typed in place of the trial input.

During this dialogue, the acquisition phase developed a program model that serves as input to the synthesis phase. The synthesis phase transforms the program model into a legal program. The present system produces LISP code, but research has shown that a block-structured language such as Pascal could be generated instead. The synthesis phase also performs program optimization using cost-of-execution time and memory size as decision factors. The optimization is extensive; even the data structures are analyzed during the optimization phase (for example, to determine whether an array or a hash table is more efficient for the particular task).

PSI:	What is the name of the program you want to write?
USER:	CLASS
PSI:	Does CLASS have a special program type?
USER:	Classification.
PSI:	Please describe CLASS.
USER:	CLASS first inputs a concept. Then it repeatedly accepts an input scene from the user, classifies it based on whether or not it fits the concept, and outputs this classification to the user.
PSI:	What is the structure of the concept?
USER:	A concept is a collection of relations. Each relation is a string.
PSI:	What is the exit test of the loop?
USER:	Stop when the user types "quit" instead of a scene.
PSI:	What is the structure of the scene?
USER:	The same as concept.
PSI:	Please define what it means for a scene to fit a concept.
USER:	A scene fits a concept if all the relations in the concept are in the scene.
PSI:	What is the format of the classification output?
USER:	Output "fit" if the scene fit, "didn't fit" if it didn't.
PSI:	Are we finished?
USER:	Yes.

Figure 2. Research in the artificial intelligence field has lead to impressive demonstrations of executing user specifications. In this dialog between the PSI system and a user the process develops an executable specification (from reference 13).

While this work is still in a research mode, enough is known for to begin applying this technology to certain problems.

Conclusions

There is a large shortage of software professionals. The need is so large and is growing so fast that the demand cannot be met just by training more people. Instead, software professionals must find ways to be more productive. Firms should start pilot projects to demonstrate the viability of using software components in their particular applications and measure the effects on productivity.

In many cases, program generators can be used for unique applications quite different from their intended use. However, this will only happen if the average software engineer has experience with these tools; creative training is needed.

Finally, we recommend that the expert systems and automated programming work in the artificial-intelligence field be closely monitored and explored by industry's applied research groups.

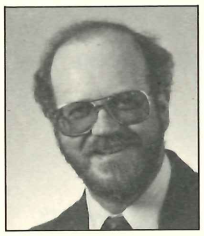
For More Information

For more information, call Norm Kerth, ext. 627-5379. □

References

1. "Strategy for a DOD Software Initiative," *RADC/ISIS,* vol. 1, p. 6, Griffiss AFB, NY 13441.
2. UNIX is a trademark of Bell Laboratories.
3. FPAC – United States Software Corporation, Portland, Oregon.
4. MTK – United States Software Corporation.
5. VRTX – Hunter & Ready, Palo Alto, California.
6. MTOS – Industrial Programming, Inc., Jericho, New York.
7. Ada is a trademark of the United States Department of Defense.
8. Allen Razdow, Ron Hackler, and Richard Smaby, "Automatic Code Generation Steps up Software Productivity," *Electronic Design,* Dec. 23, 1982.
9. Maria Heloisa, "The Use of a Module Interface Description in the Synthesis of Reliable Software Systems," Penedo, UCLA Report No. CSD-810115, January, 1981.
10. David Gries, *The Science of Programming,* Springer Verlag, October 1981.
11. Stephen C. Johnson, "YACC: Yet Another Compiler-Compiler," *UNIX Programmer's Manual,* Seventh Edition, vol. 2B, January 1979.
12. James Martin, *Application Development Without Programmers,* Prentice-Hall, 1982.
13. Cordell Green, *et al.,* "Results in Knowledge Based Program Synthesis," in *Proceedings of the Sixth International Joint Conference on Artificial Intelligence,* Tokyo, August 20-23, 1979, vol. 1, pp. 342-344.

PLUGGING PASCAL SUPPORT HOLES WITH HIGH LEVEL DEBUGGING AIDS



Charles Montgomery is a product engineering manager in MDP, part of DAD. Charlie joined Tek in 1976 from the Lawrence Berkeley Lab. Before that he developed compilers at Control Data. Charlie holds a BS from Oregon State University.

Pascal's prominence in microcomputer software design is not accidental. As a high-level language, Pascal offers considerable machine independence, allowing programming skills to be readily transferred from one processor to another. As a structured language, it easily accommodates the "top down," or structured design and implementation approach, which has become a vital tool in organizing and controlling today's large software projects.

Yet despite these formidable assets, conventional Pascal is, figuratively speaking, full of holes in the support it extends to the microcomputer software engineer. During several phases of the microcomputer software design cycle, the programmer must resort to assembly code or complex system command files to get the job done. Often these "non-Pascal" interludes consume much labor and become a major obstacle to improving design productivity.

To close these costly gaps in high-level software design support, MDP developed a new approach called the Pascal Language Development System (LANDS). Pascal LANDS is the first high-level microcomputer software design support to extend through the entire software design cycle, from initial source code editing to the final debugging in the prototype hardware environment.

This article will show how LANDS supports each major phase of microcomputer software development, including editing, compiling, integrating and debugging.

Language Directed Editor Cuts Recompiling

Since Pascal is a compiled language, syntax errors entered during initial source code editing can ultimately be quite costly in terms of time. In a conventional software development system, syntax errors creep undetected into the source file, where they wait to wreak havoc during compilation. Once the error is flagged by the compiler, the error must be located in the source code, modified, and a new source file reflecting the repair must be compiled all over again (figure 1). The result is a considerable drain, both on human and system resources.

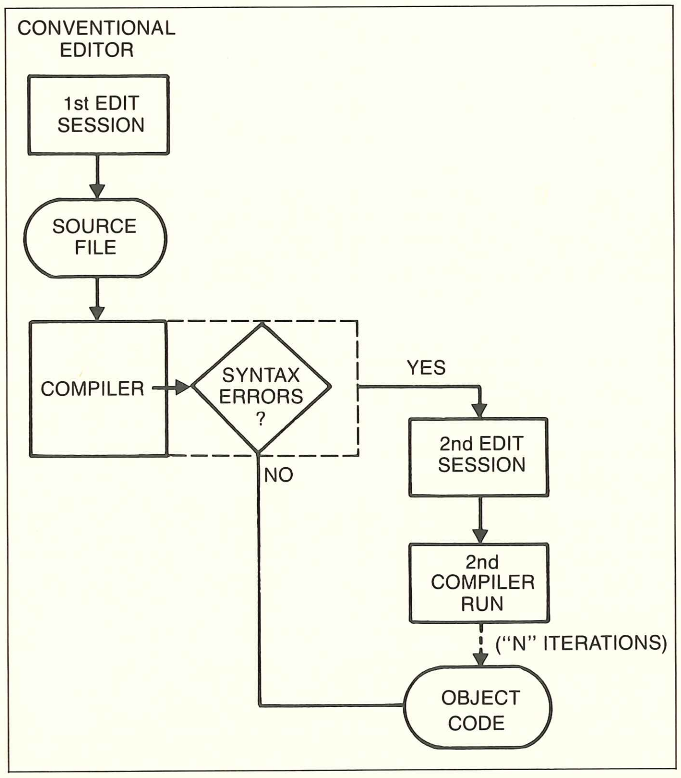


Figure 1. Because conventional software development systems allow syntax errors to creep undetected into the source file, conventional editors such as this are a time-consuming necessity.

The LANDS Language Directed Editor (LDE), part of LANDS, eliminates source code errors long before they reach the compiler. LDE has a built-in understanding of Pascal syntax and flags any syntax-related errors during source code entry and editing (figure 2). Our studies show that LDE eliminates up to 20 percent of initial coding errors and reduces later code changes by more than 50 percent. For user convenience, the syntax check can be performed at either the line level, the procedure/function level, or the program level. When an error is flagged, the user receives an immediate prompt message locating and explaining the error.

The LDE includes other Pascal-specific benefits. Program lines are automatically indented to their correct position to reflect program structure. Also, there is extensive "cut and paste" capability, which allows blocks of text to be easily moved, deleted, and

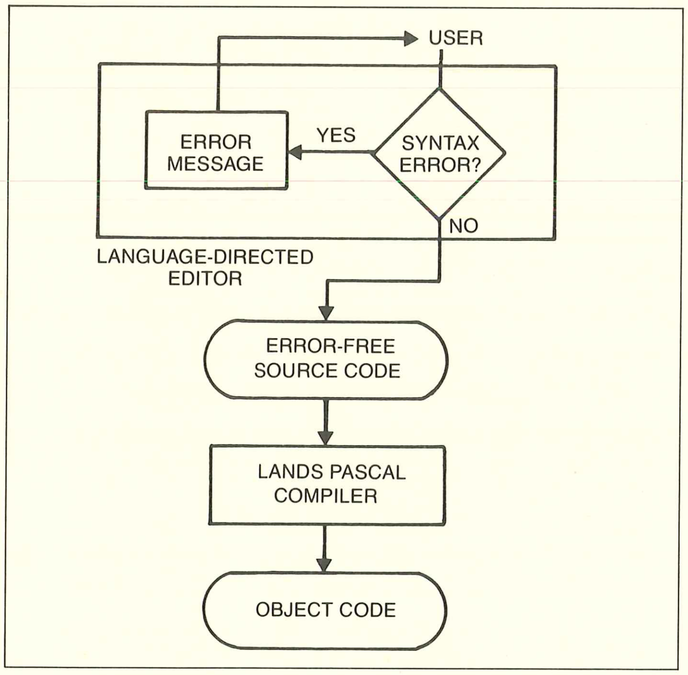


Figure 2. The LANDS Language-Directed Editor (LDE) "understands" Pascal syntax. The LDE can check syntax at the line, procedure/function, or program level, thus the editor can eliminate some initial coding errors. Later code changes are cut in half.

copied. Since LDE is screen-oriented, the user enjoys easy graphic interaction during editing. In addition, LDE allows all Pascal key words to be abbreviated, which reduces the chances of misspelling them during entry.

LANDS Compiler Targets on μC Design

Most Pascal compilers are excellent for producing code for channeling high-level data flow, but lack the enhancements for controlling the machine-level operations vital to many microcomputer applications. And they also lack the features needed to support the gradual transfer of I/O functions from the microprocessor development system emulation environment to the prototype hardware.

The LANDS Pascal compiler fills these two support gaps through enhancements aimed at team-design of microcomputer software. The LANDS compiler permits modular compilation; a large program can be divided into a series of modules, and each module coded and compiled separately. A module can contain both procedures and functions that transfer data between modules and the main program. This allows procedures and functions to be put into modular groups that form a top-down hierarchical approach to software planning and organization. This approach, also known as structured design, is a powerful tool when properly applied to large programming efforts involving many team members.

Pascal LANDS includes enhancements aimed at hardware-level code manipulations. These include the ability to manipulate data at the bit level through boolean functions, and to directly assess I/O ports. Also, through the enhancements, any variable can be assigned an absolute location in memory, not subject to modification at link time. In addition, interrupt processing procedures can be written in Pascal.

The LANDS Pascal compiler also has sophisticated provisions for simulating I/O during the early debugging, when the software execution is being emulated in the development system environment. During this state, the program may reside in the debug system memory and execute on a processor identical in function to the one targeted for the prototype. Because there may not yet be a physical connection to prototype hardware, there are provisions to simulate I/O operations using development system resources, such as the terminal, disk or printer.

From a software standpoint, I/O simulation requires that temporary code be inserted for simulated I/O, later to be replaced by code that services the actual prototype I/O. The modularity of Pascal LANDS allows all simulation code, such as procedures and functions, to be grouped into one or more modules, that are later replaced with module(s) containing the actual prototype I/O code. This modular method is much less cumbersome than "patchwork" insertions at locations scattered throughout the program.

The LANDS Pascal compiler includes an optimizer, which is optionally executed. It uses sophisticated optimization techniques to reduce the number of instructions in the compiled object code. The result is a program that requires less execution time and memory space, which are both areas of critical concern in many microcomputer design projects. In typical test cases, we were able to reduce code by as much as 37 percent.

Integration Control System Interfaces Software, Hardware

Although a Pascal compiler produces code targeted to a specific processor, it does not complete the transition from the high-level language to code executable in the prototype hardware environment. First off, the compiler can't take into account the hardware options associated with contemporary 16-bit microprocessors. For instance, a processor might have an optional co-processor to handle floating point operations, or address large or small data spaces. All these options must be reflected in the run-time support library, with the appropriate modules called in at link time, which involves generating a complex linker command file.

Also, the linker command file generated by the user must include information that places the object code correctly within the prototype memory map, with instructions and constants in their proper ROM spaces, and global variables, heap and stack in RAM. In addition, some applications call for certain vectors to be copied from ROM to RAM during initialization. In any case, compiler-imposed requirements of this sort make the linker command sequence increasingly complex, laborious, and much more prone to error.

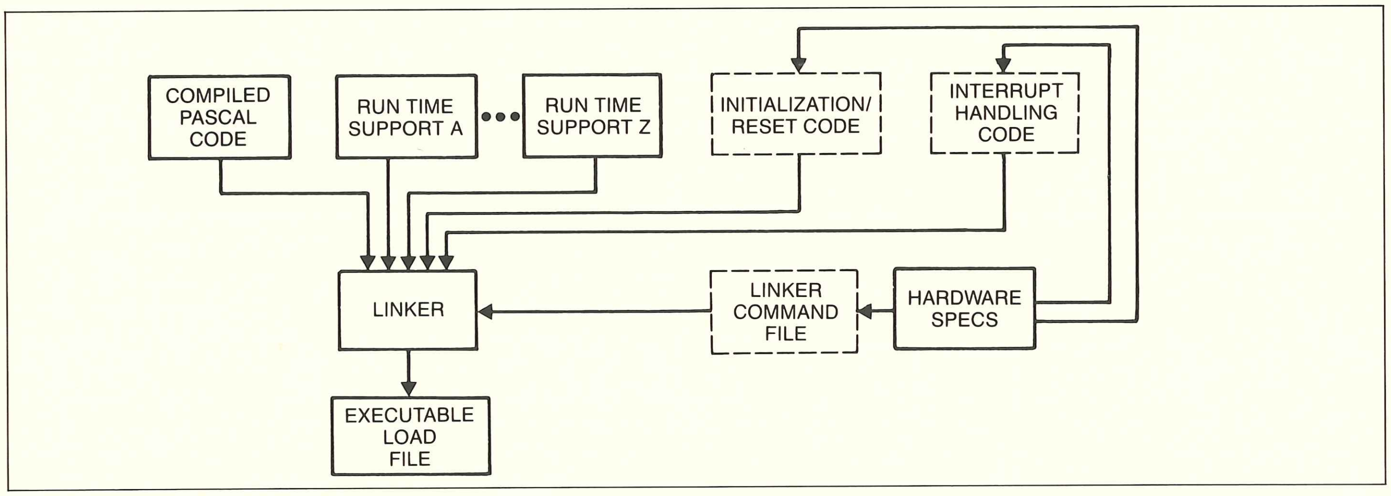


Figure 3. At link time, the user must place the object code in the prototype's memory map. The old-style linker command file does this partially with complex user-supplied data. Some applications require certain vectors to be copied from ROM to RAM during initialization. In this figure, the blocks drawn in solid lines represent existing software; broken lines represent user-furnished software. Contrast this time-consuming, laborious process to the improved process shown in figure 4.

Besides linking requirements, the user must generate assembly code that handles hardware-specific operations such as initialization, reset and low-level interrupt handling. Initialization/reset code usually involves accessing and clearing individual registers within the processor and other devices, and therefore can only be accomplished through assembly code. Interrupt handling involves the hardware's initial response to the interrupt assertions, and therefore requires assembly code to perform such tasks as identifying the interrupt source and vectoring to the proper handler. The interrupt handler connects the interrupt to the proper high-level interrupt service routine. Finally, all this machine-specific assembly code must be properly linked to the Pascal program. Figure 3 summarizes the software situation that exists at link time.

The Pascal LANDS Integration Control System (ICS) uniquely solves the problems posed by microcomputer software/hardware interfacing. With the ICS, the user's input is reduced to responding to a simple menu or creating a brief file with the system editor. Figure 4 shows how software/hardware interfacing would proceed using the ICS. Once the system editor or ICS menu is used to create an ICS source file, the ICS Processor uses this information to automatically perform all other tasks necessary to complete the interface. These tasks include generating hardware/reset object code, interrupt handling code, and the linker command file that designates the proper runtime support and governs memory mapping.

Using ICS reduces integration time by 80–90 percent. The key feature of the ICS is a simple user interface, that can reduce software/hardware interfacing tasks from days to a few minutes. The ICS source file can be created via the ICS prompter, where a simple menu lists the parameters essential to creating the interface spec (figure 5). This menu prompts the user to define all hardware interface information necessary for an 8086-based design, with prompts on the left and user inputs on the right. In this example, the hardware configuration is first defined, followed by memory mapping information. Next, the reset is affirmed and service calls for I/O simulation are ruled out. Next, the Pascal code is identified and special library and file handling are ruled out. Interrupt vectors are then placed in ROM, and identified by type number, and floating point save on interrupt is ruled out in all cases. Next the processor is instructed to halt when presented with an unspecified interrupt, and a vector for the specified interrupt is indicated. Finally, the symbolic label for restarting the code is defined.

The information entered through the ICS prompter menu is automatically converted into an integration source file. (Or, if the user chooses, this same file can be created directly through interaction with the system editor.) The ICS processor now uses the integration source file as input to implement all aspects of the software/hardware interface. These include all object code necessary for initialization, reset and low-level interrupt handling. ICS also produces a linker command file (which controls the linkage of compiled object files), run-time support files, and ICS-produced object files into an executable object file conforming to the prototype memory map. As a result, the user doesn't need to write complex linker command sequences or involved assembly code for machine-level operations.

Debug Elevated into Pascal

Once the Pascal code has been converted into an executable load file, it is ready for debugging. In a typical microcomputer software design environment, debugging is accomplished through real-time emulation, which uses a processor identical in function to the one targeted for the prototype. The emulator processor executes the code under the control of the development system's debug software. This execution allows the user to observe how the code will behave when executed by the target hardware. Observing is done by setting "breakpoints," which halt the code's execution at points specified by the user.

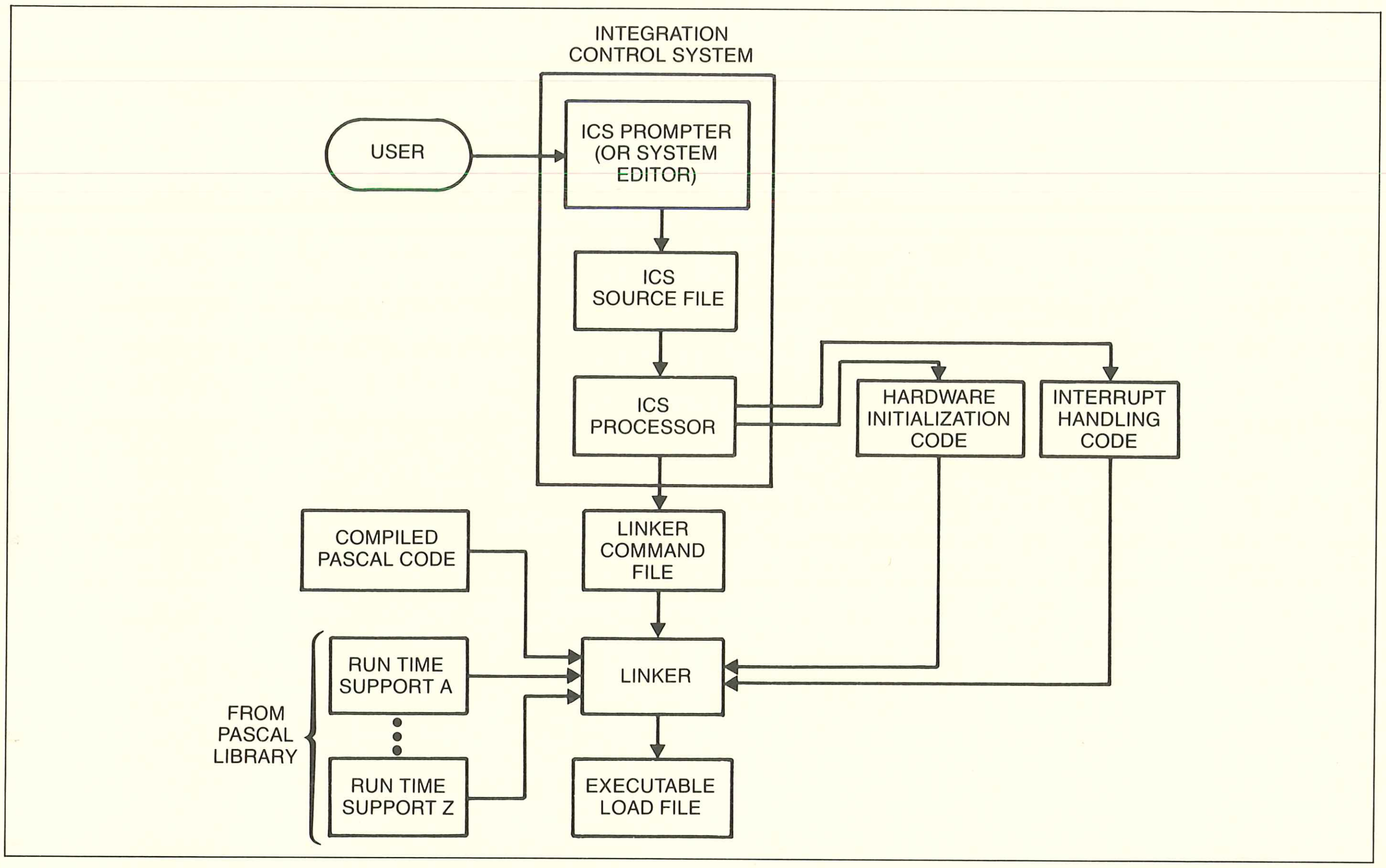


Figure 4. Mapping object code in prototype memory is a menu-prompted "piece-of-cake" for the ICS user. A simple menu prompts the scenario for creating the interface specification. (See figure 5.)

PASCAL_CONFIGURATION	ICS.MODULE
HARDWARE_CONFIGURATION	8086
INSTRUCTIONS_ROM	[0100H,03FFH]
CONSTANTS_ROM	[0400H,07FFH]
GLOBAL_VAR_RAM	[0800H,09FFH]
HEAP_STACK_RAM	[0A00H,0BFFH]
RESET_MEMORY	YES
SERVICE_CALLS	NONE
SOFTWARE_CONFIGURATION	trafficlight.po
MODULE	two.po
LIBRARY	NONE
FILE_HANDLING	DEFAULT
INTERRUPT_CONFIGURATION	ROM
INTERRUPT_TYPES_USED	30
SAVE_FLOATING_POINT	NO
EXCEPT_FOR	NONE
FAULT_NOTIFICATION	STOP
VECTOR	type30handler,30
RESTART_LABEL	PASCAL_BEGIN
END	

Figure 5. The LANDS user can create a software/hardware interface in mere minutes by responding to this menu.

Once the program is halted by a breakpoint, the debug software typically presents the user with a disassembled listing of instructions and operands as they exist in the emulator processor. This listing will also show the status of the processor's internal working registers as they responded to each instruction cycle.

Conventional debug software will also permit a "hex dump" that displays the contents of memory at the breakpoint. But the displayed information has no direct relationship to the original Pascal source code. What the user sees is an assembly-type version of compiled object code as executed by the emulator processor. If a bug is found, the user must mentally translate this assembly-type code back into Pascal in order to make the necessary repairs or analyze the specific nature of the problem. This translation is often arduous, time-consuming, and error-prone.

By elevating the entire debug tool set from the assembly level to the Pascal level, LANDS Pascal Debug eliminates 40 to 60 percent of debug time. The user can now debug completely within the context of the original source code, allowing powerful analytic insights when isolating a problem and faster corrective action once the problem is found.

For setting breakpoints at the source code level, Pascal Debug provides two methods. First, breakpoints can be specified according to the program statement numbers produced by the compiler, allowing fast access to any point in the program. Second, breakpoints can be assigned according to program elements, such as individual modules, procedures, statement labels, or variable names. This way, the user can assign breakpoints that are logically connected to a particular problem, such as an incorrect value assigned to a variable.

An important feature of Pascal Debug is the ability to set the breakpoint on variables and to qualify the breakpoint for read or write access to the variable, or both. Pascal Debug uses hardware features in the emulator to implement these breakpoints. Using LANDS Pascal Debug does not affect object program size or execution speed.

Pascal Debug also includes a single-step command that allows the Pascal program to be executed one statement at a time. Whenever execution is halted, a return command causes program execution to resume at any point specified by the user. And to start all over again, a reset command initializes the program to its original state at the time it was loaded into program memory.

At any time, the user can access a variable according to its symbolic name within the Pascal program, read its current value and, if desired, modify that value. This ability applies to all standard Pascal data types and user-defined types. The user may also query the data type of any variable by simply entering "type" and the variable's symbolic name in the source code.

Besides source-level breakpoints and variable examination/modification, Pascal Debug has two powerful features for tracing procedure activation. One is a "trace" command, which displays a message each time a procedure is entered, and each time it is exited. The values of any parameters passed to the procedure are also listed. Second, a "traceback" command lists, in reverse order, the procedures called and displays a traceback to the main program. In addition, the value of local variables for each procedure may be requested. Through these trace commands, the user clearly sees an informative picture of control flow within the program as it executes.

Applying the Pascal Language Development System

To see how the LANDS tools work in practice, consider a hypothetical application calling for an 8086-based microcomputer system. This system will control traffic signals extending across Maple Street. The signal system has three elements: (1) a traffic light with red, green and yellow states; (2) a crosswalk signal indicating "walk" or "wait," and (3) a pedestrian button to inform the system that a pedestrian wishes to cross.

Specifications for the signal system call for the traffic light to remain green and the crosswalk signal to display "wait" until a pedestrian pushes the button. This initiates a chain of events causing the traffic light to go yellow then red, and the crosswalk to go from "wait" to "walk." After a specified period, the crosswalk signal reverts to "wait," and a short time later, the traffic signal reverts to green. The system will stay in this state for a specified interval before it will act on a new signal from the pedestrian button.

Figure 6 shows the hardware, software and I/O needed to implement the system. Besides the main program, there are two modules dedicated to handling I/O. Module 2 handles the three traffic light colors and the two crosswalk messages. Module 3 handles the pedestrian button, which has an automatic hardware reset. Module 3 also handles a real-time clock interrupt, that the program requires to measure the time intervals between the signal states.

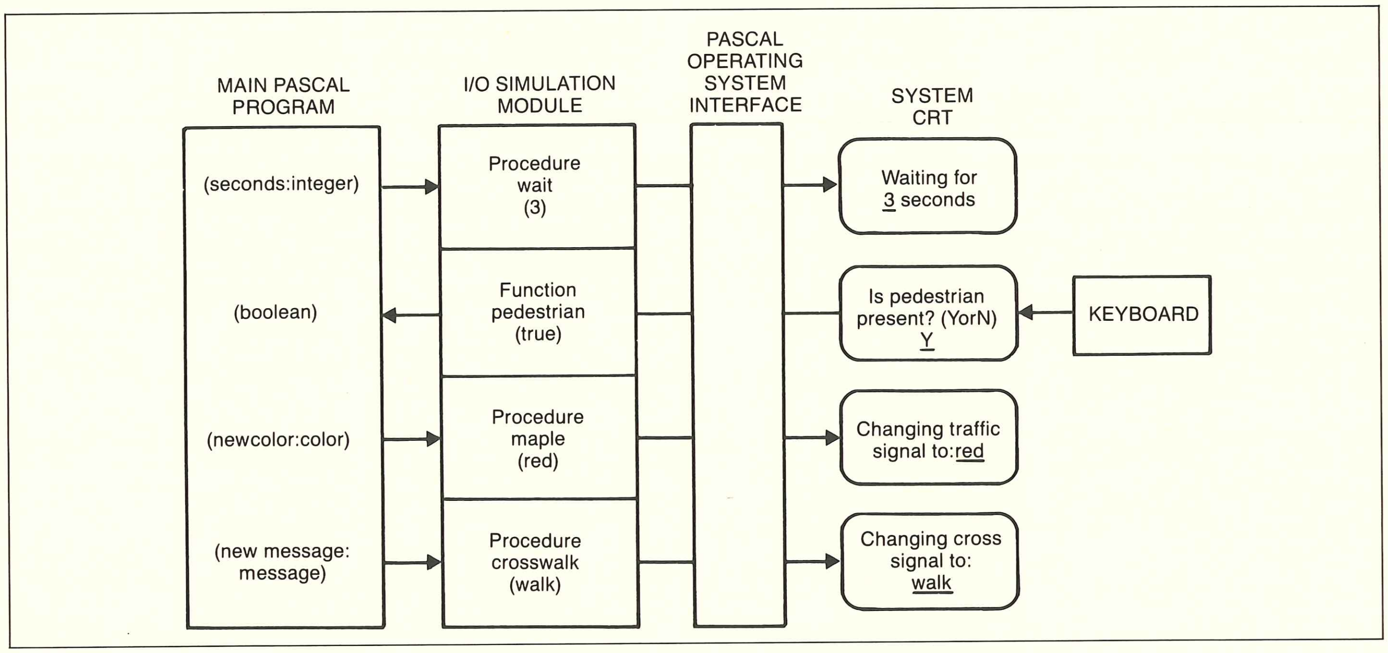


Figure 6. The Maple Street traffic control system.

```
 1                   program trafficlight;
 2
 3                   const
 4                        yellowlength = 3;
 5                        walklength = 30;
 6                        waitlength = 10;
 7                        greenlength = 45;
 8
 9                   type
10                        color = (green,yellow,red);
11                        message = (walk,wait);
12
13                   var
14             0:B             current : color;
15
16                   procedure wait(seconds : integer);extern;
17
18                   function pedestrian : boolean;extern;
19
20                   procedure maple(newcolor : color);extern;
21
22                   procedure crosswalk(newmessage : message);extern;
23
24                   begin
25     1       6:S        current : =green;
26     2      11:S        maple(current);
27     3      18:S        crosswalk(wait);
28     4      24:S        while true do (forever)
29     5      30:S             begin
30     6      30:S                  while not pedestrian do (nothing);
31     7      41:S                  current : =succ(current);
32     8      49:S                  maple(current);
33     9      56:S                  wait(yellowlength);
34    10      67:S                  current : =succ(current);
35    11      75:S                  maple(current);
36    12      82:S                  crosswalk(walk);
37    13      88:S                  wait(walklength);
38    14      99:S                  crosswalk(wait);
39    15     105:S                  wait(waitlength);
40    16     116:S                  current : =green;
41    17     121:S                  maple(current);
42    18     128:S                  wait(greenlength);
43    18                       end
44    18             end.
```

Figure 7. Valid traffic signal source code produced by the Language-Directed Editor and ready for the Pascal Compiler.

Figure 7 shows the source code for the main program as entered through the LANDS language-directed editor. A number of constants are declared for I/O with the signal devices, and a variable is declared to change the traffic light color. Three procedures and one function handle I/O operations with the signal devices and real-time clock. All of these are declared to be "external," meaning they reside in modules outside the main program.

The main program itself begins by defining the system's initial state and then describes the sequence for responding to a pedestrian pushing the button. Notice that the left-hand column shows assigned statement numbers. These numbers can be used later to specify breakpoint locations during debugging.

Imagine that the Pascal keyword "procedure" was misspelled in "procedure maple." Without the LDE, this simple mistake would have been included in the source file sent to the compiler. The compiler would have caught it and signaled an error requiring the source code files to be fixed and then entirely recompiled. However, LDE catches the misspelling during the editting session and the error is corrected in seconds.

Suppose the programmer had used LDE's abbreviation feature. Reducing the Pascal key word, "procedure," to "P^G" probably would have prevented the spelling error.

The traffic light application code is now ready to be run through the LANDS Pascal compiler. This will produce the object code used in the debug phase and executed on the emulator processor. To provide the simulated I/O for emulation and debug, the compiler creates a module containing the three procedures and one function declared external to the main program (figure 8). When the main program executes it calls each procedure when needed and passes data to it.

Once the emulation and debug are complete, the two I/O modules described in figure 8 can be coded to replace the simulation module in figure 8. Module two will contain procedures for I/O with the prototype hardware controlling the traffic and crosswalk signals. Module three contains the I/O function for the pedestrian button, and procedures for handling the interrupt-generated real-time clock.

Each of these modules contains LANDS enhancements aimed at machine-level manipulation of microcomputer hardware. Figure 9 shows the final I/O module used to control the traffic signal and crosswalk signal. The microcomputer hardware has memory-mapped I/O, with the signal devices controlled through I/O ports located at addresses 0200H and 0202H.

Conventional Pascal has no direct means of sending and receiving data through these ports, so special I/O assembly modules would have to be written and linked to the main program. However, with Pascal LANDS, I/O access is direct and simple. The two signal ports are declared as variables (mapleport and crossport) that are specifically located at 0200H and 0202H. Each port has a procedure that receives new output data from the main program and translates this data, with a case statement, into binary for the signal device.

For instance, if "procedure maple" in figure 9 receives a "newcolor" of red, it uses a case statement to assign the traffic signal port (mapleport) a data byte of 0100_2, which is then output to the traffic light, causing it to turn red. Likewise, if "procedure crosswalk" receives a "newmessage" of "walk," it uses a case statement to output 01_2 on the crossing signal's I/O port (crossport).

Interrupt handling and data manipulation at the bit level are two other important low-level attributes supplied with Pascal LANDS. Both are illustrated in I/O module 3, shown in figure 10. LANDS allows interrupt servicing to become an integral part of the high-level code – without requiring assembly code. With conventional Pascal, interrupt servicing is often troublesome because complex assembly coding is needed to define the relationships between interrupt service routines and the source code.

In figure 10, two procedures, "enable" and "disable," are declared external to module 3. These procedures are located in the runtime support library and will be called in at link time. They are used to turn interrupt 30 on and off in an 8086-based system. "Procedure wait" receives a "seconds" value from the main program and assigns it to a variable, "count." Next, the interrupt is turned on through "procedure enable" which activates "type30handler," which decrements the "count" variable each time an interrupt pulse is received. This process continues until the count reaches zero, and control is returned to the main program. In this manner the interrupt pulses, which occur at one second intervals, have been used to decrement a counter value assigned by the main program through "procedure wait." In effect, the programmer has supplied the Pascal code with a real-time clock/timer without ever resorting to assembly-level programming.

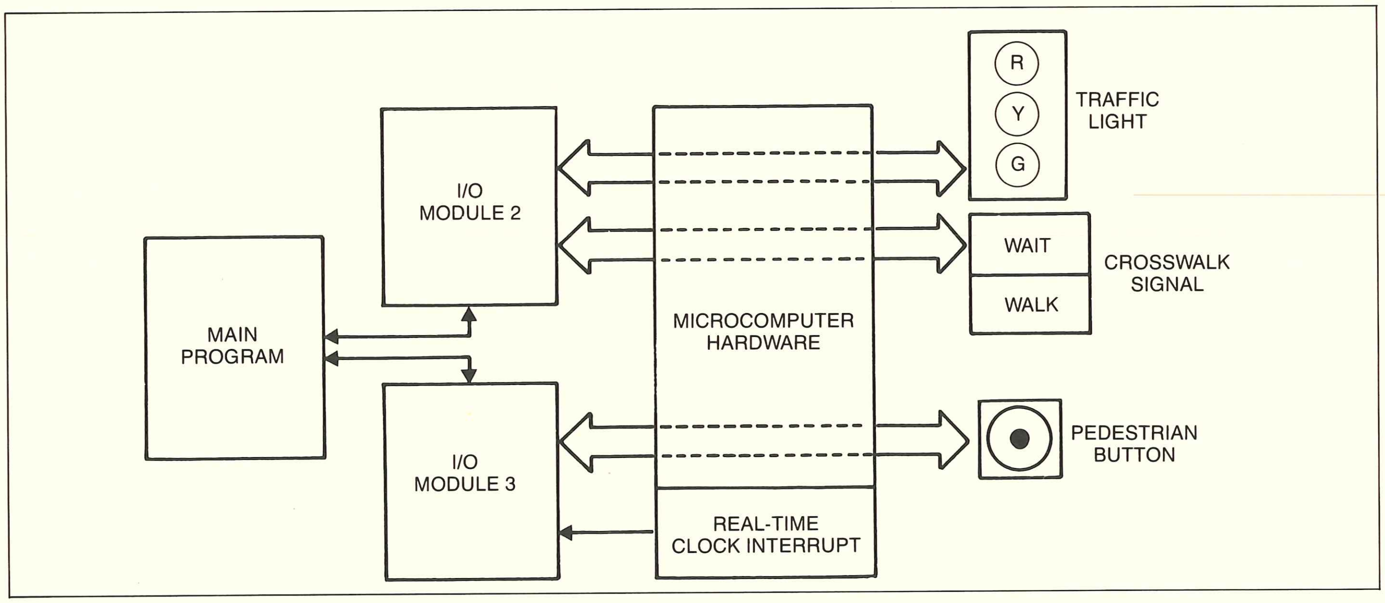


Figure 8. The I/O simulation used during emulation of the processor and debug of the object code for the traffic signal system.

```
                      module two;

                      type
                                  color = (red,yellow,green);
                                  message = (walk,wait);
                                  byte = 0..255;

                      var
                                  mapleport [port16#0200] : byte;
                                  crossport [port16#0202] : byte;
                0:B               lastcolor : color;

                      procedure maple (newcolor : color);public;
                          begin
       1        7:S           case newcolor of
       2       14:S               red:    mapleport : = 2#0100;
       3       26:S               yellow: mapleport: = 2#0010;
       4       38:S               green: mapleport : = 2#0001;
       4                      end;
       5       49:S               lastcolor : = newcolor;
       5                  end;

                      procedure crosswalk(newmessage : message);public;
                          begin
       6       66:S           case newmessage of
       6                      walk:
       7       73:S               crossport : = 2#01;
       7                      wait:
       8       85:S               crossport : = 2#10;
       8                      end;
       8                  end;

                      end.
          0 Errors
          0 Warnings
```

Figure 9. The finished I/O module used to control the traffic signal and crosswalk signal. The module contains microprocessor extensions.

"Function pedestrian" in figure 10 illustrates data manipulation at the bit level through a boolean operation. "Pedport" is the I/O port that receives input from the pedestrian button, which supplies a 0000_2 input as long as the button has not been pushed. This input is ANDed with the value 0110_2. If the result is 0110_2 then the boolean expression is not equal to zero, and "function pedestrian" returns a "true" value to the main program, which in turn activates the entire traffic signal cycle.

To interface the "traffic light" program with its 8086-based hardware, the user interacts with an integration control system menu similar in format to that shown in figure 5. This interaction provides interface parameters such as memory allocations, interrupt specifications, and hardware configuration.

Once all information requested by the menu is completed, an executable load file can be built as shown in figure 11. From a set of special library routines, ICS constructs object code for both initialization/reset and handling a type-30 interrupt from the real-time clock. It also determines which specific run time support libraries must be searched by the linker for the specified hardware and software configuration (in this case, an 8086 processor without 8087 co-processing for floating point). The ICS creates a linker command file that causes the compiled Pascal code, runtime support modules, interrupt handler code and initialization/reset code to be linked into an executable load file which conforms to the prototype memory map. With minimal effort, the user now has a program ready to be debugged in the prototype hardware environment.

```
1                    module               three;
2                    type                 byte = 0..255;
3            0:B     var                  count: integer;
4                                         pedport [port 16#100] : byte;
5
6                    procedure enable;extern;
7
8                    procedure disable;extern;
9
10                   procedure wait(seconds:integer);public;
11                        begin
12      1    7:S                          count : = seconds;
13      2   20:S                          enable;
14      3   23:S                          while count > 0 do {nothing};
15      4   51:S                          disable;
16      4   14:S          end;
17
18      4            procedure type30handler [interrupt];
19                        begin
20      5   65:S                          count : = count - 1;
21      5                 end;
22
23                   function pedestrian: boolean; public;
24                        begin
25      6   94:S                     pedestrian : = (pedport and 2#00110) < > 0;
26      6                 end;
27                   end.
          0 Errors
          0 Warnings
```

Figure 10. Microprocessor extensions – interrupt attributes.

The "traffic light" program is now run under control of the LANDS Pascal Debug. To initiate the debugging process, the executable load file is transferred to the development system's program memory via Pascal Debug where it can be directly accessed by the emulator processor, which controls the prototype hardware through a probe connecting it to the vacant processor socket on the prototype board. This way, the emulator can directly control I/O operations in exactly the same manner the target processor will control them in the final product.

In the course of running the program on the emulator, a major bug is encountered. At a time when the program changes the pedestrian crosswalk signal to "walk" it also changes the traffic light to green – a potentially lethal situation in the real world. To understand the nature of this problem, the main program and the I/O module for the crosswalk signal and traffic light are presented in figure 7 and 9. Statement 11 of the main program is the point where the traffic light should be changed to red. This operation is carried out by "procedure maple" in module 2, which receives the new traffic light color through the parameter "newcolor." This procedure then uses a case statement to select a statement which outputs a function code for the new color as a binary value through the I/O port "mapleport."

Figure 12 lists the Pascal Debug sequence used to track down the problem. A set of line numbers (1 to 27) has been added for reference to its contents. The "–" character is an input prompt from Pascal Debug to halt the code's execution at line 11 of the main program where the light should be changed to red. Line (4) shows that hardware breakpoint one (HW1) did indeed halt the program at line 11. Line (6) requests the current value of the variable "current" in the main program, and line (7) shows that this value is red. From this information, we now know that the main program passed the correct argument to "procedure maple," and that the problem most likely resides somewhere inside module 2, which contains "procedure maple." For this reason, the next breakpoint on statement number 4 of "procedure maple" is set in line (8) to see what happens after the argument is received. Line (11) shows the program stopped at statement number 4 in module two, where the execution of "procedure maple" is complete. Line (13) requests the current value of "mapleport," and line (14) shows it to be the integer one, whose binary equivalent at the port's output is 0001, which corresponds to green at the traffic light.

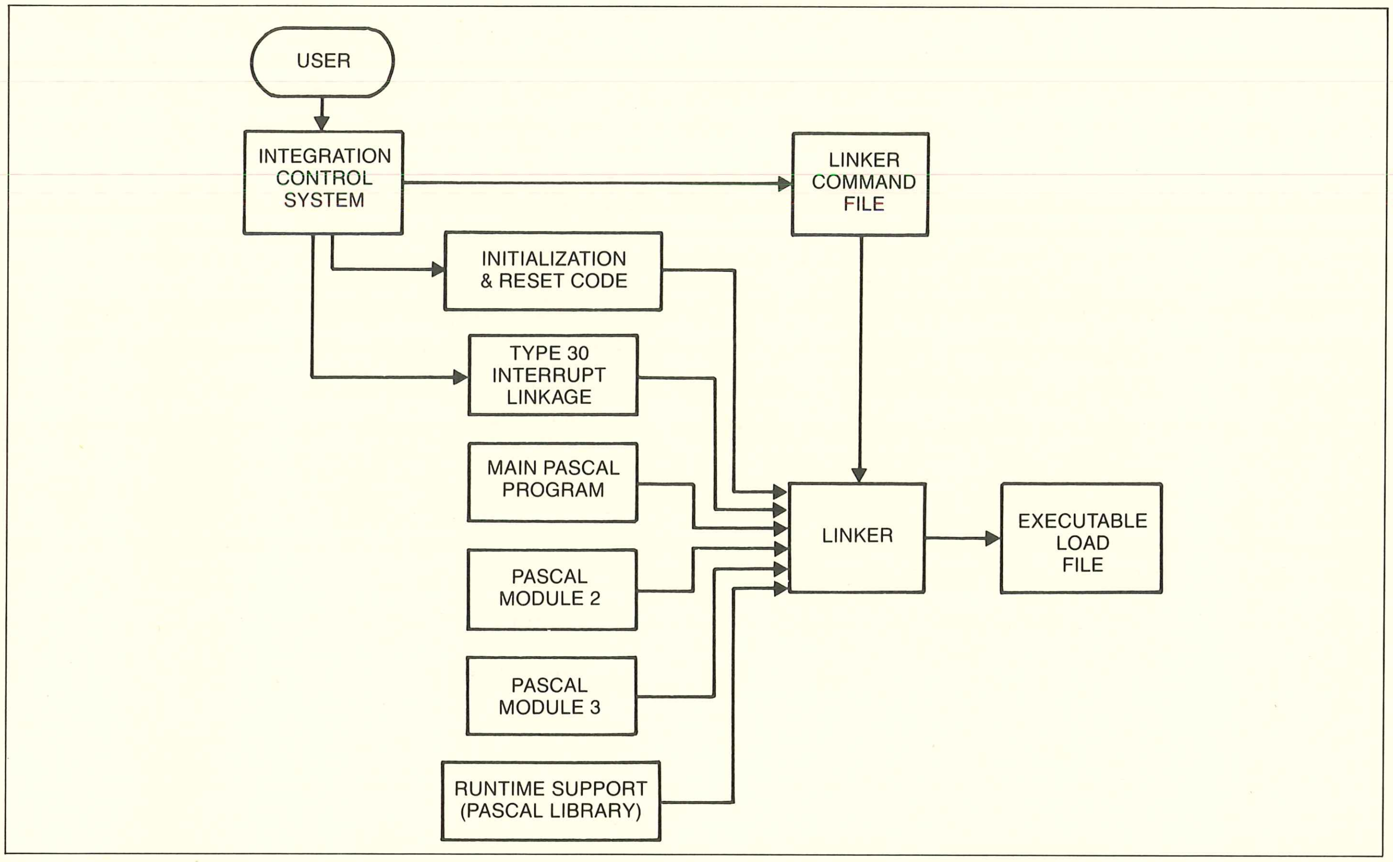

USER
INTEGRATION CONTROL SYSTEM
LINKER COMMAND FILE
INITIALIZATION & RESET CODE
TYPE 30 INTERRUPT LINKAGE
MAIN PASCAL PROGRAM
PASCAL MODULE 2
PASCAL MODULE 3
RUNTIME SUPPORT (PASCAL LIBRARY)
LINKER
EXECUTABLE LOAD FILE

Figure 11. After the user provides the menu-prompted information, an executable load file can be built.

```
 1 # break trafficlight#11
 2 # go
 3 # Breakpoint Encountered:
 4 HW1:    Used     R/W     TRAFFICLIGHT#11
 5 Program stops at statement 11 of TRAFFICLIGHT
 6 # current
 7 CURRENT = RED
 8 # break two.maple #1
 9  # go
10 Breakpoint encountered:
11 HW2:    Used     R/W     TWO.MAPLE#1
12 Program stops at statement 4 of TWO
13 # mapleport
14 MAPLEPORT = 1
15 # newcolor
16 NEWCOLOR = GREEN
17 # tb
18 $0:      MAPLE          1AE at statement number 4
19                Parameter(s):
20                     NEWCOLOR = GREEN
21 $1:   TRAFFICLIGHT       E2 at statement number 12
22 # type newcolor
23 COLOR
24 # type color
25 (RED,YELLOW,GREEN)
26 # type trafficlight.color
27 (GREEN,YELLOW,RED)
```

Figure 12. The Pascal debug sequence that identified the walk/green light problem, which is the way "procedure maple" interprets the argument pass.

It is now known that the main program is passing a "red" argument to "procedure maple," which in turn is outputting a "green" value to the traffic light control hardware. Line (15) requests the value of the local variable "newcolor" used by the case statement, and line (16) shows it to be "green." It now appears that the problem centers on the way "procedure maple" is interpreting the argument passed from the main program. To verify this, line (17) requests a traceback of activation for "procedure maple." Line (18) shows this procedure was currently activated when the last breakpoint occurred, and line (20) shows that its parameter was "green." Line (21) shows that "procedure maple" was called from the main program, "traffic light."

Based on the traceback, it becomes apparent that the color data is being inadvertently switched from red to green during the transfer from the main program to "procedure maple" in module two. This indicates that there may be a data type problem, so line (22) requests a data type check on the variable "newcolor," and line (23) shows it to be "color." Since "color" is an enumerated type, the next step is to check its declaration within the current procedure, "mapleport." line (25) shows it declared here as ("red, yellow, green"). Next line (26) requests the declaration for the data type "color" within the main program and line (27) shows it to be ("green, yellow, red").

Comparing the data type declarations for "color" within the main program and calling the procedure in module two pinpoints the problem. The declarations are inconsistent as to the order in which the values are placed. In the main program the order is green, yellow, red. In module two, the order is switched to red, yellow, green. This means that "red" data sent from the main program will be interpreted as "green" when it enters module two and "procedure maple." As a consequence, the variable "newcolor" receives a "green" value through the mixed up data declaration and is used by the case statement to switch on the green light even though the main program originally requested "red." With larger design teams and modular programming, problems like this are commonplace.

To track down this same problem using assembly-level debug tools would have been a formidable task. At each step, the debug data would have to be translated by the user back into its source code context. Besides consuming time, it means the user's concentration on the real problem is continually interrupted, making accurate analysis doubly difficult. Pascal Debug, on the other hand, keeps the user constantly in the Pascal program environment and creates a much more productive problem-solving climate.

Summing Up LANDS Pascal

The Pascal Language Development System is the first high-level microcomputer software development package that extends fully to every phase of program development. The Language-Directed Editor provides complete Pascal syntax checking, and thus eliminates a major source of errors that lead to recompiling. The LANDS Pascal compiler extends high-level control right down to the hardware level, removing the need for assembly coding of low-level operations. The Integration Control System reduces software/hardware interfacing to a brief session with a menu or file. Pascal Debug elevates all debug commands and resultant trace information to the source code level, allowing the programmer to debug entirely in Pascal. Taken either as parts or a whole, Pascal LANDS offers an efficient, cost-effective pathway to increasing software design productivity by 40 to 50 percent.

For More Information

For more information, call Charles Montgomery, 629-1100. □

This article was adapted from material that was later published as an article by Charles Montgomery in *Electronic Design,* July 21, 1983. This material was also presented at ELECTRO 1983 by Doug Johnson.

IMI STANDARDS SERVICES AVAILABLE

In 1981, Technical Standards added a microfiche reader/printer and a supply of Industry and Military Standards to its library of standards. Recently we improved our system by changing to the services of Information Marketing International (IMI). IMI provides more pertinent and timely information for standards users. Copies from the IMI system are dark gray on white, legible and clear. Copies are not slick – so pencils, markers and pens do not smear. All standards are updated every 60 days.

Requests for copies are filled within two days. If you need copies of a current military or industry standard, or more information, please contact Lena Hankins, 627-1800. □

Technology Report
MAILING LIST COUPON

☐ ADD
☐ REMOVE

Not available to field offices or outside the U.S.

MAIL COUPON TO 53-077

Name:________________________ D.S.: ________
Payroll Code: ________________________
(Required for the mailing list)

For change of delivery station, use a directory change form.