DESIGN OF A GRAPHICS DISPLAY SYSTEM

by

John J. Hudak
Electrical and Computer Engineering Dept.
Carnegie-Mellon University

1984

THE DESIGN OF A

VIDEO DISPLAY GRAPHICS PROCESSOR

Submitted In Partial Fulfillment Of The Requirements For

The Degree Of Master Of Science

Electrical Engineering

by

John J. Hudak

Electrical and Computer Engineering Department

Carnegie-Mellon University

Pittsburgh, Pa.

April 30, 1984

A Video Display Graphic Processor

Abstract

To aid in the study of neural reflexes of the ocular nerves of various subjects, the Biomedical Department is in need of a general graphic display processor. The primary purpose of the graphic display in this case is to display a variety of objects and have them move in certain pre-programmed paths. These moving objects, when placed before a subject such as a frog, will serve to stimulate the ocular nerves whose impulses will then be recorded and analyzed. Further on in the cycle of experiments, the processor will be connected to a host 68000 based multi-processor system. This system will enable the 68000 to download images and control the movement of patterns displayed to the subject under test and in effect, close the feedback loop.

The motivation for this project is two-fold in nature. The first concern is that a versatile, low cost interactive graphic system can be implemented with current state of the art MSI and LSI designs and secondly, by using LSI devices with the appropriate support circuity, design and build the necessary hardware and specify the software outline to produce such a graphic system. Until recently, many designs to produce low cost, interactive graphic displays centered about devices such as CRT controllers and single chip/system microcomputers. While it is possible to produce relatively simple graphic images by these methods, it becomes almost impossible to generate more complicated multiple (>3) images and support the animation program while at the same time acting

interactively with the user.

The use of a "two part" system consisting of a microprocessor and video graphic processor would alleviate the problems encountered above and would fulfill the premise of a low cost interactive graphic system. The microprocessors' function would include interpreting user commands, update display files and implementing the animation subroutines. The video processor would control the storage, retrieval and refresh of the display data in the dynamically refreshed memory, provide the necessary video timing (ie. H sync V sync, etc.) and originate the composite video for the CRT display. This method provides an efficient solution to the project goals.

This project is centered around the hardware design and implementation, along with a limited set of graphic primitives to demonstrate the workability of the design. To date, graphic system implementations have taken the following approches. They are: 1) frame buffers (single and multiple), 2) real time scan conversion, 3) run length encoding and 4) cell organized displays. Generally these techniques employ fast, expensive hardware and optimized software which increases the cost of many systems. With the advent of a new generation of commercially avaliable IC's (termed a Video Display Processors (VDP)), the realization of lower cost and versatility become closer at hand.

Some of the more prominent design goals to be addressed by the project are: 1) the capability to generate and animate various shapes such as open and solid geometric shapes (ie. circles,

triangles, etc.) 2) multiple images, 3) varying images sizes, 4) reverse video capabilities and 5) flicker free animation. The display area will be 192 by 256 pixels (aspect ratio of 3:4) and will also have the capability of a limited gray and color scale. Therefore, the project will consist of designing and assembling the essential hardware and specifying the necesary software that needs to be be incorporated to yield a stand alone interactive graphic display. Basically the hardware building blocks will consist of a R6502 microprocessor system in conjunction with the VDP and associated support hardware such as video RAM, clock stretching and deskew circuitry and NTSC video signal conditioning circuitry. The system must also be able to service either a terminal and/or a high speed serial data link to the host 68000 system. A list of graphic primitives will be specified and some will be implemented to demonstrate the workability of the hardware. These primitives would fall into the skeletal form of the graphic operating system.

ACKNOWLEDGEMENTS

The author is indebted to many people for their help and generosity during the development of this thesis. The author is very grateful to teachers, relatives, and friends who have assisted and encouraged him during all these years.

The author thanks Dr. Arthur Sanderson who provided the guidance and persevered throughout the duration of the project. Also the author acknowledges the thesis committee: Dr. Douglas Jensen and Dr. Ronald Krutz.

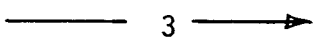
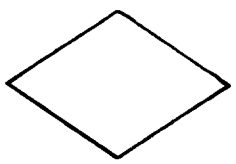
The author is grateful to Dr. Ronald Krutz for allowing him to use the computer facilities at the Mellon Institute, Computer Engineering Center and for his encouragement throughout the project. A special thanks goes to Glen P. Williams of the Computer Engineering Center a close friend who always had time to listen and critique ideas and offered support during the project development.

The author can never fully acknowledge the support of his parents, Mr. John Hudak and Mrs. Mary Hudak for their understanding and support.

Finally, the author is grateful for the skillful proof-reading and critiquing of this thesis by Mr. Tron McConnell, Ms. Agapi Svolou, and to his sister, Mrs. Janet Skoner. Special thanks go to Mr. Gordon Wenneker for helping the author prepare the system schematic and to Ms. Trisha McAleavey for helping to assemble the manuscript.

CONTENTS

Conventions Used Throughout This Thesis

$	Represents hexidecimal notation
subroutine GETCHR	Subroutine names are capitalized
variable CHAR	Variable names are capitalized
	Indicates a subroutine GETCHR, Numbers indicate possible error codes generated by the subroutine.
	Represents program flow, number indicates path number
	Indicates a two or three way decision
|	Logical OR or union
*	Closure operator indicating zero or more instances
()	Indicates a subset
/	Represents division
2 exp n-1	Equlivalent to 2^{n-1}

1. Introduction

The term "graphics system" is one of the many terms in electrical engineering and computer science that has multiple definitions. Graphics systems can range from a video game as simple as the early PONG game through large systems such as General Electric's CALMA systems. Recently, video games represent the first major use of computer graphics in the home. Basically these games create and manipulate pictures with the aid of a computer (microprocessor). The user provides feedback to the system through the use of joysticks, push-buttons or potentiometers. The video display quality provided by these systems range from low to medium resolution. In general these systems are limited in their video quality by two factors. The first is the display technology. Generally, the displays used in these games are color or black and white "television quality" raster scan devices that have a video bandwidth of 13-15 MHZ. Secondly, the graphics engine that creates and clocks out the pixel information is relatively slow compared to high performance systems. At the present time, the cost of high performance CPU(s) and video memory rises exponentially with the speed factor. These are the two main limiting hardware conditions that exist in designing a graphics system.

At the other end of the spectrum, the high end graphics systems, are things such as real time flight simulators/trainers and CAD/CAM systems. These systems consist of extremely high resolution displays with video bandwidths greater than 100 MHZ that can accommodate at least a 1024 by 1024 pixel display format. In

addition, the graphics engine is for the most part subdivided into a number of tightly coupled engines. They are the object generation Central Processing Unit (CPU), display list manager cpu and object animation/display cpu. In many cases these machines are 16/32 bit cpu's, or specially designed bit-slice/LSI devices. Depending upon the desired performance of the product, these building blocks are optimized and combined accordingly.

This thesis deals with the lower end of the graphics system spectrum. The goals of this work can be summarized by the following points:

a. To design and implement a low cost graphics system with the ability to easily create and animate objects.

b. To specify, and to a limited extent implement, a graphics language that can generate objects and that has the necessary "software hooks" to implement the part of the language aimed at animation.

The keywords to be highlighted are "low cost", "specify", and "animation". Low cost is defined to mean that the hardware cost should be in the range of $100 to $150 for a stand-alone graphics processor. "Specify" means to develop the language constructs that are needed to do the job. "Animation" is used in the context of using the frequency, direction (x,y,z axis), and delay as motion attributes of an object.

This thesis discusses the methodology used to design both hardware and software. The initial thrust of this project was to develop only the graphics hardware. Through subsequent discussions

with advisors and other graduate students, the language aspect became more significant. This thesis represents the results of these discussions along with some enhancements deemed appropriate by the author.

1.1. Graphics Systems

Graphics systems can be configured in many ways, depending upon their intended use. At one extreme is the "stand alone" graphics system shown in Figure 1-1. Nearly all of the system's computing capabilities and most of its memory resources are dedicated to the display task. Systems such as image processors, computer aided design (CAD), and graphics work stations fall into this category.

At the other end of the extreme is the configuration depicted in Figure 1-2. Nearly all of the graphics-generation functions must be performed by the host computer's hardware and software. The graphics system is reduced to a graphics terminal and the display generation elements of the host interface.

Raster graphics systems fall midway between these two configurations. With few exceptions, smaller stand-alone graphics systems rarely have the computing power to perform all of the extensive modeling and viewing functions required to convert source data to graphics form. The generation of a raster graphics display can, at the same time, impose a severe burden on a host computer's processor and memory resources. To overcome this problem, the concept of a graphics engine or controller has been developed.

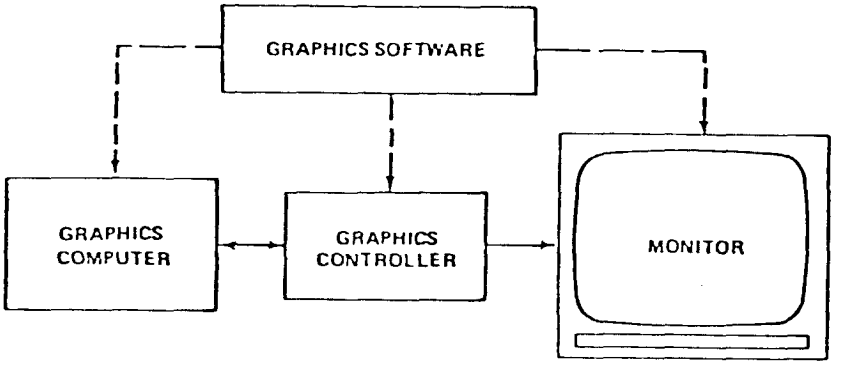


Figure 1-1 Distribution of graphics-system software functions.

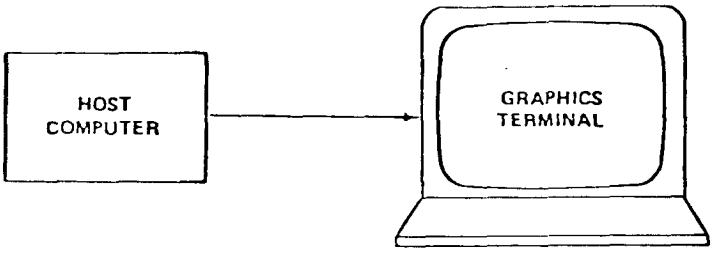


Figure 1-2 Intelligent terminal with "graphics" capabilities.

The graphics engine is responsible for a large portion of the repetitive graphics functions such as video memory accesses, pixel clocking, and in more advanced implementations, updates of display lists and user interface.

The term "graphics engine" can be applied to devices that range from integrated circuit (IC) components to dedicated mini and mainframe computers that are utilized for graphics tasks. The early video games are examples of IC implementations of this type of system. In the move towards offloading the microprocessor of various graphics tasks, the video display processor (VDP) or graphics display controller (GDC) have surfaced. These devices may contain some or all of the following features: eight bit parallel/ bidirectional data bus, interrupt structure, direct video RAM access/update, microprocessor memory DMA, generation of National Television Systems Committee (NTSC) composite color video information/horizontal/vertical signals, high speed (~25 MHZ) pixel clock rates, and video RAM refresh if dynamic RAM's are used.

1.2. Hardware Design Overview

1.2.1. The Microprocessor System

The approach that was used to accomplish this project was to design a 6502 microprocessor based system and couple it with a Texas Instruments TMS 9918 VDP. The microprocessor system was designed to include the following capabilities:

a. Sufficient RAM to create/edit/store display lists.

b. Sufficient serial ports to communicate with the user's terminal and also to down/upload information to a host computer.

c. Parallel ports to interface to user devices such as a joystick, trackball, etc.

d. Sufficient timers to permit the animation of a number of objects or to start/stop tasks in a multi-tasking environment.

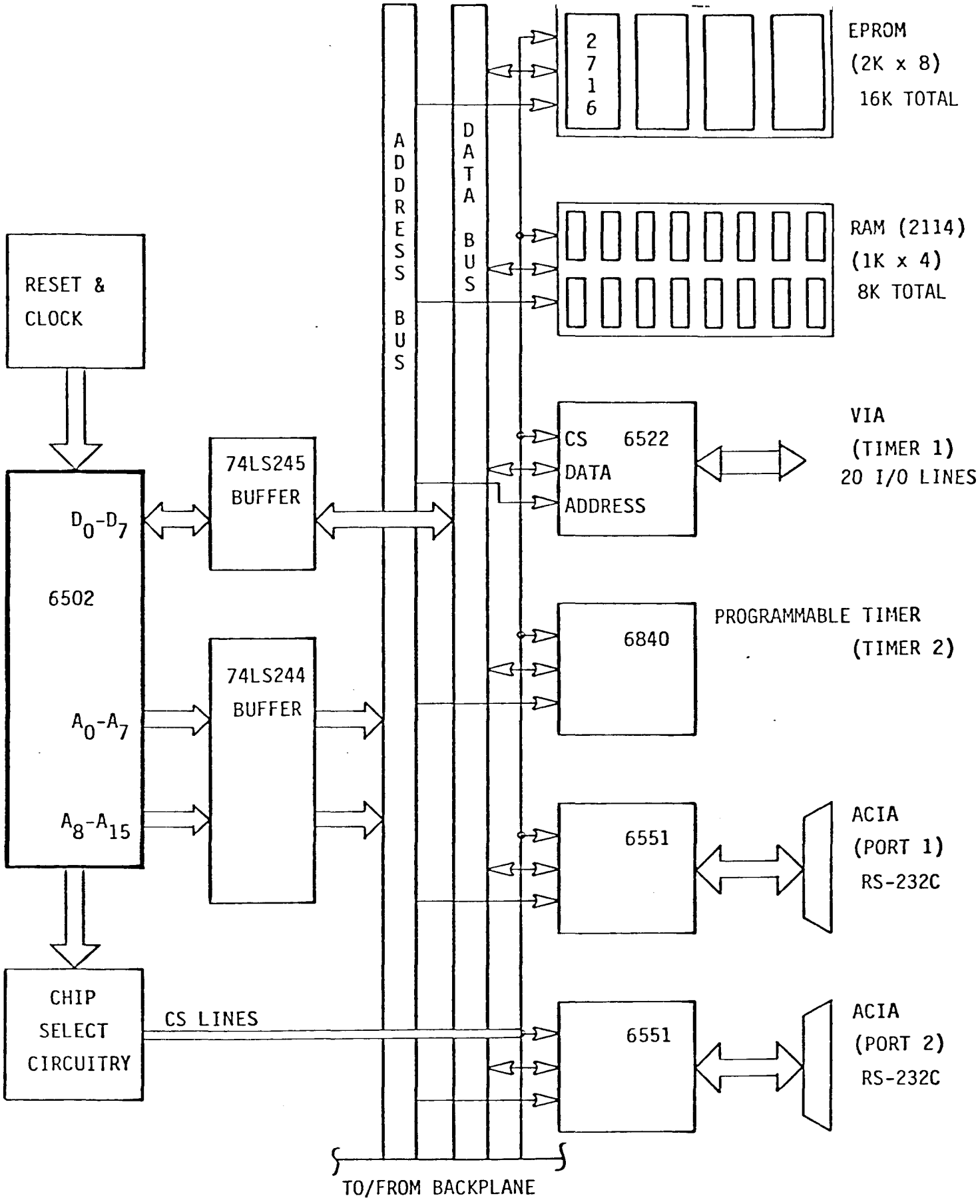
The designed system block diagram can be seen in Figure 1-3. The 6502 based system resides on one S-100 form factor wire-wrap prototype board. It includes:

- Two R6551's Asynchronous Communication Interface Adapters (ACIA) for the RS-232 link to the users terminal and host computer.
- A MC6840 timer that contains three hardware timers able to operate in various modes.
- A R6522 Versatile Interface Adapter (VIA) that contains two hardware timers and two 8-bit parallel ports.
- 8K bytes of static RAM.
- 16/32K bytes of ROM that contain the micro-processor/VDP monitors, download program and graphics interpreter.

1.2.2. The Video Display Processor System

The microprocessor system communicates with the video display section. The graphics board specifications were as follows:

- The display area was to be 192 X 256 pixels (3:4 aspect ratio).



Microprocessor Board Block Diagram

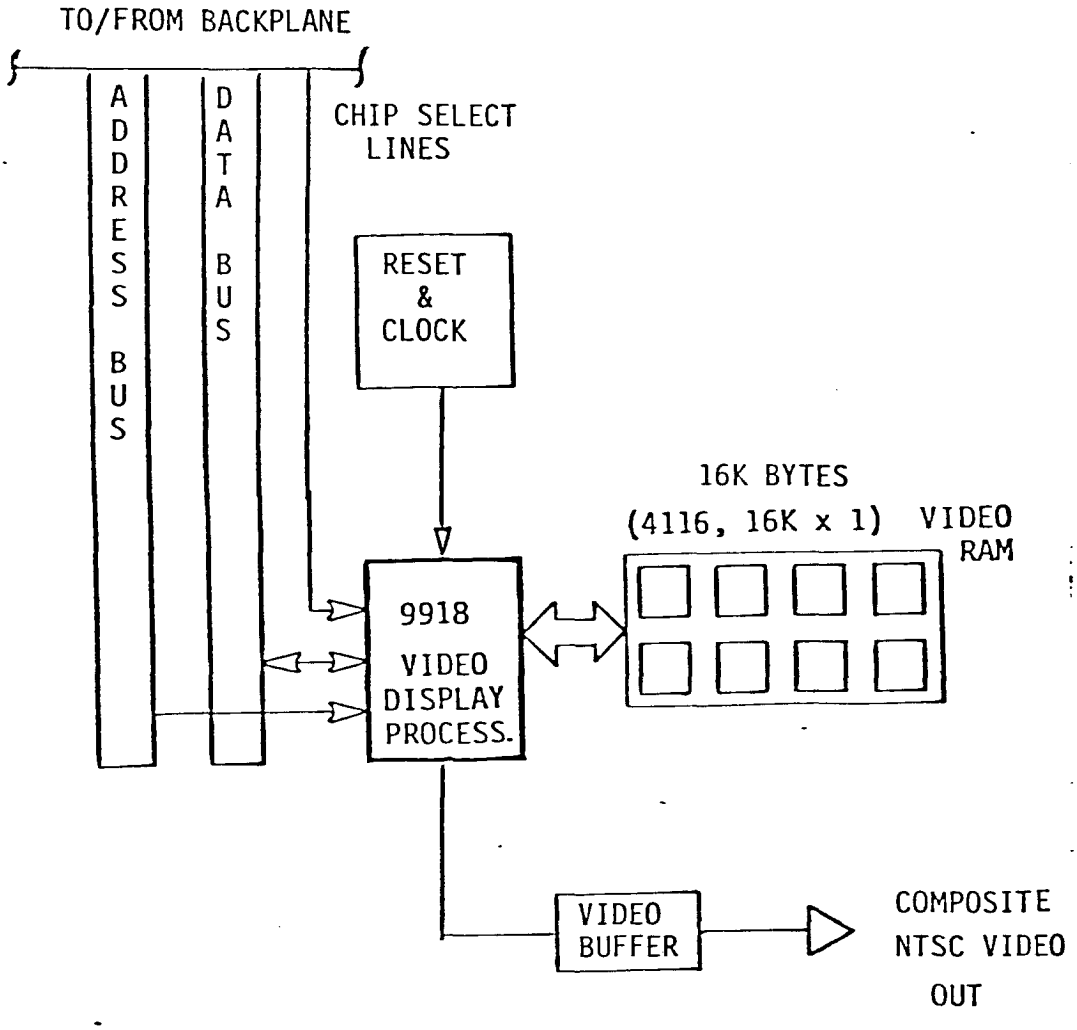
Figure 1-3

- Black and white was essential and color was not necessary.
- The movement of objects should not overburden the microprocessor.
- The foreground/background colors could be exchanged instantaneously (reverse video).
- Object movement should be flicker free.
- System should be expandable (re: multiple display planes or VDPs).

Many designs for this portion of the system were considered. The areas investigated include single and multiple frame buffers, real time scan conversion hardware, optimized run-length encoding hardware and cell organized displays. As will be pointed out later in the paper, these approaches had both good and bad qualities. The reason they were discarded was their cost was too high. The VDP had just appeared on the market and although it was a first generation device, it had a number of important attributes that allowed such a system to be designed at a low hardware cost.

The graphics board block diagram can be seen in Figure 1-4. It consists of the following devices:

- The TMS 9918 VDP.
- 16K bytes of video RAM.
- Composite video amplifier and CRT interface.
- Interface to the host CPU.
- Address decoding.



Video Display Processor (VDP) Board Block Diagram

Figure 1-4

The graphics subsystem was connected to the S100 form factor backplane, and the VDP appeared as a memory-mapped peripheral in the main (6502) processor's address space, allowing fast and efficient information transfer.

1.3. Software Design Overview

In order to make the graphics system functional, a set of graphics commands and support software had to be designed and implemented. The scope of this task was large enough to easily exceed the project definition. In order to make the hardware useful, a limited set of support software was specified and written. This included:

- A microprocessor monitor (adapted from another system).
- A graphics processor monitor.
- A download program.
- Terminal handler (user interface).
- Lexical Analyzer.
- Command interpreter.
- Graphics language.

The scope of the graphics language was sufficiently large that only a subset was implemented in the current system. The terminal handler, lexical analyzer and command interpreter design had to take this into account. Support for the implementation of the entire graphics language command set, and also an editor, were included in the system.

In order to make the system as independent as possible for support of file systems, an editor would be required. Because of the size of the machine, a rather simple, but useful editor, capable of executing 14 commands was specified. The problem of fully developing this graphics editor was left for future implementation.

To provide the reader with an introduction and overview to the language and its grammar, the system commands can be reviewed below. While a complete set of system commands were specified, only a subset were implemented. The commands implemented are marked with an asterisk.

1.3.1. Graphics Primitives

* a) MOV M,X1,Y1 - Move cursor to new location specified by Mode (M), to X,Y

* b) VEC M,C,X1,Y1 - Draw a line from current position to a point specifed by Mode (M), position X,Y, with Color (C).

c) BARC M,C,X1,Y1 - Draw an arc with this point as a defined beginpoint with Mode (M), Color (C), at position X,Y.

d) MARC M,Xn,Yn - Draw an arc with these sets of control points specifed by Mode (M) thru points X,Y.

f) EARC M,Xn,Yn - Draw an arc with this point as the defined endpoint specifed by Mode (M), to point X,Y.

g) ARC M,R,C,X,Y,D - Draw an arc thru a set of defined points with radius (R), Color (C), center (X,Y), thru D degrees.

h) DTX C,"xxx" - Display the string characters with Color (C).

* i) CIR M,C,R,X,Y - Draw a circle with Mode (M), Color (C), Radius (R), with center at X,Y.

* j) RECT M,C,X1,Y1,X2,Y2 - Draw a rectangle (square) with Mode (M), Color (C), with Length X1,Y1, Width X2,Y2.

1.3.2. Windowing Commands

a) SWD X1,Y1, X2,Y2 - Set window using X1,Y1 and X2,Y2 as length and width from current cursor position.

b) SVP X1,Y1, X2,Y2 - Set viewport using X1,Y1 and X2,Y2 as length and width from current cursor position.

1.3.3. Display File Creation and Minipulation

a) DEFIL xxx - Define an image file with name xxx.

b) ENDFL - End image file name xxx.

c) LSTFL xxx - List the image file xxx.

d) DSFIL xxx - Execute file name xxx.

e) RMFIL xxx - Remove image file with name xxx.

f) CALLI xxx - Subroutine call to image file xxx.

g) RETIF - Return from subroutine call.

1.3.4. Animation File Attributes

a) PATHXY X,Y - Indicates the points X,Y thru which the object will move.

b) VEC X1,Y1,X2,Y2 - Describes a linear path for an object between endpoints X1,Y1 and X2,Y2.

c) DELAY nnn - Defines the time period in seconds to delay object motion.

d) REP nnn - Defines the number of times to repeat the objects movement. The movement between endpoints A and B would have the form: A-B, A-B,....A-B.

e) OSCIL nnn - Permits the motion of an object to be repeated in an oscillatiory manner, (i.e. A-B-A-B-A-B..), nnn times.

f) START xxx - Starts the movement of object xxx.

g) STOP xxx - Stops the movement of object xxx.

h) TIME nnn - Defines the time in seconds for the objects traversal of a path.

i) SYNC n,X,Y,i - Starts the movement of object n when object i reaches point X,Y.

1.3.5. Control and Miscellaneous Commands

* a) CLRSC - Clear the screen.

* b) INTIG - Initiate graphics mode.

* c) REVID - Reverse video (Reverses foreground and background color).

* d) PAINT ccc - Colors an object with color specifed.

* e) CURSO - Turns cursor on.

* f) CURSF - Turns cursor off.

* g) CURPOS - Displays the absolute cursor coordinates.

* h) SETBGC ccc - Sets the background color.

* i) SETFGC ccc - Sets the foreground color.

* j) SETBDC ccc - Sets the backdrop color.

* k) SETCC ccc - Sets the cursor color.

l) STAT - Displays the system status consisting of: cursor X,Y position, window and viewport dimensions, foreground, color, number of display files, and memory available.

* m) COLCUR ccc - Sets the cursor color to the color specified.

1.3.6. Editor Commands

a) ed xxx - Invokes the editor for use with filename xxx.

b) i - Inserts a new line.

c) d - Deletes a line.

d) w xxx - Writes out the newly edited file.

f) q - Quits the editor without saving any changes made.

g) a - Append to the end of the file.

h) b - Go to the beginning of a file.

i) . - Print current line.

j) .+n - Go to and print line n lines ahead of current line.

k) .-n - Go to and print line n lines before current

line.

l) cr - Advance thru file line by line.

m) - - Go backwards thru a file line by line.

n) s - Substitute a character or string for a character or string.

o) /xxx/ - Search for a character or string.

2. Hardware Design

The hardware design aspect of this project provided many challenges because of the constraints imposed by the specifications and the requirement for low hardware cost. The objects to be generated were relatively simple geometric shapes, such as circles, triangles, and rectangles. Difficulties arose from the fact that they had to be moved around on the screen without any flicker. There could be as many as three objects, each with a different direction and rate of movement. The objects could either be outlines or solids. In some cases, it would be required to have the screen exhibit reverse video characteristics without flickering. With these specifications outlined, the task of technology evaluation and hardware design began.

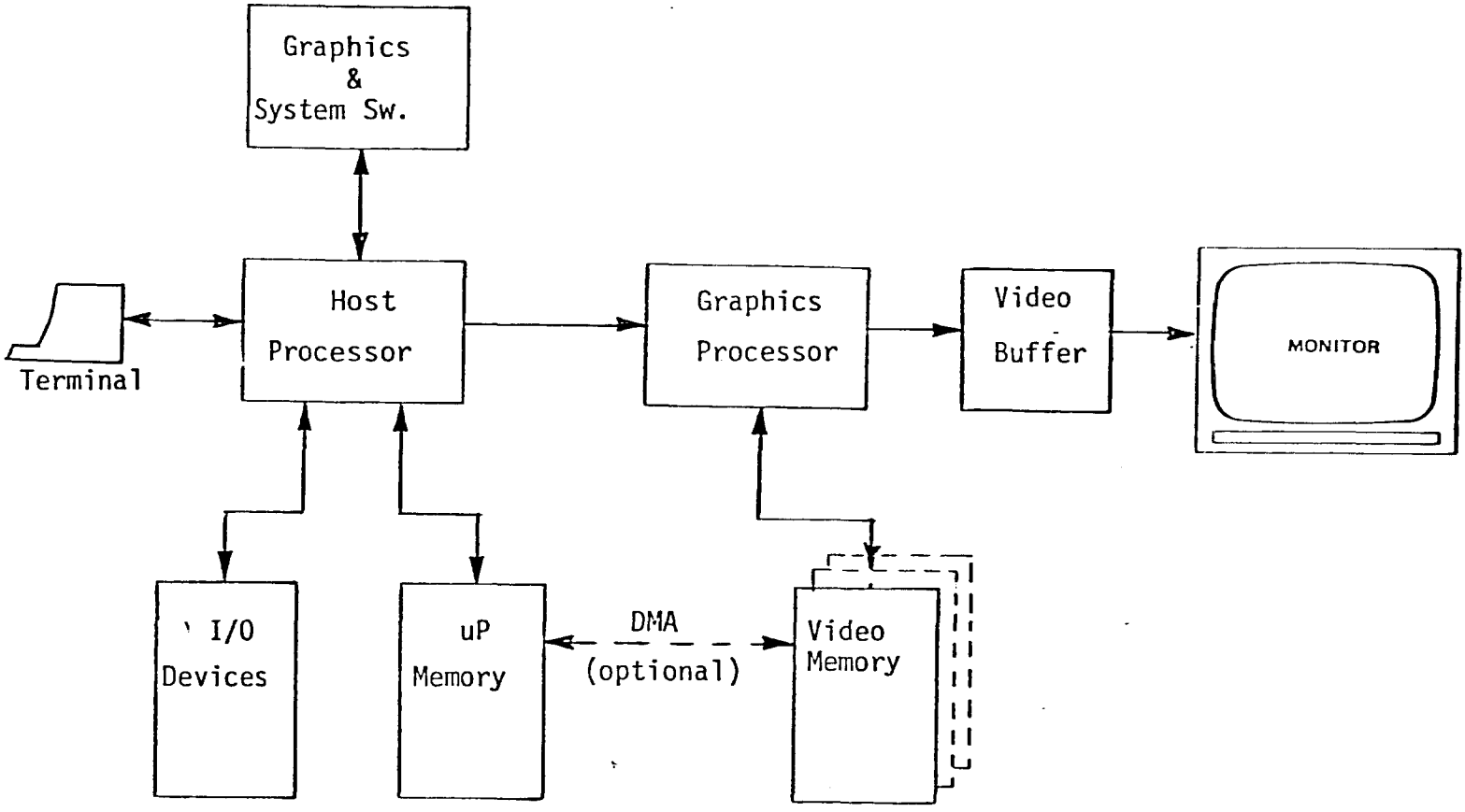
The first task that was performed was an assessment of display technologies. The specification governing this aspect of the project was animation of objects. Many different types of display devices and methods exist for implementing graphics systems. They range from liquid crystal displays (LCDs), dot matrix displays, and projection screens to electroluminescent displays. Many of these types of display devices were not suitable for a variety of reasons. The survey of display technologies revealed basically two types of usable displays: a basic cathode ray tube (CRT) and a CRT storage tube. The CRT could be used either as a raster or vector device. The storage tube closely resembles a direct writing CRT with one important difference. The storage tube construction consists of a writing beam gun and a flood gun. These guns, in

conjunction with a collector and storage grid allow objects to be displayed on the screen without very fast refresh cycles, as compared to a vector or raster CRT. In terms of performance, the storage tube is somewhat inferior to the refreshed CRT. Only a single level of line intensity can be displayed, and only green phosphor tubes are available. Since object movement was specified, the refreshed CRT was the necessary choice.

The choice of a refreshed CRT raised the question of whether to use the vector or raster display update techniques. Vector display electronics are very expensive and display uniformity can be lost in the process. In order to display a large number of objects on the screen, the electron gun has to move very rapidly across the screen. As the number of objects and/or the object complexity increases, the "flicker" of the displayed image becomes more noticeable. In order to overcome the flickering, a higher bandwidth CRT and faster video digital-to-analog (d/a) converters are required, which in turn would drive the cost up prohibitively. With a high speed raster display, the display memory access and subsequent parallel to serial conversion can be done at a relatively high rate of speed for lower cost as compared to the vector display electronics. It was therefore decided to use the raster display CRT.

2.1. General Graphics System Architecture

The basic hardware architecture of a general purpose graphics system can be seen in Figure 2-1.



Hardware architecture of a
general purpose graphics system

Figure 2-1

It consists of the host processor and associated memory, the video memory or frame buffer, the video controller, and the CRT driver electronics. A microprocessor was the obvious choice for the host processor, but what about the video processor? Three possible solutions were considered:

1) A microprocessor/ bitslice processor.

2) Discrete MSI/LSI TTL logic.

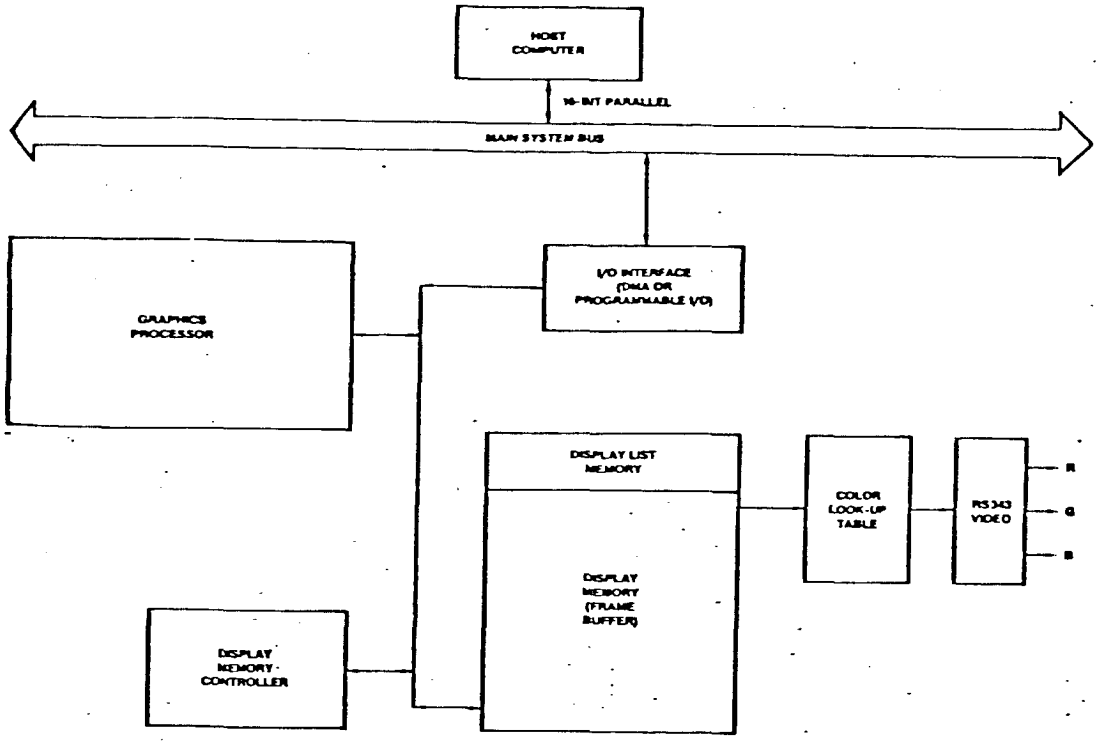
3) 3) NMOS video display processor (VDP).

Each design had both pros and cons associated with it which are reviewed below.

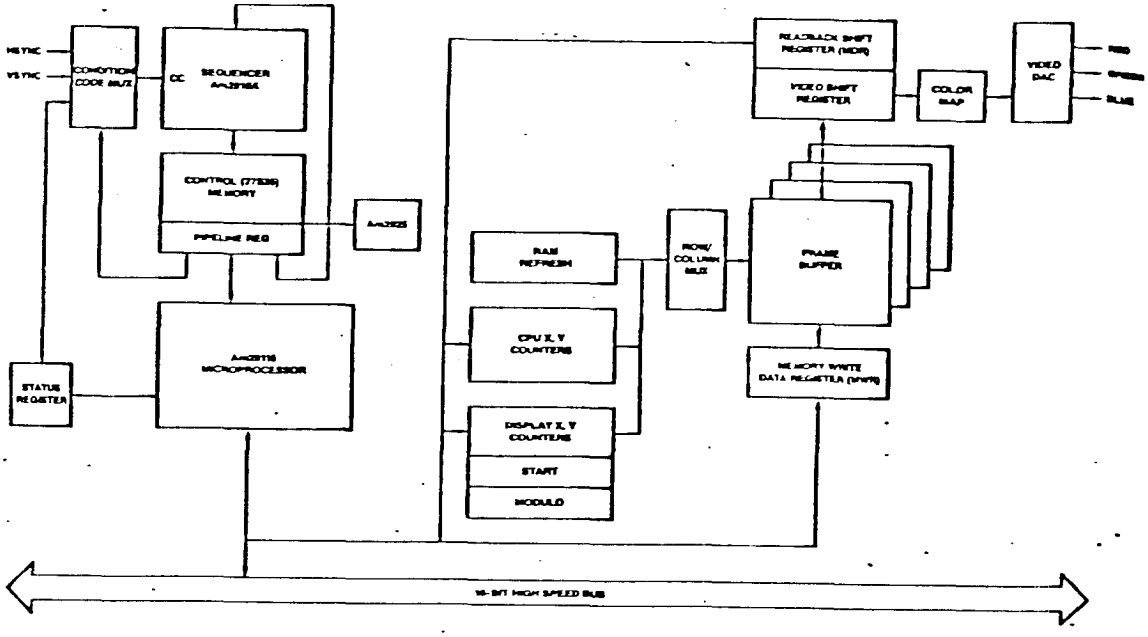
2.1.1. Microprocessor/Bit-Slice Design

The microprocessor/bit-slice design looked rather favorable. A basic system architecture appears in Figure 2-2. The processor would require two banks of memory. The first bank containing the algorithms to selectively "poke" video RAM with the correct information, and to maintain color tables. The first bank of memory might also contain software generators to construct straight lines and circles. The second bank of memory would be the video RAM (frame buffer), which would be dual ported to allow frame buffer updates as well as to clock out the video information to the screen.

The support circuitry would be responsible for generating the



(a) System Configuration



(b) Bit-Slice Graphics Processor
Block Diagram

Figure 2-2

horizontal and video sync signals, the video (either NTSC composite or RGB), and handling various screen attributes if required. The generation of the sync (horizonatal and vertical) signals could be done by the programmable timer, which could be easily changed depending on the screen resolution.

Since the system was only to be monochrome with perhaps three bits of gray scale, multiple color planes were not considered in the design. The incorporation of multiple planes would increase the complexity of both the microprocessor's software and the hardware to switch between planes.

The display processor would have to be a rather high speed machine in order to obtain the necessary throughput. Consider the case of the straight line Digital Difference Analyzer (DDA) algorithm residing in the display processor. The host processor would pass the endpoints of a line to the display processor through a buffer. The display processor would then calculate each point to be "turned on" and in turn, update the video RAM. The faster it could do this, the more quickly complicated objects could be "drawn" in video RAM. In order to provide animation capabilities, an object would have to be erased and then re-drawn at some other location according to the object's rate of movement. This process is called screen updating. The screen update time is dependent upon two factors: the processor speed and video RAM access time. Faster processors and/or video RAM improve the uniformity of motion that can be achieved. The uniform movement of small symmetrical objects would not pose a problem. As the number of objects

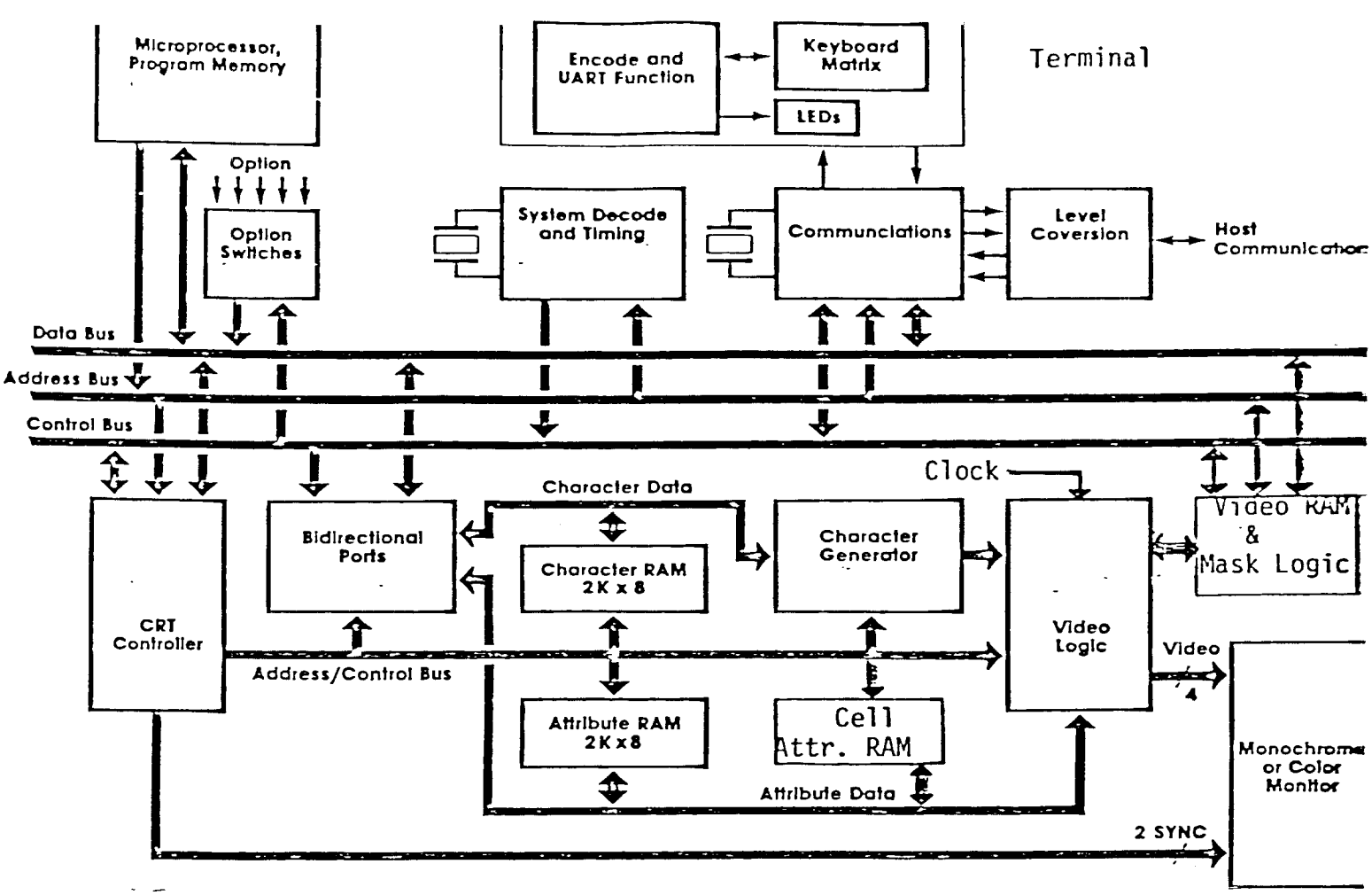
increases, an undesirable situation occurs where parts of one object move in one frame display and the remainder of the object moves in the next frame display. If the objects to be moved consists only of an outline, speed would not be a big problem. However, solid areas would require larger processor bandwidth.

At the time of design of this project, the faster 16-bit processors were just beginning to become available. Their cost was many times that of the 8-bit machines. In addition, the memory requirements were doubled, thereby increasing the memory cost. The 8 and 16 bit processors, being manufactured using NMOS technology, were not capable of running at high speeds. At that point, the bitslice technology looked promising with the AMD 29116 a likely candidate. The introduction of the VDP chip, to be described later, proved to be a better alternative.

2.1.2. The MSI/LSI Design

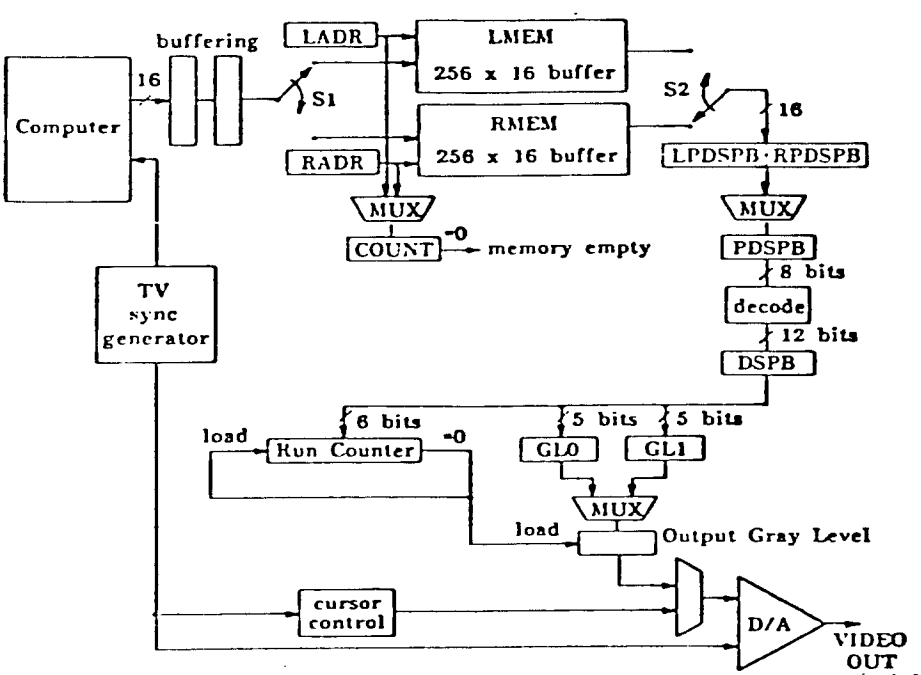
2.1.2.1. Cell Organized Displays

The MSI/LSI design approach was also considered. A block diagram of this system can be seen in Figure 2-3. It consists of a group of chips generically called "CRT controllers", the necessary hardware line generators, and video RAM controllers. This design approach yields a cell-organized display in which the screen is divided into a number of square or rectangular cells, each of which is treated like a character position of an alphanumeric display. Each horizontal row of picture cells is stored as a character string.



Cell organized display - Block Diagram

Figure 2-3a



Run length display - Block diagram showing left and right memory buffers (LMEM,RMEM), decoding registers, (PDSPB, DSPB), and gray level mapping (GL0, GL1).

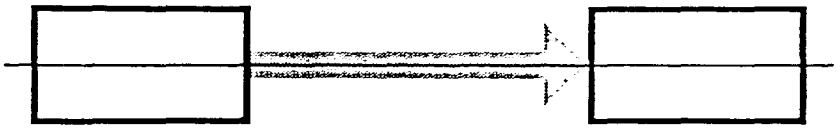
Figure 2-3b

By means of a special font, together with masking and overlay logic, the correct pattern is displayed in each cell to generate the required picture. Depending upon the cell size, a large repertoire of characters is required to display line drawings in this way. By using techniques of cell masking, concatenation, and mirror imaging of characters, the number of different characters could be reduced. The best reduction techniques yield a font of about 100 different cells.

2.1.2.2. Run Length Encoding

Another MSI/LSI implementation approach that was considered was run-length encoding. This is a technique for more compact storage of images involving solid areas of gray tone or color. An example of such an image is shown in Figure 2-4a. In examining a typical scan line from this image, it can be seen that many consecutive pixels have the same intensity. Instead of storing each pixel separately one can store the length and intensity of each run of identical pixels. Therefore each encoded scan-line consists of one or more instructions, each instruction defining a run length and intensity value (Figure 2-4b).

The use of run-length encoding can offer considerable savings in the amount of memory and memory bandwidth needed to store certain classes of images. For objects such as the one in Figure 2-4a, memory requirements can be as little as 1% of the requirements for a frame buffer. More complicated objects prove to be the weak point of run-length encoding.



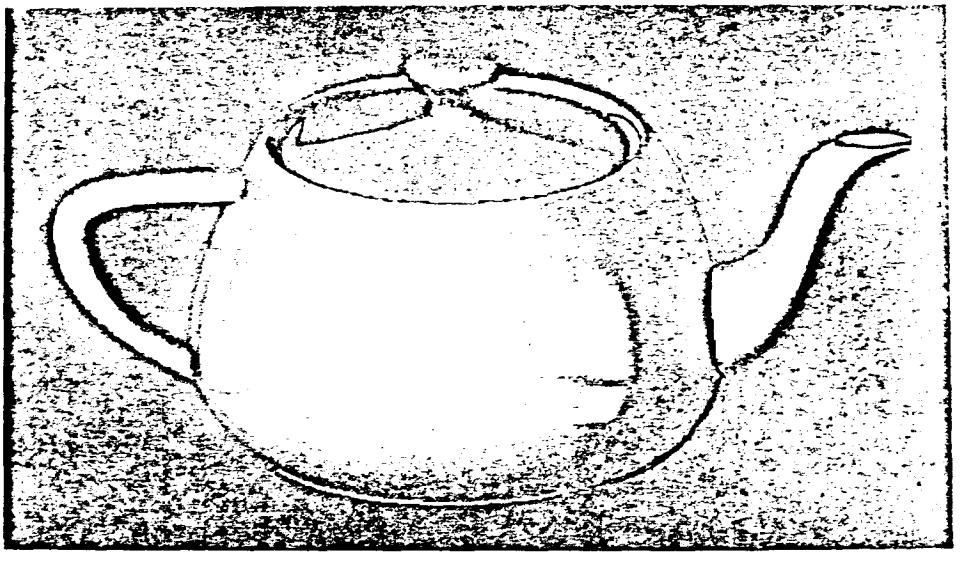
(a)

white 10	black 5	white 100	black 5	gray 160	black 5	white 100	black 5	white 10

(b)

The scan line indicated in (a) is shown encoded in (b).

Figure 2-4



A difficult object to run length encode. Note the highlights and shading.

Figure 2-5

Consider the image in Figure 2-5. The pattern of alternating on/off or grayshade/off pixels would require approximately 1.5 times as much memory requirements to store in run-length encoded format as to store in a standard frame buffer format. For each run, both length and intensity values are stored. If each pixel alternates, the amount of information required for that scan line doubles.

Neither run-length encoding nor the use of character cells is a satisfactory approach to the construction of an interactive display. Both approaches make modification of the displayed object difficult, violating a prime requirement for object animation. In run-length encoded images the sequence of run-length codes must be rearranged every time a scan line is modified, and the need for instruction sequences of variable length puts a heavy load on the free memory storage allocation/deallocation system that provides blocks of memory to store the sequences. The cell organized display has a very similar problem, although a jump mechanism to different cell fonts can make display modification easier. In addition, both the run-length encoded display and the cell organized display have performance limits beyond which they begin to distort images. Only the frame buffer can provide display capability beyond these limits. Because the display processor was to be as general as possible and was to be able to deal with almost any object and its motion, the run-length encoding and cell organized display techniques were not considered further.

2.1.3. The Video Display Processor (VDP)

The advent of ICs of higher density and increased complexity led to the introduction of the single package IC version of a video display controller. The VDP contains a number of features that make object generation, motion, and display, easier and more cost effective. During the course of the hardware design of the video system described in this thesis, the video processors that were available were oriented more towards game and terminal applications rather than graphics display functions. The Texas Instruments TMS9918 VDP, for example, is oriented towards game applications, but it contains features that, when used in unique ways, allow it to become more of a general purpose graphics engine.

A partial list of the TMS9918 VDP features are as follows:

a) Single chip interfacing to monitors (either NTSC or RGB).

b) 256 x 192 resolution, 60 hz screen refresh rate.

c) 15 unique colors, 8 gray levels.

d) Direct writing to video RAM.

e) Multiple VDP use and synchronization.

f) Thirty-five display planes.

g) Thirty-two of the thirty-five planes directly X-Y positionable.

h) Four modes of operation.

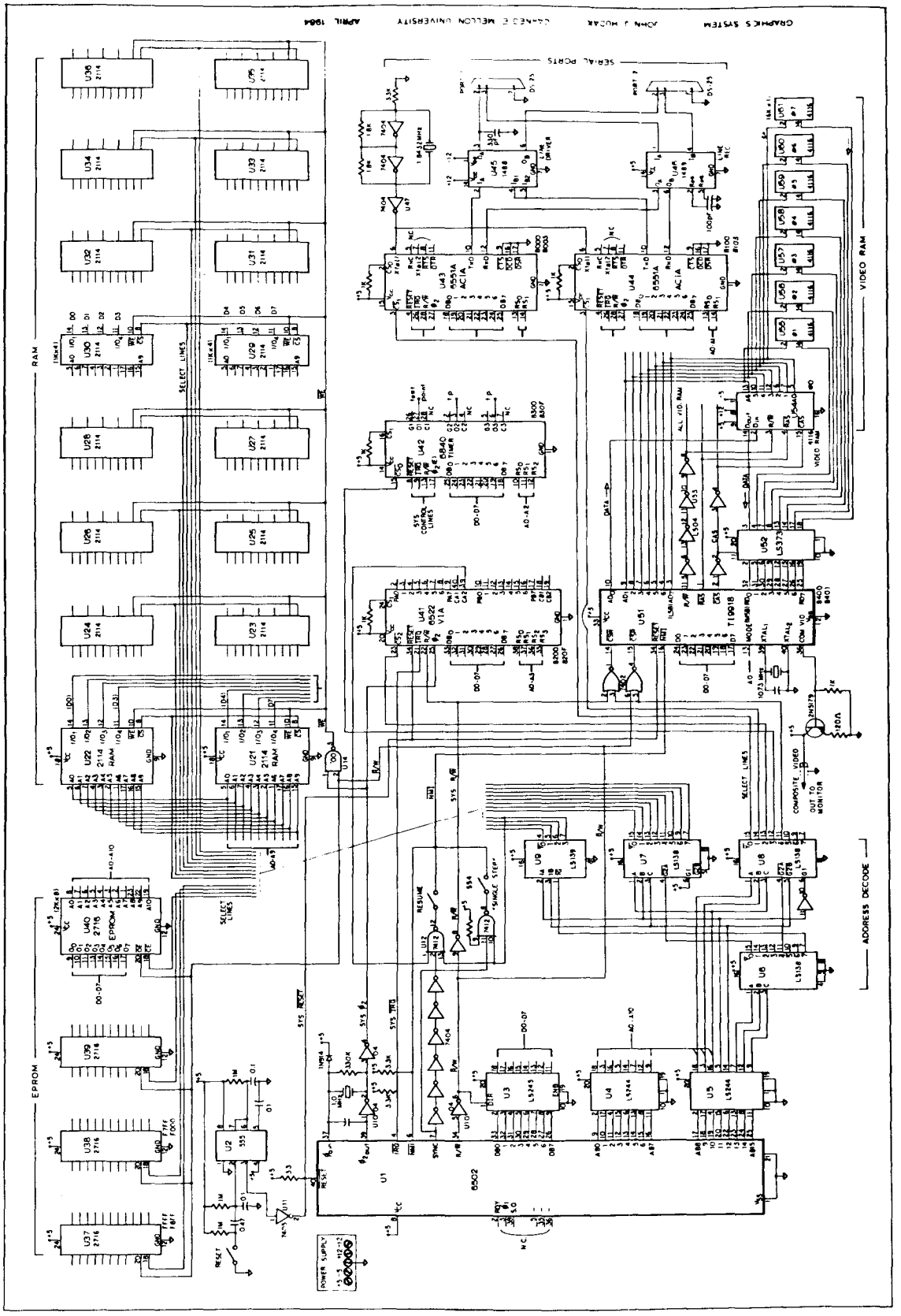
These features allowed an economical but powerful graphics system to be configured. While it is true that there are some

deficiencies with the system, the system was designed around this device because of some of its unique features, especially the display plane concept. The VDP's limitations will be pointed out in the subsequent discussion, and the solutions used to circumvent them will be described also.

2.1.3.1. System Design With The VDP

The overall system design is shown in the schematic, Figure 2-6. The 6502 microprocessor is configured with up to 8K/16K of EPROM space, 8K of RAM space, two ACIA's, a combination parallel interface/timer(2), and a multimode timer chip that contains three timer/counters. The use of the two ACIA's is somewhat obvious: the first serves as the user's terminal interface, while the second one is the host machine's interface. The terminal interface is supported by about 2K of software. This terminal handler software supports character echo, delete/rubout, horizontal tabs, and control-C traps. These features plus additional capabilities are discussed in the Software Development section.

The need for the second ACIA stems from the possibility of connecting to a host machine. The software supporting this port consists of a download program which can accept assembled 6502 code, in ASCII hex-space format, and can load it into the microprocessor's RAM. Much of the development work of the graphics software was performed using this capability.



Graphics System Schematic

Figure 2-6

The timers were required for a number of functions, but were included primarily to provide a source of motion rates. A maximum of five different objects can be moved at different rates. All of the timers are connected to the microprocessor's interrupt (IRQ) line, thus providing a fast and efficient method of updating object positions.

A timer was also required for frame buffer time slicing. This slicing is done both on the single frame itself and could also be done with multiple frames. This function will be discussed in a later section.

As indicated by the schematic, the VDP is memory mapped into the microprocessor's address space. The VDP can be read and written through a sequence of successive accesses. For example, to write to video RAM, the host processor must first write the high byte of the video RAM to the VDP address followed by the low byte of the video RAM address, and then the desired data. Similarly, a read of video RAM is performed by two successive writes, providing the video RAM address, followed by a read from the VDP. One must ensure that this process is not interrupted by another read or write operation because these sequences are not reentrant. A complete data sheet for the VDP is included in the Appendix. Although many aspects of the VDP's operation will be discussed in this paper, the reader is encouraged to read the VDP data sheet to gain a full understanding of its operational characteristics.

The VDP can interface directly to 4K or 16K dynamic or static RAM's. If VDP is initialized to interface to dynamic RAMs, it

automatically produces the RAS and CAS refresh signals. Any display updates are done without any interruption of the refresh sequence. Also, the refresh is interleaved with video RAM display fetches. This interleaving yields a flicker free screen update at 60 hertz without missing any RAM refresh.

The VDP maps color (or black and white) information onto the pixel information. The resultant signal is National Television Standards Committee (NTSC) format and may be directly input to a monitor or a RF modulator for use with a standard television receiver. The hardware consists of a transistor buffer that isolates, amplifies, and provides impedance matching for directly connecting the signal to a video monitor. In addition, two or more VDP's can be operated in parallel with their composite video signals synchronized for multiple, 35 plane displays. This is one way in which some of the hardware limitations of a single chip system can be overcome. A specific example is the display of multiple "sprites" on one scan line. A sprite, as defined by TI, is an eight by eight pixel cell that can be moved by simply changing the X and Y coordinate bytes associated with it. The VDP has a limit of four sprites on any one scan line. If a fifth sprite is moved onto that scan line, it will not be displayed and the fifth sprite bit in the VDP status register will be set. Multiple VDP's would greatly reduce or eliminate this situation.

2.1.3.2. Video Memory Organization

The TMS9918 VDP has seven read/write registers and one read-

only status register. The registers are as follows:

R0,R1	VDP control/option registers
R2	Name table base address
R3	Color table base address
R4	Pattern generator base address
R5	Sprite attribute table base address
R6	Sprite pattern generator base address
R7	Text mode color generator
R8	VDP status

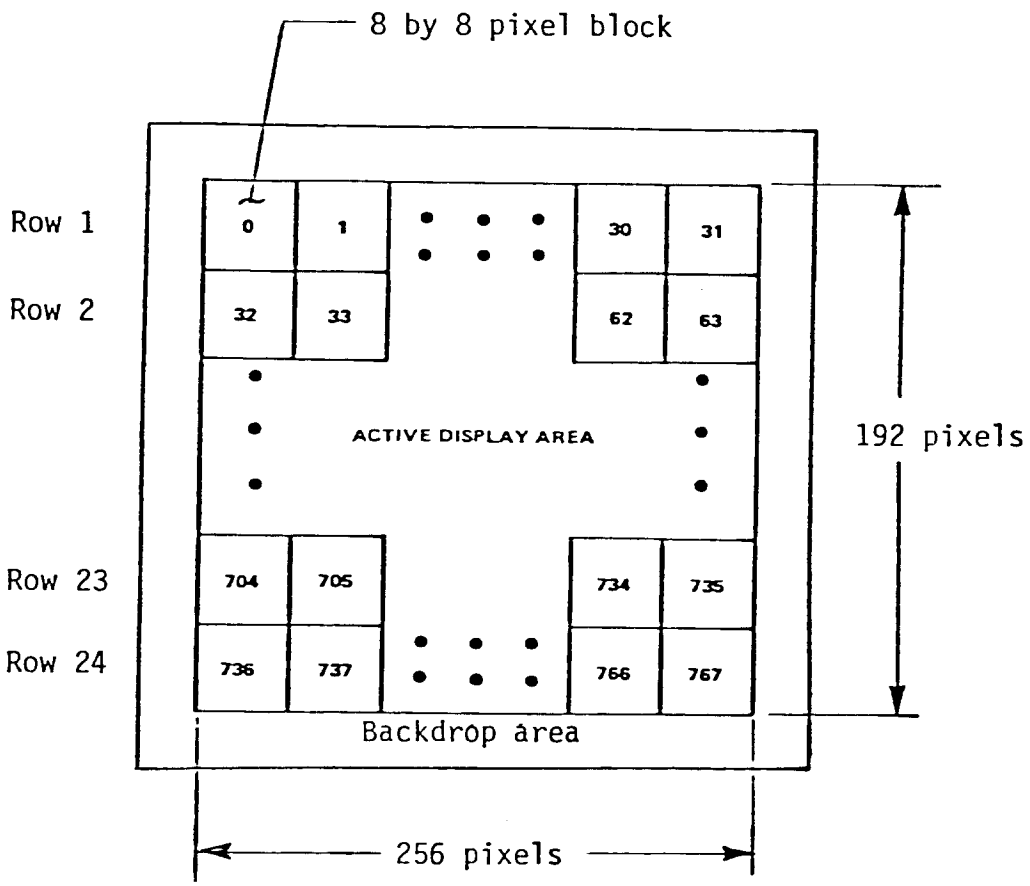
Registers R0, and R1 set up the VDP for video RAM space, sprite size, and screen blanking. They are generally initialized at the beginning of a program and are not often rewritten later. In addition, one of four possible graphics modes is configured in R1.

Registers 2, 3, and 4 are used in conjunction with each other to name, define, and color an object. According to VDP specifications, R2 must contain an address in video RAM that is a multiple of $400. R3 must contain an address in video RAM that is a multiple of $40, and R4 must contain a multiple of $800. To describe how an object is constructed and displayed we will use the following example. In this example, it is desired to turn on one pixel at some location on the screen. First, R2 and R3 must be initialized, but to do this, one must know the pixel-to-screen mapping that the VDP uses.

The screen is divided into 768 8x8 pixel pattern positions

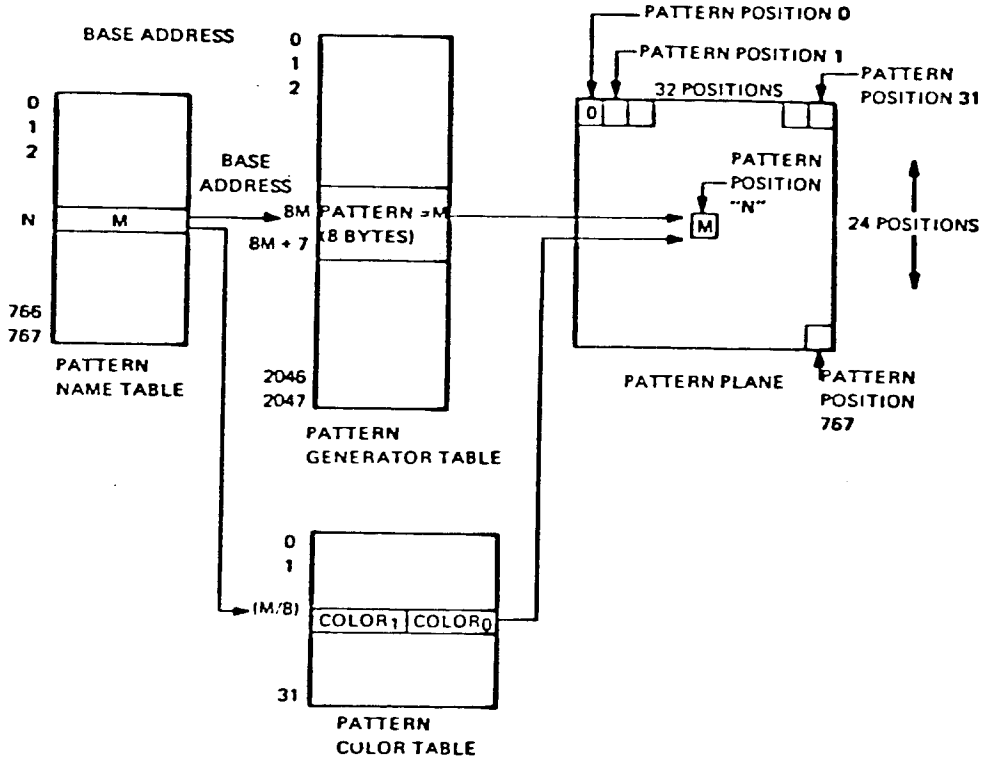
starting with zero in the upper left hand corner (see Figure 2-7). There are 32 pattern positions horizontally and 24 vertically resulting in a display area of 32x8 by 24x8 or 256x192 pixels. The image to be displayed in any one of these cells is stored in binary form in the pattern generator table. The base address of this table in video RAM is in R4. Each group of eight bytes in the pattern generator table is associated with two colors that are stored in the pattern color table, whose base address is in R3. This table is 32 bytes long, the first byte containing the two allowable colors for pattern positions 0-7, the second byte containing the two colors for pattern positions 8-15, etc., with byte 32 containing the colors for pattern 248-255. It is impossible therefore to use a single VDP to produce more than three distinctly colored pixels within a group of eight bytes. This color mapping limitation and one possible solution will be discussed later in this thesis.

The pattern name table contains a numeric pointer value to a pattern in the pattern generator table. As indicated in Figure 2-8, the pattern generator table is 2048 bytes long and is logically divided into eight bytes per pattern name yielding the 8 X 8 cell described earlier. (Again, a possible limitation for direct bit-mapped graphics exists here). For this example, suppose that R2 is loaded with $0. This places the base of the pattern name table at $0000 in video RAM ($0*$400=$0000).



Video Display Screen Mapping

Figure 2-7



Pattern Name Table Register (R2) = 0, VRAM Addr. = $000-$2FF
Pattern Generator Tbl. Regis. (R4) = $1, VRAM Addr. = $800-$FFF
Pattern Color Table Register (R3) = $C, VRAM Addr. = $300-$31F

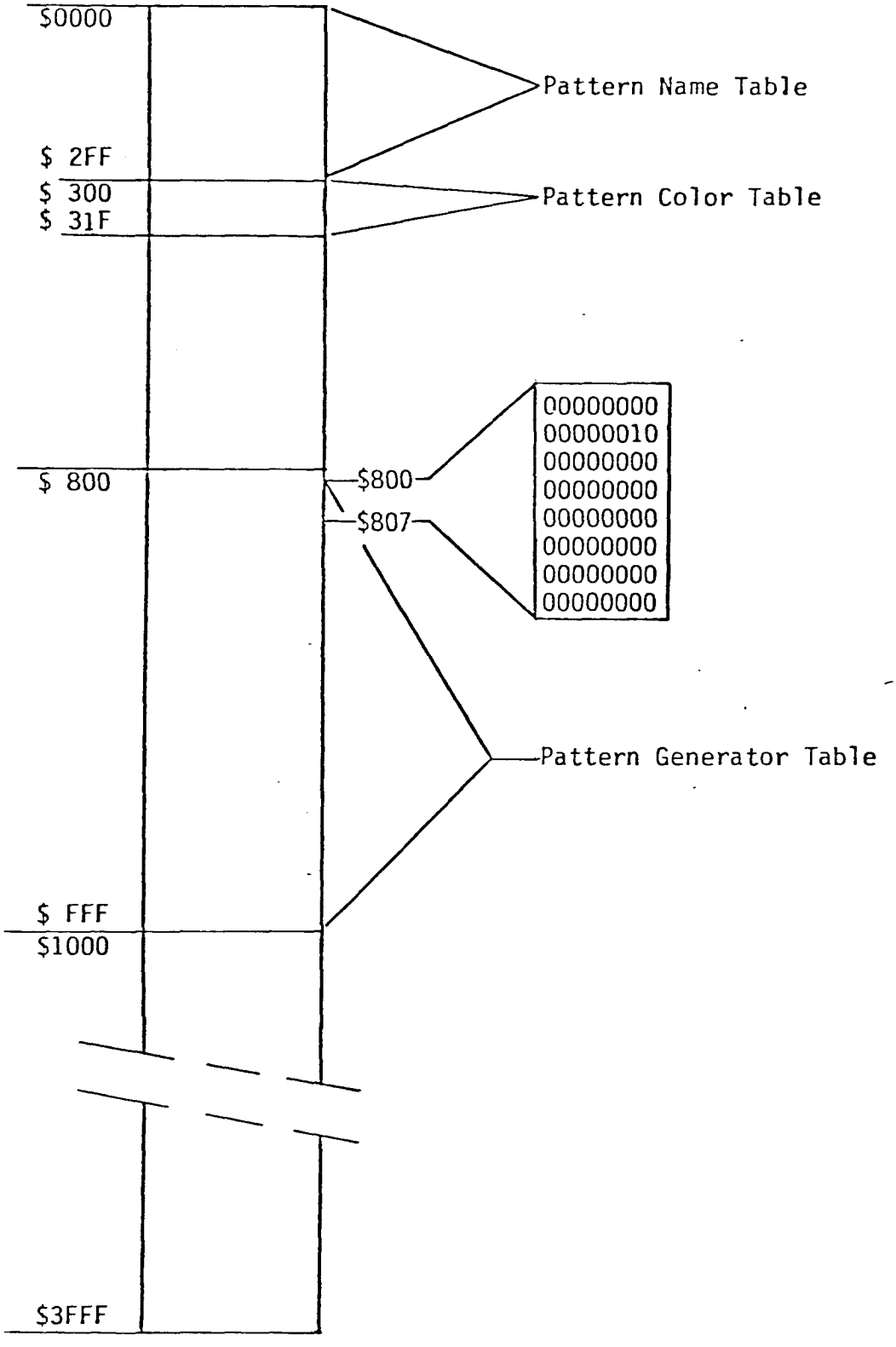
Video Display Processor Table Mapping Scheme

Figure 2-8

The pattern generator register (R4) is loaded with $1, placing the base address of the pattern generator table at $800 hex ($1*$800=$800). The color table (R3) is then loaded with $C which places it's base address at $300 ($C*40=$300). This produces the video RAM map indicated in Figure 2-9.

Keep in mind that the goal is to turn on a single pixel in the upper left hand position (cell). To do this, when multiplied by eight, location 0 in the pattern name table is loaded with a 0. This 0 provides an index into the pattern generator table. The pattern generator table location $800-$807 is loaded with the desired pattern. If only one pixel is to be on, then all but one bit of the eight byte cell will be zeros. At this point, the pattern and the position on the screen have been defined; the desired color must still be defined.

The color block is composed of 32 bytes. The lower nibble of each byte defines the color of the zeros and the upper nibble defines the color of the ones. There are sixteen possible colors available, which can be represented as shown in Figure 2-10. There is another limitation revealed here. In order to map all of the colors onto the screen, the pattern table should contain at least 768 bytes. To accomplish complete pattern plane mapping, each byte in the color table would govern the color for eight cells. That is, byte 0 in the color table colors pattern plane cells 0-7, byte 1 colors cells 8-15, etc., until byte 31 colors cells 760-767. Therefore, to conclude this example, video RAM location $300 would be loaded with 1F (all 0's are black, all 1's are white).



Video RAM Map of Example

Figure 2-9

COLOR HEX	COLOR	TMS9918A		TMS9928A/9929A		
		LUMINANCE (DC) VALUE	CHROMINANCE (AC VALUE)	Y	COLOR DIFFERENCE R-Y	B-Y
0	TRANSPARENT	0.00	-	-	-	-
1	BLACK	0.00	-	0.00	.47	.47
2	MEDIUM GREEN	.53	.53	.53	.07	.20
3	LIGHT GREEN	.67	.40	.67	.17	.27
4	DARK BLUE	.40	.60	.40	.4	1.00
5	LIGHT BLUE	.53	.53	.53	.43	.93
6	DARK RED	.47	.47	.47	.83	.30
7	CYAN	.67	.60	.73	0.00	.70
8	MEDIUM RED	.53	.60	.53	.93	.27
9	LIGHT RED	.67	.60	.67	.93	.27
A	DARK YELLOW	.73	.47	.73	.57	.07
B	LIGHT YELLOW	.80	.33	.80	.57	.17
C	DARK GREEN	.46	.47	.47	.13	.23
D	MAGENTA	.53	.40	.53	.73	.67
E	GRAY	.80	-	.80	.47	.47
F	WHITE	1.00	-	1.00	.47	.47
-	BLACK LEVEL	0.00	-	0.00	.47	.47
-	COLOR BURST	0.00	.40	0.00	47(28A) 73(29A)	.1(28A) .2(29A)
-	SYNC LEVEL	-0.40	-	-.46	.47	.47
-	EXTERNAL VIDEO	-	-	0.00	.47	.47
-	LEVEL	-	-	0.00	-.46	-.46

VDP Color Assingments

Figure 2-10

2.1.3.3. Limitations and Solutions

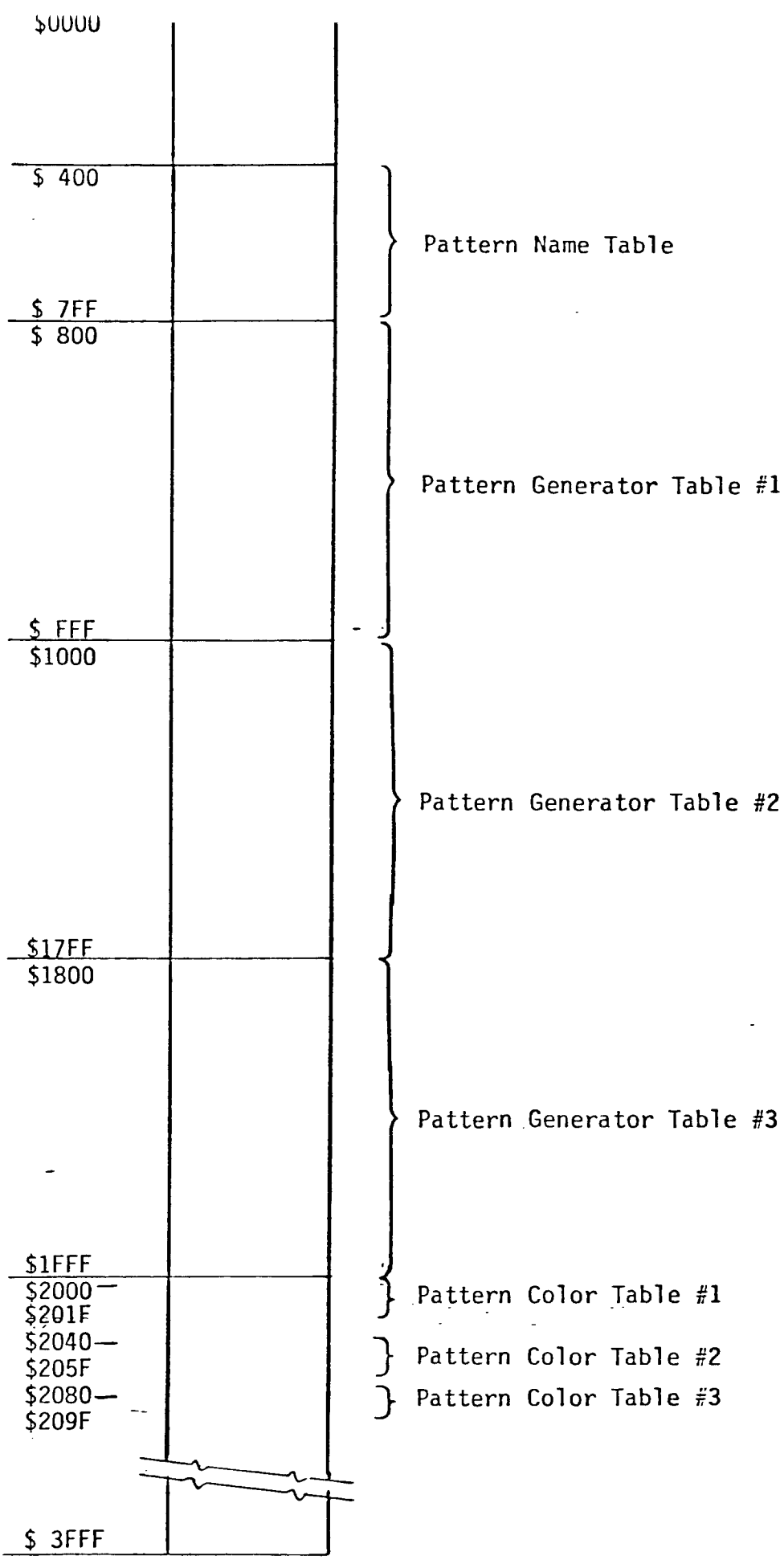
Several aspects of using the VDP were brought out in the above example. Some are good, and some are not as favorable depending upon what is to be accomplished. One good feature of the VDP is that to fill the screen with one eight-by-eight pattern, the pattern must only be made once. All the entries in the pattern name table would point to the single pattern. Two very important points about the possible limitations of the VDP were brought to light in the above discussion. If one considers the application of the VDP in a totally bit mapped graphics machine, it can be seen that the hardware does not easily accommodate this. The first shortcoming is the length of the pattern generator table. If one wishes to generate different patterns in each pattern position on the screen, the length of the pattern generator table must be 768 x 8 or 6144 bytes long. However the pattern generator table can be only one third this size.

The second limitation is the size of the color table size. In order to define the color of each 8 x 8 pattern plane cell independently, the length of the color table must be 768 bytes. To define the color of each pixel individually, the color table needs to be 256x192/2 = 24.5K bytes. In actuality, any one of a number of mapping techniques could be used to significantly reduce this size. Regardless, these are two limitations that must be taken into consideration.

2.1.3.3.1. Pattern Table Length Solutions

The approach used in this project to solve the pattern generator length problem consisted of time slicing the video display. This solution uses the fact that although the pattern generator table is only one third the actual length needed, by simply changing the base address of the pattern generator table base register (R4), a completely new set of patterns can be displayed. The register modification can easily be done in the 16.2 uS which is the sum of the times for horizontal retrace and displaying the boarder on the VDP. To implement this solution, the pattern name table is filled with 0 to 255, indicating 256 different patterns. Three pattern generator tables were then created adjacent to each other so that register updates amounted to nothing more than incrementing R4. The video memory map for the current system implementation appears in Figure 2-11.

The time slicing was implemented by use of a timer and the vertical sync pulse pin on the VDP. The vertical sync pin was connected to the NMI line of the 6502. Whenever an interrupt from the VDP occurred, the first timer value was loaded into the 6522 and the VDP pattern generator register was loaded with $1. This pointed to video RAM locations $800 to $FFF. Whenever this timer timed out, it generated an interrupt. The second value was loaded in the timer for the second third of the display screen, and the pattern generator register was updated. The next interrupt caused by the timer would signal the processor to change the pattern

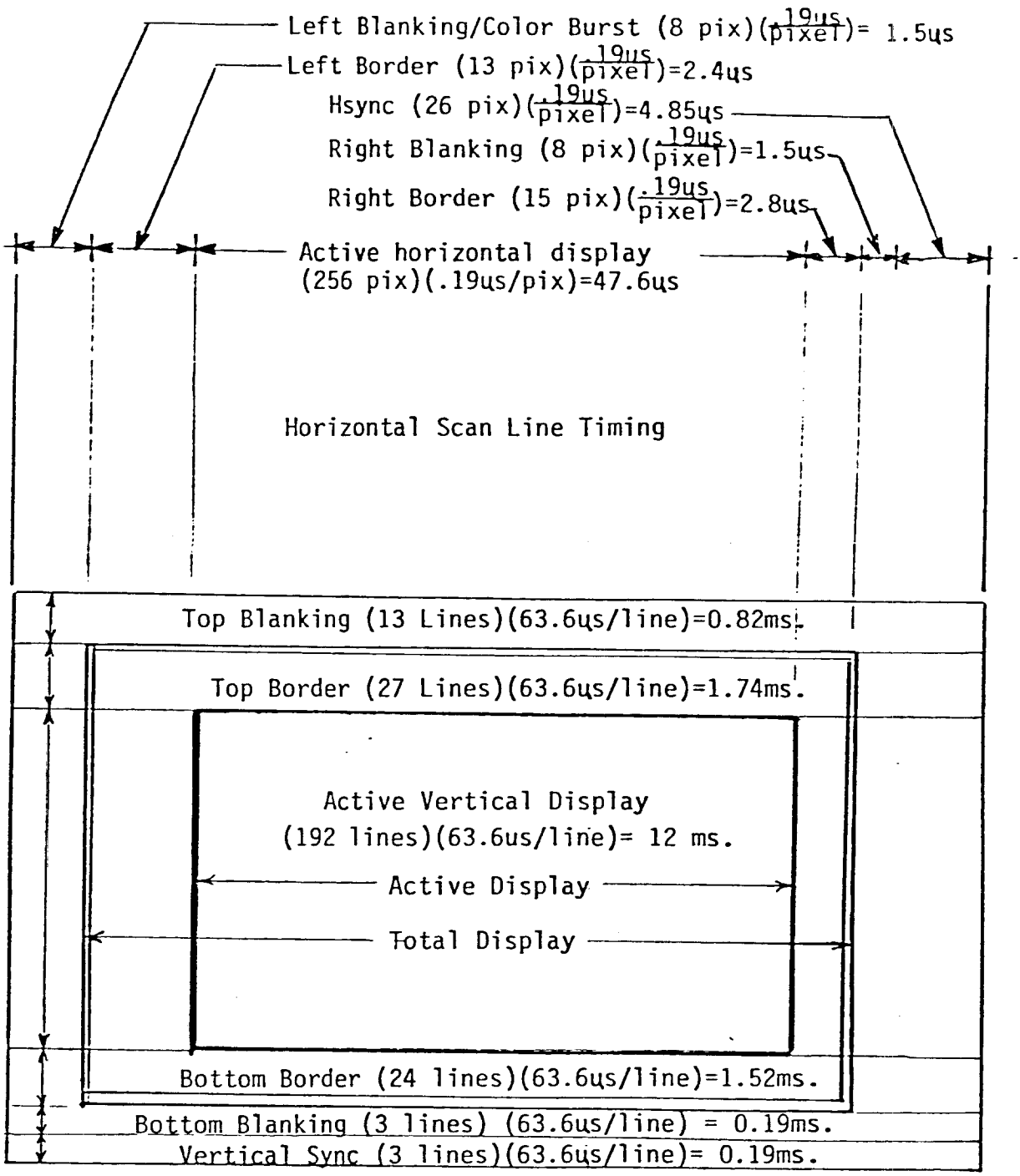


System Video RAM (VRAM) Map - Figure 2-11

generator table base register to point to the location of the last third of the display screen. Notice that the timer would not have to be reloaded because the next interrupt would originate from the VDP, which is attached to the NMI line.

A very specific device polling order is required to make this scheme work. First, all actions of the interrupt service routine should be as short as possible. Second, the very first device tested should be the timer. If the pattern generator table is not changed immediately, picture jitter will occur, depending upon how much service time is required. Actually, the maximum time to update the register is 13.05 uS. To have the maximum time to update this register, the update cycle must begin at the beginning of the display of the right border and must be complete by the end of the left border (see Figure 2-12). More time can be allotted for changing the entire screen at the end of an old one by starting the change after the last or 256th scan line of the active display area. This time period is approximately 4.5 ms.

Another useful feature of this technique is that multiple planes can be displayed. No matter how many different frames are displayed, the refresh rate is still 60 hz. This fact can make the movement of objects much easier: while the first plane is being displayed, the second plane can be updated. This method, although acceptable but somewhat restrictive with respect to the possible directions of motion that can be displayed, is not advisable if sprites (to be described later) are going to be used.



Vertical Scan Line Timing

Figure 2-12

2.1.3.3.2. Color Table Limitations

The problem with the color table can now be addressed. Since independently shaded pixels were not required, the limitation of the pattern color table size was not addressed in the software written to date. There are however, some compromises that may be made to accommodate color. First of all, every pixel in a predefined 8x8 pixel cell can have only one of two colors. No "tricks" can be used to get around this. In addition, trying to time slice color table pointers every eight pixels to get different colors is impossible due to time constraints. The best possible compromise is to allow the use of only two colors, referred to as the foreground and background colors, throughout the entire screen. If multiple, parallel VDP's were used, the idea of color planes could be utilized. Basically this would result in one plane defined as green, one as red, and the other blue. In order to draw a white line, all three planes would contain the line. A number of colors, up to a maximum of eight, could be displayed according to the rules of color mixing.

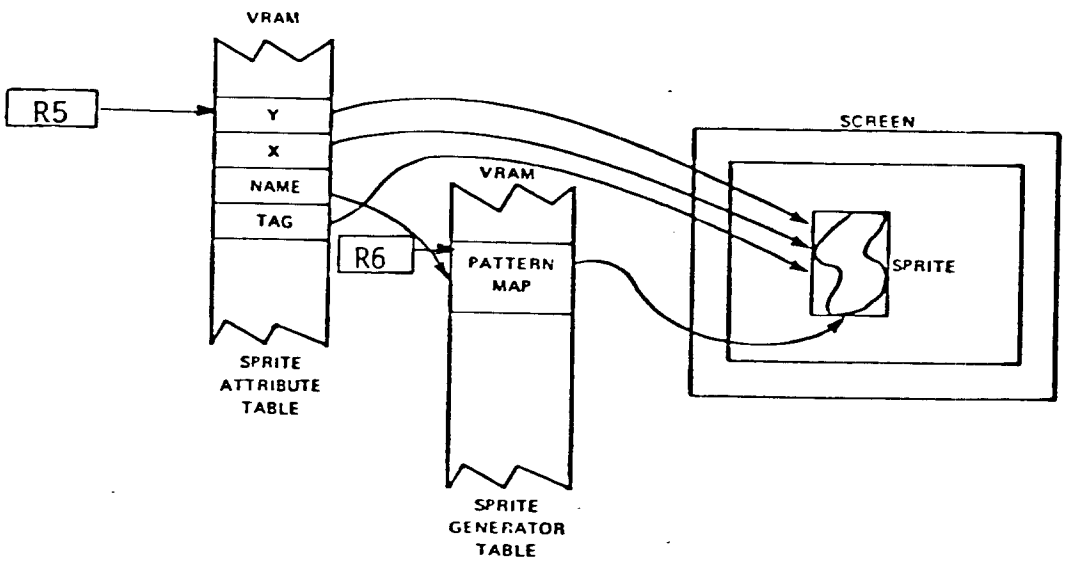
A point should be brought up here. After the software was designed to accommodate the pattern generator table time slicing, an improved version of the 9918 (9918A) was released. One of the improvements was to provide a means of making the pattern generator table 6144 bytes long. To deal with the capabilities of this new version, the software could be modified, but the original time slicing method should work with the new version also.

2.1.3.4. Sprites and Their Use

Sprites are 8x8 cells having a number of special attributes that make them useful for animation. Sprites are defined in the same basic manner as patterns are defined. There are two registers that govern the area in video RAM where sprites are stored. The sprite attribute table base address is stored in register five (R5) and the sprite pattern generator base address is stored in register six (R6) of the VDP (Figure 2-13). The effective video RAM address of the sprite attribute table is obtained by multiplying the contents of R5 by $80, (R5*$80=video RAM address), and the effective video RAM address of the sprite pattern generator table is obtained by multiplying R6 by $800, (R6*$800=video RAM address). There is a maximum of 32 sprites allowed by the VDP.

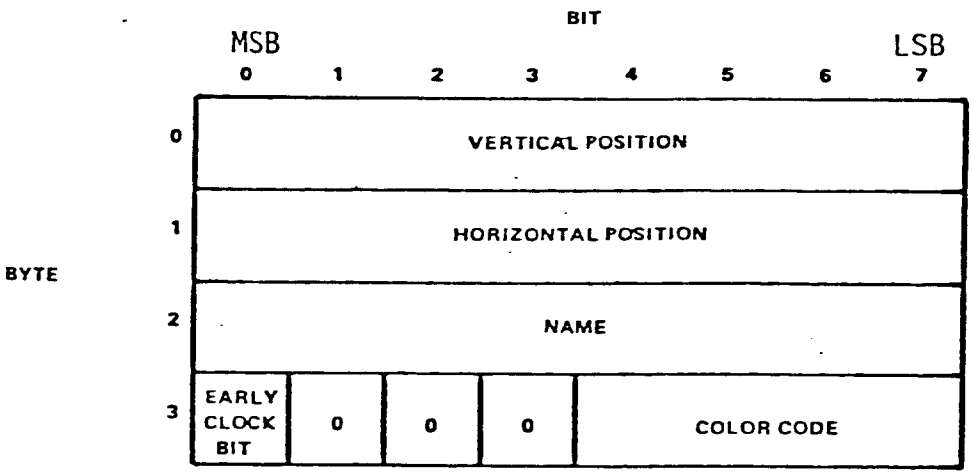
2.1.3.4.1. Sprite Pattern Generation

Sprites are built in much the same manner as other patterns. The sprite attribute table is composed of four bytes of information per sprite (Figure 2-14a). Therefore, for 32 sprites, the attribute table is $80 bytes long. The first byte is the screen Y position in pixels, the second byte is the X screen position in pixels, the third byte is the sprite name, and the fourth byte is the tag. A sprite can be moved simply by changing the contents of either the X or Y position bytes in its attribute table. The update of a sprite's position is done at the end of each frame display, thus preventing a sprite from being partially moved.



Sprite Mapping

Figure 2-13



a) Sprite Attribute Block

SIZE	MAG	AREA	RESOLUTION	BYTES/PATTERN
0	0	8 × 8	single pixel	8
1	0	16 × 16	single pixel	32
0	1	16 × 16	2 × 2 pixels	8
1	1	32 × 32	2 × 2 pixels	32

b) Sprite Magnification Factors

Figure 2-14

The name byte in the sprite attribute table refers to the sprite number. To obtain the actual video RAM location where the sprite pattern is stored, the value of the name byte is multiplied by four. The resulting index is then added to the base address of the sprite generator table, to point at an eight byte array which contains the actual sprite pattern definition. As in the other types of pattern generation, asserted bits (1) define an "on" pixel and unasserted (0) bits are "off".

The color of the sprite can be any one of the 16 colors provided for by the VDP (Figure 2-10). The color attribute field is the low nibble of the fourth byte of the four block sprite attribute table. Sprites with the same color as the background color appear invisible.

Sprites can also be magnified in one of three ways. Basically, the magnification factor can result in a sprite size quadrupling to 16 x 16 while still maintaining the one to one pixel mapping, or the sprite size can increase by a factor of 16, to 32 x 32 pixels, in which case there will be a one bit to two pixel mapping (Figure 2-14b). The magnification factor, which can be set by the appropriate bits in control register one, can be very useful for the generation of larger objects. No matter what magnification factor is selected, the sprites maintain an ordered priority when displayed. That is sprite number one has the highest priority, followed in order of decreasing priority by sprite number two, sprite number three, etc. This prioritization is important because an overlap condition may occur in the process of moving sprites

around. If sprite one overlaps sprite three, the overlapped portion of sprite three will be covered by sprite one and will become invisible. If sprite three should overlap sprite ten, the overlapped portion of sprite ten will be covered by sprite three. In the current configuration of the system, sprite zero is the cursor. The remaining unused sprites, are given by default, numbers $FF.

2.1.3.4.2. Sprite Motion

Several aspects of using sprites can be explored. First, to use sprites for object animation, a sprite usage table could be constructed which would present to the "operating system" an allotment of sprites available to construct an object. Second, in order to obtain the maximum benefit from the sprite, a magnification factor of 16x16 or 32x32 would have to be selected. Third, in order to compose larger objects than, say 16x16, several sprites could be concatenated to act as one large object. Since a sprite is moved by updating its X and Y location bytes, a composite object would have to be moved by changing all the X and Y bytes of the sprites of which it was composed. However, this would have no noticeable effect on the observed object motion.

In making composite objects from sprites, it is generally a good idea to use sprites in succession to avoid mixing the display priority of the composite objects. To illustrate this, suppose a square that was composed of sprites 2, 3, 4, and 10 ran over a rectangle composed of sprites 8,9,11,12,13, and 14. Depending upon

where the two composite objects overlapped, a part of the rectangle could show through part of the square, when it was actually desired to have the entire square at a higher priority than the rectangle. Consecutive groups of ordered sprites used to produce a composite picture would reduce the partial "show through" problem.

The preceding paragraphs reveal some attributes of sprites that must be considered when using them to compose large object. To effectively structure a large object some restrictions and prior information about the object to be constructed would have to be in effect. For example, if a triangle were to be constructed, a triangle command keyword would have to exist rather than requiring the user to perform a piecewise construction of a triangle. Also, since the triangle is being constructed with the intent of having it be able to be moved, a special animation attribute would have to be added to the command line. This way the object generation algorithm would know to use the sprite "free list", and construct the triangle of consecutive free sprites.

Depending upon the application, some of these limitations may be difficult to live with. A possible solution is to add on additional parallel VDP's. If another two were added, a total of 96 sprites could be generated. Since the composite video of the three VDP's would be mixed, the 32 sprites of VDP1 would have the same priority as those of VDP2 and VDP3. As a result of this paralleling, the software for object construction, sprite free list management and sprite movement would be somewhat more complicated, but more flexible.

2.1.3.5. Text Information

The VDP has a text display mode available to the user. When in this mode, the use of sprites is not permitted. In order to display alphanumeric characters, it is suggested that a character font be established in system EPROM to directly load the characters into the pattern display area. This method has a few other advantages. First, programmable character fonts could be made available to the user. Second, character attributes such as double height, or slanting could be implemented in the display string command. Currently, no text display is implemented on the system.

2.1.3.6. Object Mapping Algorithm

2.1.3.6.1. Point Mapping Into Video RAM

As can be seen by the previous discussion on the architecture of the VDP, it is not exactly a bit mapped graphics engine. In order to generate objects in the three pattern generator tables, a fast and efficient mapping algorithm to map the X,Y coordinates to a video RAM address had to be developed. Each time the object generation algorithm derived the X and Y coordinates of a point, it would immediately call a subroutine called MAP. The function of MAP is to map the XY coordinate into the video RAM pattern generator table, one pixel at a time. During the development phase for this software, a temporary table was created into which the generated XY coordinate pairs were stored, after which they were "dumped" to the mapping routine. The advantage of this approach was that a faulty object generation routine could be debugged by

examining the coordinates it generated. The drawback of this method was that extra overhead was placed on the system in terms of subroutine jumps and returns. Two additional subroutines had to be written, one to put values into the table, and a second to remove them from the table. In effect, this doubled the mapping subroutine overhead and increased the processing time. Also, a table had to be created in RAM to hold the generated points. Depending on the complexity of the line to be drawn, a large number (>800) of points could be generated. This would translate directly into the same number of bytes that would comprise the table. Since memory is at a premium in small systems the creation of the table to temporarily hold the X,Y points for any purpose other than debugging would be a waste of RAM.

2.1.3.6.2. Mapping Algorithm Implementation

The problem of how to set the correct display bit on or off has at least two known solutions, but the one described here is more efficient. The screen can be divided into two numbers of eight bit by eight bit squares as shown in Figure 2-15. For the purposes of discussion, the effective base address of video RAM is $800, but this bit mapping algorithm is independent of what base address is used. The screen is decomposed into twenty-four rows by thirty-two columns. With each row, pixel addresses range from 0-255 plus the video RAM base address. The following algorithm is used:

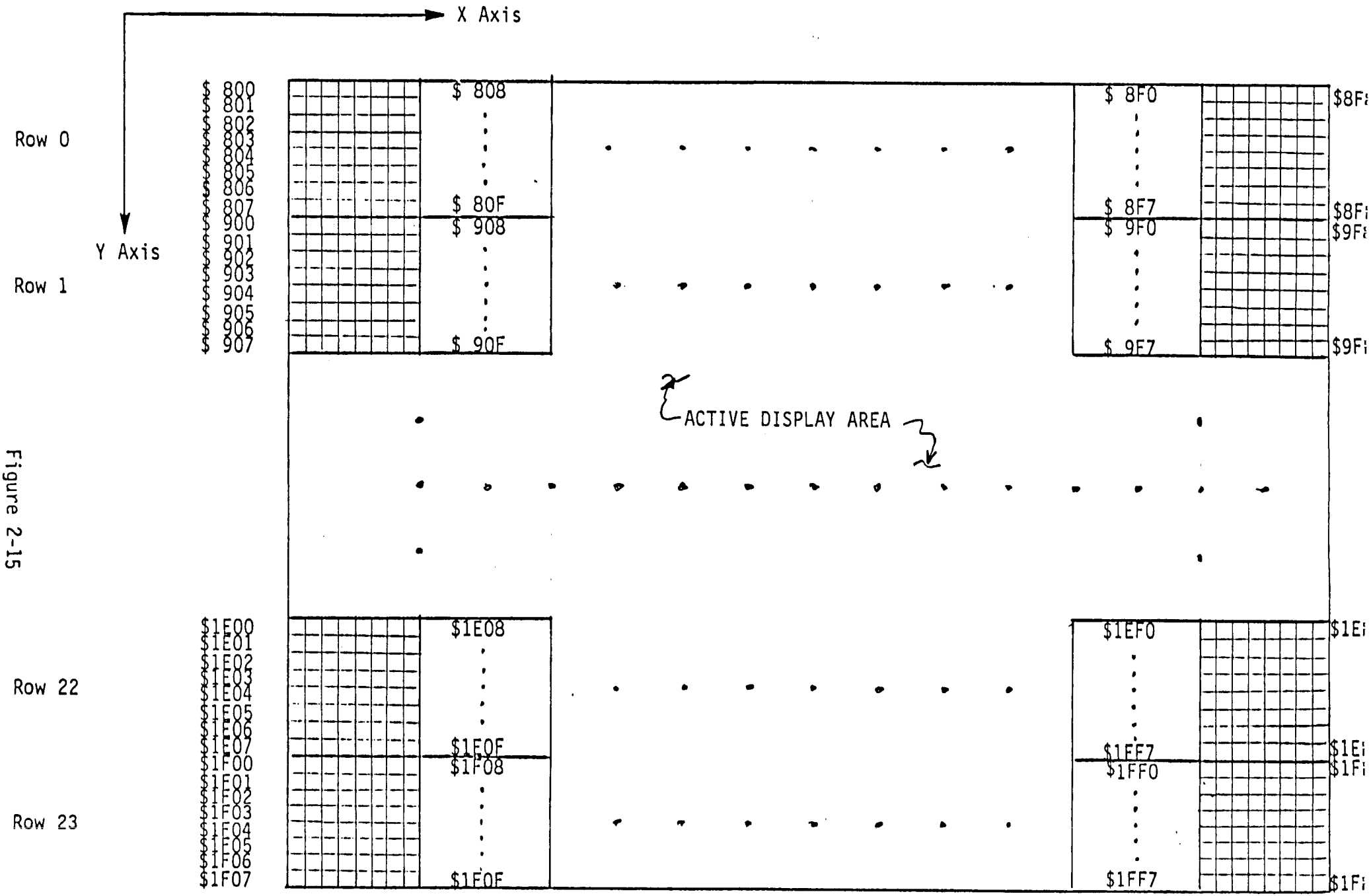


Figure 2-15

Video RAM to Display Screen Mapping

a) Locate the row (Y coordinate) by using the specified Y coordinate.

b) Locate the column (X value) by using the specified X coordinate.

c) Locate the byte within the cell by using the three least significant bits of the specified Y coordinate.

d) Locate the bit within the byte by using the last three significant bits of the specified X coordinate.

e) Get a bit mask of the bit to be turned on or off by using the last three significant bits of the specified X coordinate as an index into a lookup table. Example: last three bits of X are 011, then the mask is 00000100.

The assembly language routine works in the following manner (see Figure 2-16): First the Y row address is obtained by dividing the Y coordinate by eight and storing the result in VADH. The three least significant bits of the X coordinate are masked off, yielding the X location of the eight by eight cell on the screen. This result is added to the three least significant bits of the Y coordinate. This intermediate result is added to the base value of video RAM resulting in the actual address in the video RAM to be modified. The bit to be changed is derived by a table lookup using the three least significant bits of the X coordinate as an index into a "one of eight" table. This value, along with the fetched value from video RAM, are EXOR'ed together and then re-written to video RAM. The execution of this routine takes approximately 70 micro seconds.

The routine turns bits on or off by ex-oring the current value of video RAM with a mask. If the bit to be turned on is already on, that bit is erased. For example, if a circle command had been issued previously and a circle was displayed on the screen, if an

identical circle command were reissued, the circle would disappear. If one object intersects another object, the point of intersection will be seen as an "off" pixel.

The reasons for this choice of method of pixel illumination were to maintain consistency and to ease implementation of future enhancements. The command to draw or erase an object is the same, therefore no special "undraw" commands would have to be devised and remembered by the user. In addition, an "undo" key could be implemented that would erase the previous command. Obviously, the command buffer would have to be multiply buffered to do this, but this concept has additional benefits which will be described further in the Software Design section of this paper.

```
;
; Subroutine MAP (MAP) takes the corresponding X,Y coordinate pairs and
; maps them into the VDP VRAM.  See the documentation for how the
; VRAM is divided for various display modes.
;
; Called by routines: VECDWG
; Calls routines: GETTBL
;
; Registers modified: A, X
; Registers not modified: Y
; Variables possibly affected: XPIX, YPIX, TEMPP, PBASE, VADL, VADH
; Error codes possible (HEX):
;
;
map:    pha                     ; Save the A and X registers.
        txa                     ;
        pha,                    ;
        tsx                     ;
        lda     $105,x          ; Else load the Y coordinate and start to dec
        lsra                    ; Find the Y'th row (0-24) by Y/8.
        lsra                    ;
        lsra                    ;
        sta     vadh            ; And store it in the high address byte.
        lda     $106,x          ; Get the X value and mask off the lower 3 bi
        and     #$f8            ; This gives us the x'th column position.
        sta     tempp           ; Squirrel that away for the time being.
        lda     $105,x          ; Get the y value back
        and     #$07            ; The lower three bits tell us how far we are
        ora     tempp           ; from the base.  Add it to the index calc.ab
        clc                     ; Now add it to the base (See documentation)
        adc     pbase           ; To get the low byte of the absolute VRAM ad
        sta     vadl            ; Store it in the WVRAM routine low byte addr
        lda     vadh            ; Get the high byte to transform it,
        adc     pbase+1         ; Add high byte + Pattern Gen Base(high)+c
        sta     vadh            ; Store it in the high byte of WVRAM.
        lda     $106,x          ; Get the x position again for the actual bit
        and     #$07            ;
        tax                     ;
        lda     bittbl,x        ; Map 0-7 into one of eight bits to be turned
        sta     temp            ; Store the bit to be turned on for second
        jsr     rdvram          ; Get the present bit pattern
        eor     temp            ; EOR in the new bits.
        tax                     ; Stick it into X reg for WVRAM.
        jsr     wrvram          ; and go write (turn on) the bit!
        clc                     ;
        pla                     ; Retrieve and restore the X and A regs.
        tax                     ;
        pla                     ;
        rts                     ; bye
;*****************************************************************************
;
```

Figure 2-16

3. Software Design

3.1. Overview

During the design phase of this project the need for a large number of system level and graphics level software primitives was foreseen as a necessary part of a fully functional graphics system. However, due to time constraints the scope of the project could not encompass all the features that one would desire. The ideas presented in this section, although not all implemented, were taken into consideration during the design and implementation. The software was written in such a manner so that future work would mesh more easily with the existing code. (Throughtout the Software Design Section of this thesis, detailed operation and implementation of the software design will be discussed from time to time. The purpose for this is two fold. First, it offers considerable insight to the interested reader about the structure and software concepts (i.e. subroutine/coroutines, parameter passing, etc.) used during the implementation phase of the interpreter. Second, if future modifications and enhancements are made to the software, enough information is avaliable to the user to make this task less difficult. This document, in conjunction with the commented source code, should make the task of software enhancements more manageable.)

The design of the software used the method of top-down design. The top-down decomposition took the form of starting with a specific area or task, and subdivided it into a number of smaller tasks that when properly assembled, would porduce the desired

result. The result of this methodology used 6502 assembly code to implement the tasks and subtasks. An example of a small, low level task is checking the range of a numerical value that is passed to the subroutine. An example of a larger task is lexical scanning.

The manner inwhich new commands may be added is made simpler by this design approach. To add additional commands, the programmer would determine the tokens required, structure the parser driver subroutine to get the tokens in the proper sequence, and then implement the algorithm to utilize the acquired tokens.

Throughout the subsequent discussion of the software development, the reader may find it helpful to refer to the subroutine flow diagrams (Figure 3-5, 3-9, 3-10a, 3-10b), and the actual assembly code listing in the Appendix, as many specific references to routine variables and their purpose will be made.

The system software can be divided into two functional areas: system and graphics levels. The system level software contains such functions as: 1) Microcomputer monitor/debugger, 2) Graphic hardware monitor, 3) Download programs, 4) Terminal interface, 5) Editor, 6) Assembler, and 7) File system. The graphics level software is composed of: 1) Lexical Scanning, 2) Command interpretation, 3) Display list execution, 4) Error handling, and 5) Drivers for the video display user interface hardware (ie. VDP, joystick/trackball, etc.). A description of the concepts and design of the software for these two functional areas will follow.

3.2. System Concepts

One of the design goals of the graphics system was to make it a stand alone system as much as possible. Many low-cost graphics systems function in a mode where they must be connected to a host computer system on which the user develops and enters a display file which is then downloaded and executed by the graphics machine. This works well, but incurs the overhead of a host machine and limited interactive capabilities. The premise behind this project was to identify what a user would need in a stand alone system and include these capabilities in the design. Of the the system software functions described above, items one through four were implemented in this project and the remainder were conceptually specified.

3.2.1. Microcomputer Monitor/Debugger

The microcomputer monitor/debugger can be used for a number of applications. Probably the most prominent ones are debugging development software for the system and creating special assembly language routines to create and manipulate graphics objects. To expedite the development process, an existing monitor source was used and modified to work on this machine. A short synopsis of the monitor commands is presented in Figure 3-1, with a more detailed description of the monitor, its usage and operation in the Appendix. Briefly, the monitor can examine and modify microcomputer memory/register locations, single-step through an assembly language program, and set, cancel, and resume

Command	Explanation
aaaa0	Open for examination the memory location at address $aaaa. The next prompt will show address $aaaa and it's contents.
LF	Open for examination the memory location whose address is one less than the currently open location. Note: LF stands for the line feed key on the terminal.
CR	Open for examination the memory location whose address is one more than the currently open location. Note: CR stands for the carriage return key on the terminal.
ddM	Place the data value $dd into the currently open memory location.
G	Begin execution of a program at the currently open memory location.
<sp>A	Open the memory location in page zero where the copy of the 6502 A register is stored. This value may be modified at any time using the M command.
<sp>X	Open the memory location in page zero where the copy of the 6502 X register is stored. This value may be modified at any time using the M command.
<sp>Y	Open the memory location in page zero where the copy of the 6502 Y register is stored. This value may be modified at any time using the M command.
<sp>S	Open the memory location in page zero where the copy of the 6502 stack pointer register is stored. This value may be modified using the M command.
<sp>P	Open the memory location in page zero where the copy of the 6502 processor status register is stored. This value may changed at any time using the M command.
b<sp>B aaaa	Activate breakpoint b at address $aaaa.

Microcomputer Monitor/Debugger Commands

Figure 3-1

from up to seven breakpoints.

3.2.2. Microcomputer Monitor Graphic Extention

As depicted earlier in Figures 1-4 and 2-6, the VDP contains 16K of video RAM that is not a part of the microprocessor's address space. The modification of video RAM requires interaction with the VDP. It is often necessary to examine/modify video RAM when debugging various video display programs. An extended command set was created to enable a user of the microprocessor's monitor to do this. A list of these commands appears in Figure 3-2, with a detailed description of their usage and operation in the Appendix. Basically, these extensions allow a user to read and to modify video RAM locations.

3.2.3. Download Program

To take advantage of the use of a host machine, a download program was written and used extensively during the software development. The program accepts ASCII information at 9600 baud via one of the two serial ports on the microcomputer board. During software development, assembled 6502 code was downloaded into the graphics system to be run and debugged. The download program accepts information in ASCII hex-space format, and stores the information in RAM according to the assembler start address directives. The program checks for format errors, such as an invalid character, computes and verifies the checksum, and prompts the user when done. The appropriate error messages are output in

Command	Explanation
aaaaO	Open for examination the video display memory location at address $aaaa. The next prompt will show address $aaaa and it's contents.
LF	Open for examination the memory location whose address is one less than the currently open location. Note: LF stands for the line feed key on the terminal.
CR	Open for examination the memory location whose address is one more than the currently open location. Note: CR stands for the carriage return key on the terminal.
ddM	Place the data value $dd into the currently open video memory location.
<sp>S	Displays the contents of the Video Display Processor's (VDP) status register (R8).

Video Display Processor Monitor Commands

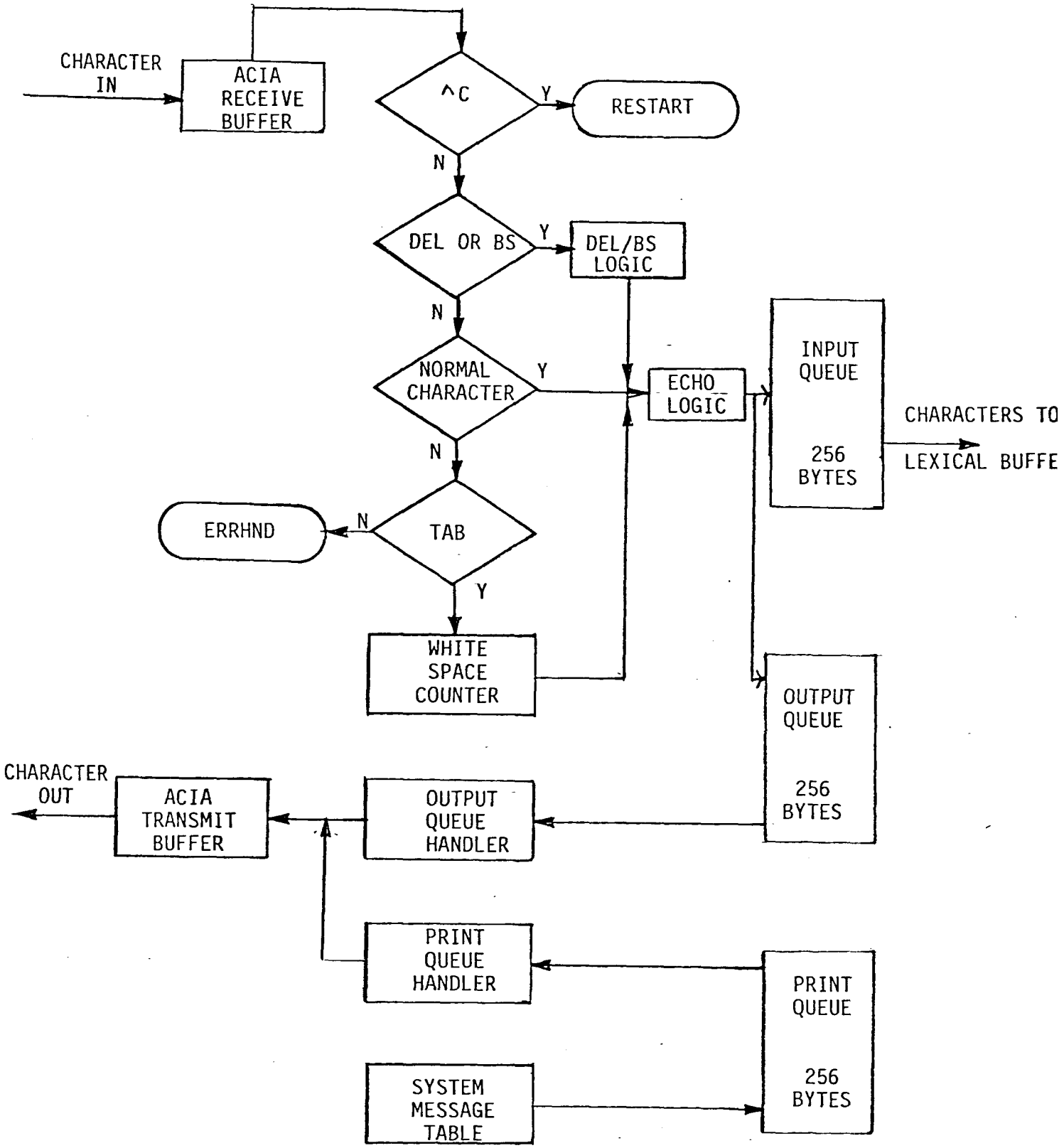
Figure 3-2

the event of an error. A description of the ASCII hex-space format, (interfacing to a Data I/O system 19 PROM programmer), and the operating system (Unix V7 UCB) it functioned under are provided in the Appendix.

3.2.4. Terminal Interfacing

During the initial development of the terminal interface software, it became apparent that implementation of a "dumb" program would result in an unacceptable user interface. (For example, a dumb terminal interface would respond to a RUBOUT character by displaying a "/" followed by the character being deleted instead of backspacing over the character and displaying a space.) The interface device was originally intended to be a terminal, through which the programmer could enter graphics primitive commands, which, in turn would either be displayed on the monitor or would be incorporated into a file. As indicated in Section 2.3, the addition of a joystick/trackball, and redefinition of the "h", "j", "k", and "l" keys for cursor movement would be desirable or additional features. In order to be able to build upon the software, a modular and expandable approach to the terminal handler was employed. An overview of the interface hardware/software is shown in Figures 3-3 and 3-4.

Figure 3-3 depicts the general flow of information to and from the terminal and into the input and output queues. Basically, the incoming character is received in the ACIA's receiver buffer. This triggers an interrupt to the processor, which determines through



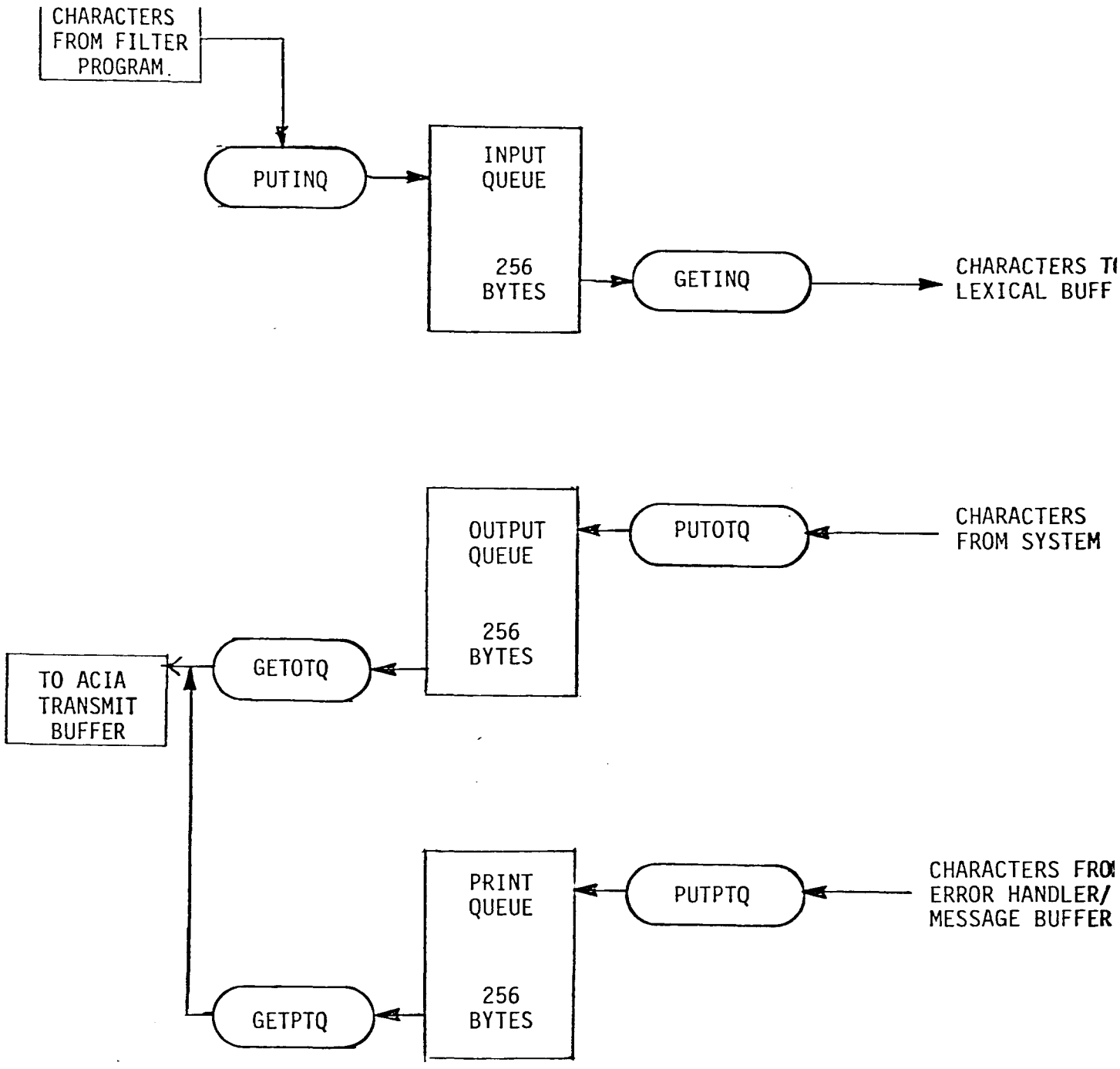
CHARACTER HANDLING OVERVIEW

Figure 3-3

the use of a prioritized check of interrupting devices, that the ACIA needs attention. The character is then read from the ACIA receiver buffer and is passed to a "filter" routine which performs several checks. First, if the character is a ^C, the terminal monitor is restarted. Next if the character has a special meaning, such as delete or backspace, appropriate actions are performed on the contents of the input and output queues, and the queue pointers are adjusted accordingly. Otherwise, if the character is a "normal" printing character, it is inserted into the 256 byte circular input queue (INQUE), and subroutine ECHO is then called to echo the character by inserting it into the output or echo (OUTQ) queue. The transmission of characters is also performed on an interrupt basis. The interpretation of characters stored in INQUE is discussed in the Section 4, "Language Design", of this document.

3.2.4.1. Queues and Queues Handling

As can be seen in Figure 3-4, the system has three queues. The input queue is for incoming characters, the output queue is for out going (echoed) characters, and the print queue is for displaying system messages. Because the system has extensive error reporting capability, the possibility of intermixed messages exists. With the two queues, this problem is reduced. Currently, the error messages are short (i.e. numeric), so they are handled via the output queue. Future enhancements would call for the error messages to be stored in ROM as English-style messages, with the error numbers acting as indices to the appropriate messages. The messages would then be transferred in a block move to the print



SYSTEM QUEUES

Figure 3-4

queue, and output accordingly. Since both queues transmit to the same ACIA, it is possible that messages and echoed characters could become interleaved. The approach taken here is that it is important to let the user know what is wrong, rather than correctly echo the line s/he is currently working on. If the system generates an error message, once the error message(s) is loaded into the print queue, that queue "hogs" the path to the ACIA. A return to normal operation occurs when error message transmission is completed. It is believed that the queue sizes are large enough to accommodate any terminal I/O condition that may occur. Since each queue is 256 bytes, which is approximately three 80 character lines, this would convey enough information. The error message look-up table has not been designed but the queue contention software has been incorporated into this version of the software.

Another design technique of interest involves the interactions between the queues. Since there are two queues, one of four conditions may exist:

	Input Queue	Output Queue
1.	Not full	Not full
2.	Not full	Full
3.	Full	Not full
4.	Full	Full

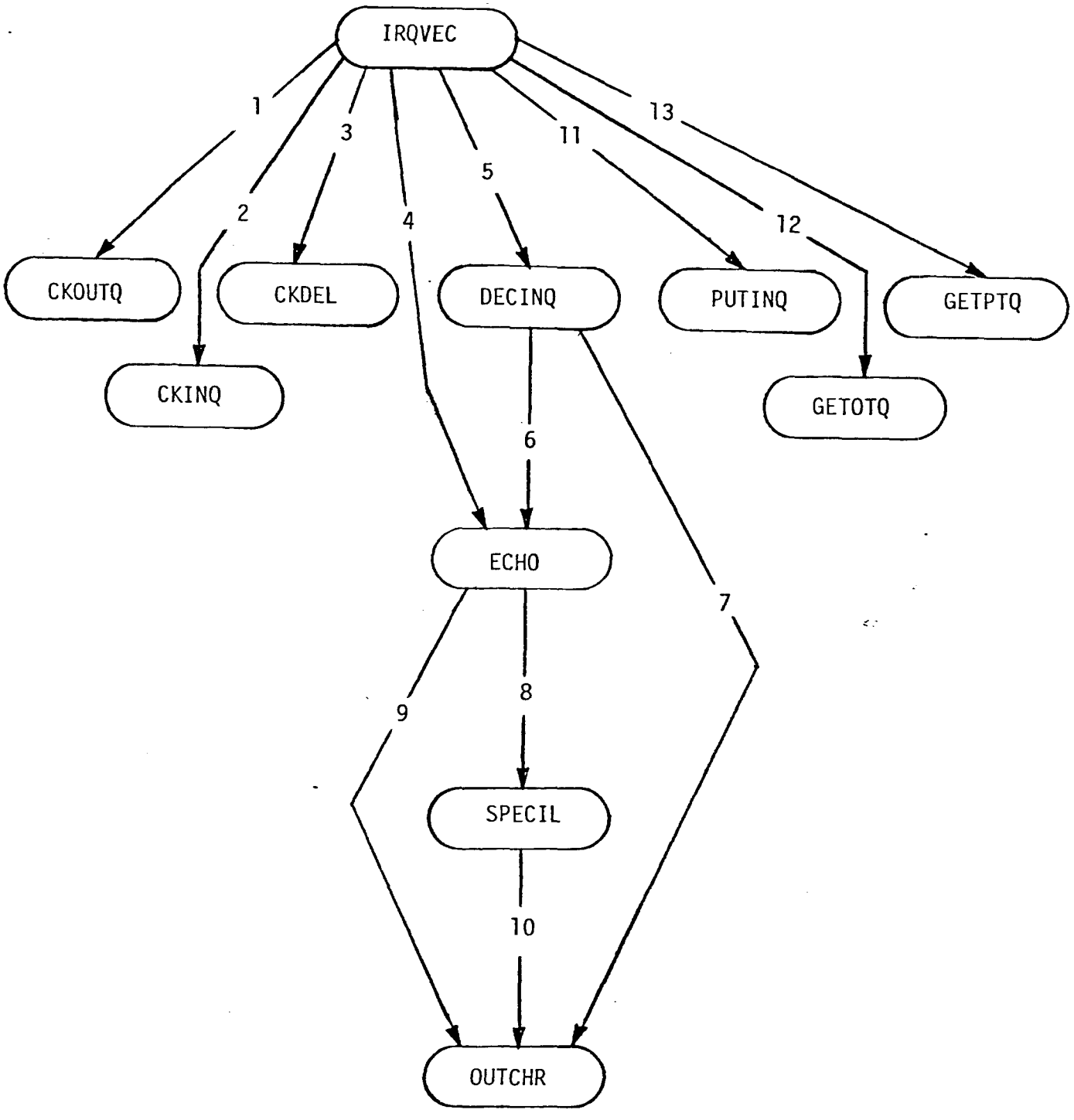
If an ACIA receiver interrupt occurs and condition 1 exists, the character is echoed in the normal way. If either condition 2 or 4 exists, the character is ignored. If case 3 exists, a "bell"

character is placed in the output queue to signify to the user that the input queue is full. Since the software must be able to echo two or more characters as a result of a single input character, as might occur due to the system's support of horizontal tabs, backspace/delete, and future editing features, the variables INQFRE and OTQFRE are checked to insure that enough space exists to echo the correct character string. If enough space does not exist, the command is ignored and a bell character is placed in the output queue. Under normal conditions, it is very unlikely that these error conditions will occur.

Another design feature incorporated into the system software that deals with the queues, is that of queue pointer images. A problem associated with interrupt driven systems, that makes queue handling more difficult, is the randomness of the input and output interrupts. It is conceivable that one software routine may be trying to put something into the queue while another is trying to take a character out. To alleviate the problem of the queue head and tail pointer variables both being accessed during a critical window, copies of the head and tail pointers are made at the beginning of a queue I/O sequence. The copies are manipulated during the queue operation and then the real parameters are updated at the end of the queue access routines.

3.2.4.2. Software Implementation

To illustrate the subroutine flow of the terminal handling software, the reader is referred to Figure 3-5. This figure,



TERMINAL HANDLER SUBROUTINE
FLOW DIAGRAM

Figure 3-5

in conjunction with the software listings, provides a detailed presentation of the terminal interface logic. The system is normally in a wait loop. All of the hardware devices are attached to the IRQ line of the processor as described in the Hardware Design section, and are initialized to act as interrupting devices. Suppose that an interrupt occurs and it is the terminal ACIA. Upon the acknowledgement of an interrupt, the interrupt service routine (IRQVEC) is entered. Preliminary checks are made of the higher priority devices such as the VDP and VIA. Interrupts from these devices are of primary importance because they deal with the time slicing of the display frame. This display technique is described in more detail in Section 2.1.3.3.1.

Subsequently the receiver interrupts are checked. If the terminal ACIA did not cause the interrupt, the interrupt is declared spurious and the interrupt service routine returns. Additional interrupt checking can be inserted in the IRQVEC routine as required by the addition of new devices. However, it is important to maintain relative and absolute priorities of the VDP and timer interrupt checking when additions are made. The status of the receiver buffer is then checked, and if it is full, provisions are made for handling this error condition. If the transmitter was the source of the interrupt, subroutines GETOTQ and/or GETPTQ are entered to empty the echo or print queues (path 12 or 13).

Incoming characters are handled in the following manner. First, the state of the output queue is checked in subroutine

CKOUTQ. If it is full, the character is ignored. Otherwise, a check is made, in subroutine CKDEL, for either a DELETE or BACKSPACE character and if this check is true, subroutine DECINQ "winds back" the input queue. If the character to be deleted is anything but a horizontal tab, the routine passes a backspace, space, backspace to subroutine OUTCHR, which places these three characters in the output queue and enables transmitter interrupts (path 1,2,3,5,7).

If the character to be deleted is a horizontal tab, the input queue is backed up to the last carriage return, and the entire current line is redisplayed minus the horizontal tab. This sequence of events is handled by the DECINQ, ECHO, and OUTCHR subroutines (path 1,2,3,5,6,8,10 or path 1,2,3,5,6,9 depending on what characters the line to be rewritten contains). Subroutine DECINQ "removes" the entire line from the input queue by backing up the tail pointer to the last carriage return. Then, the characters are output in the normal way through subroutines ECHO and OUTCHR. Subroutine ECHO handles normal printable characters between $20 and $7E. Subroutine SPECIL handles other special characters such as a bell, horizontal tab, and carriage return/line feed that may already be in the line to be re-displayed.

Handling of "normal" characters, ranging from a space ($20) to a tilda (~) ($7E), is illustrated in Figure (path 1,2,3,4,9,11). The character is received, the queue status is checked by subroutines CKOUTQ and CKINQ, and if events follow the queue logic outlined earlier, the character is echoed in subroutine ECHO,

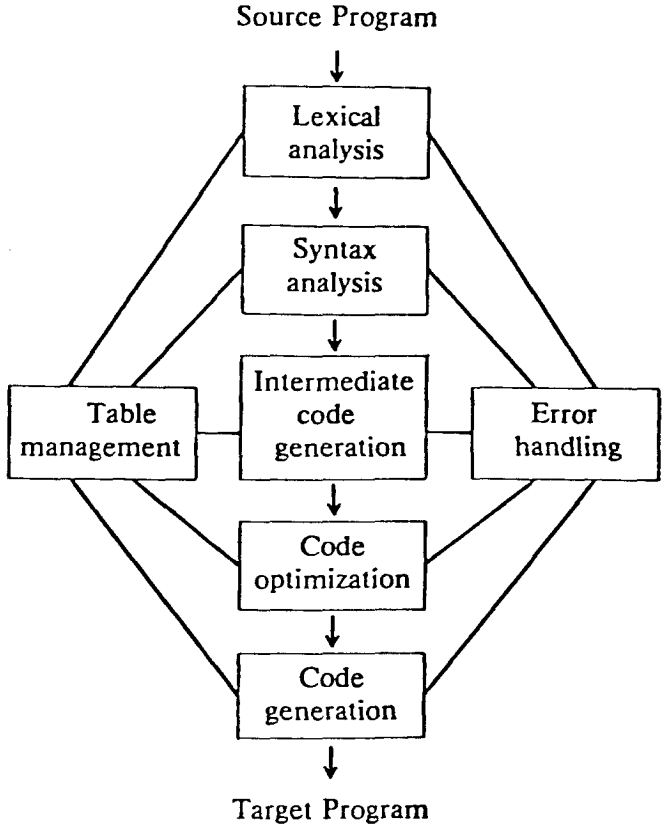
handled appropriately if it is determined to be unique by subroutine SPECIL, and output by subroutine OUTCHR (the special character handling is done in path 1,2,3,4,8,10,11).

The input, output and print queues are the connection between the raw character handling and the command interpretation. Subroutines CKPTQ, GETINQ, PUTOTQ and PUTPTQ are invoked by the lexical scanning layer of the software. Briefly, subroutine GETINQ retrieves a character from the input queue, subroutine PUTOTQ puts a character into the output queue, subroutine CKPTQ checks the status of the print queue for full or not full, and PUTPTQ places a character into the print queue. The interaction of these routines with the overall system will be described later.

3.2.5. Lexical Scanning and Parsing

3.2.5.1. Overview

The process of desiging a command interpreter is a subset of the process of designing a compiler. The block diagram of a compiler's components, or phases is shown in Figure 3-6. (A phase is a logically cohesive collection of programs the produce an output compatible with the next phase.) The design of the graphics interpreter embodied many of the phases shown in the figure, but the development of a complete graphics compiler was beyond the scope of this project. For the implementation of the animation language, a software design approach that incorporated many compiler phases was determined to be the best way to proceed. This methodology had many benefits, of which the most important ones



Phases of a Compiler

Figure 3-6

were a well defined program flow for the interpreter and a good foundation of subroutines that could be totally incorporated in the compiler development phase.

The design of the interpreter software was subdivided to include the following functions: lexical analysis, syntax analysis, semantic analysis, table management, and error handling. Although the scope of this project did not warrant the incorporation of all of the remaining phases of a compiler design, there is reason to discuss these areas. The incorporation of various data structures, logical assignments, and the scope of variables must be considered for future versions of the language.

3.2.5.2. Tokens

The lexical analyzer is the interface between the graphics source program and the interpreted displayed action. The lexical analyzer reads the source program one character at a time and determines when a token, or a sequence of characters that can be treated as a single logical entity, is found. There are two kinds of tokens: specific words such as VEC or IF, and classes of strings such as constants and labels. The difference is that strings such as VEC and IF can not have a numeric value. Therefore, a token can be thought of as having two attributes: a token type, and (a possibly NIL) token value. In the design of this graphics language, the following types of tokens were allowed:

Token Class	Token Type
Keywords	01
Identifiers	02
Constants	03
Strings	04
Operators	05
Punctuation	06
other	07

The value of the token type associated with each token class is used in the actual assembly code to encode what class of token was desired or encountered. For example, if a keyword is desired, a $01 is passed to the appropriate routine. In the current version of the system, strings, operators, and punctuation are not used in the language. The subroutines that deal with these tokens were written to permit the language to make use of them at a later date with minimal difficulty.

3.2.5.3. Transition Diagrams

A relatively simple but robust way of determining token classes in the design of a lexical analyzer is by the use of a transition diagram. In a transition diagram, the boxes of a flowchart are drawn as circles and are called states. The states are connected by arrows called edges. The labels on the various edges leaving a state indicate the input characters that can appear after that state. These diagrams are a relatively compact way of

indicating what types of characters can comprise various tokens.

Figure 3-7 shows a transition diagram for an identifier, defined to be a letter followed by any number of letters or digits. The starting state of the transition diagram is state zero, the edge of which indicates that the first input character must be a letter. If this is the case, we enter state one and look at the next input character. If that is a letter or a digit, we re-enter state one and look at the input character after that. This process continues until a specified delimiter is reached. According to the language syntax and token class, this could be a punctuation mark such as a comma or semicolon, or a space. Upon reaching the delimiter, state two is entered. State two could be a return to a driver program set up in either a subroutine or coroutine form of execution.

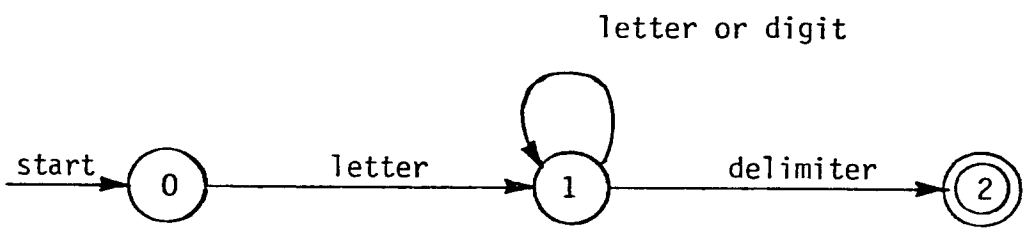
3.2.5.4. Command Lines

Graphic languages are generally composed of command lines. The primitives that enable the user to draw a circle with relative ease can be of the form:

keyword<0:>attribute<0:>attribute<0:>....attribute<0:>

NOTE: <0:> denotes a delimiter

where the keyword can be, for example, CIRCLE, and the attributes can be the radius, X,Y position of the center, and perhaps the



Transition diagram for identifier

Figure 3-7

color. Keywords, such as CIRCLE, that cannot be used as a label or identifier, are termed reserved words. This strategy is used in the development of the language so that once this word is identified in the system symbol table, the code for handling that keyword can direct the gathering of token classes (parsing) according to the syntax and grammer of the language.

3.2.5.5. Regular Expressions

The type of lexical analysis and the use of reserved words can be related to other concepts and design philosophies of languages, specifically, regular expressions and finite automata. Consider the above form of the command line to consist of a string of characters, represented in a binary form, which can be considered a binary alphabet consisting of ones and zeros. These strings can have properties of sets such as concatenation, union, and concatenation of closure. The way in which these operations are put together in a string defines the language. In the example of the identifier transition diagram given earlier, a regular expression notation would look like:

identifier = letter(letter|digit)*

The vertical bar means "or," that is, union, the parentheses are used to group subexpressions, and the asterisk is the closure operator, meaning "zero or more instances." It is from a description of this nature that a program can be constructed to "look for" various tokens. The grammar of a language is formed by a set of production rules for generating the words of a language.

A regular expression is basically the formal description of a language acceptable by a finite automata. It tells how the language is built up from atomic languages by the use of regular operations (i.e. concatenation, union, and concatenation of closure).

The tokens discussed above are described by the following regular expressions.

reserved word = barc|cir|clr|curpos|cursf|curso|dspst|earc|intg
|marc|move|paint|revid|setbdc|setbgc|setfc|vec

identifier = letter(letter|digit)*

constant = (digit)*

string = (letter|digit)*

relative operator = <|<=|>|>=|<>|>|>=

punctuation = ,|;|:|}|{

operator = +|-|*|/|**|

These expressions were used to develop the lexical analyzer. It should be pointed out that programs exist to build lexical analyzers. One such program is LEX. Basically, LEX takes its input in the form of regular expressions and produces a lexical analysis program. Although LEX is a common tool on UNIX operating systems, the output of LEX is in 'C' and the author did not have access to a 'C' to 6502 cross assembler. However, essentially the same

methodology that is used by LEX was used to generate the assembly language code for the interpreter.

3.2.5.6. Context-Free Grammar

As pointed out above, regular expressions are capable of describing the syntax of tokens. Any syntactic construct that can be described by a regular expression can also be described by a context-free grammar. A context-free language is a language that may be described syntactically by direct unconditional substitution and concatenation of symbols. Moreover, the substitution of a phrase for a symbol is independent of the symbol. Both the language and the grammar are then called called context-free. There were several reasons why regular expressions were used instead of a context-free grammar. First, the lexical rules are usually simpler and there was no need for a notion as powerful as context-free grammar. The fact that the graphics language was more of a command language rather than a programming language supported this decision. Second, it is easier to construct efficient recognizers from regular expressions than from context-free grammars. Third, separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler/interpreter into manageable-sized components.

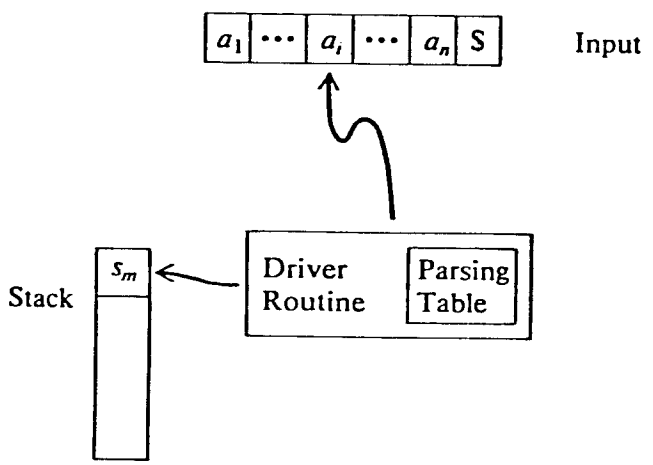
There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. Regular expressions are most useful for describing the structure of lexical

constructs such as identifiers, constants, and keywords, etc,. Context-free grammars, on the other hand, are most useful in describing nested structures such a balanced parentheses and matching "begin-ends". These nested structures cannot be described by regular expressions. This does not imply that multiple loops cannot be implemented. In the case of "begin-ends", variable scope can change, which would alter the contents of these variables. This in turn changes the construction of the lexical analyzer and makes it more difficult to implement.

3.2.5.7. LR Parsing

In the design of the graphics interpreter, the model of a LR parser was used. A LR parser scans the input line from left to right and constructs a rightmost derivation in reverse. The LR parsing mechanism was chosen for two reasons. First, LR parsing is more general than operator precedence or any other common shift-reduce techniques. Second, LR parsers can detect syntactic errors as soon as it is possible to do so on a left-to-right scan of the input line. This allowed the design of the parser to detect syntactic and semantic errors at a specific level during the interpretation of a command line. The implementation details will be more throughly discussed in Section 3.2.6 , Assembly Language Implementation.

The LR parser is composed of the input string, the output stack, the driver routine and the parse table (Figure 3-8). The input is read from left-to-right one symbol at a time. These



LR Parser Block Diagram

Figure 3-8

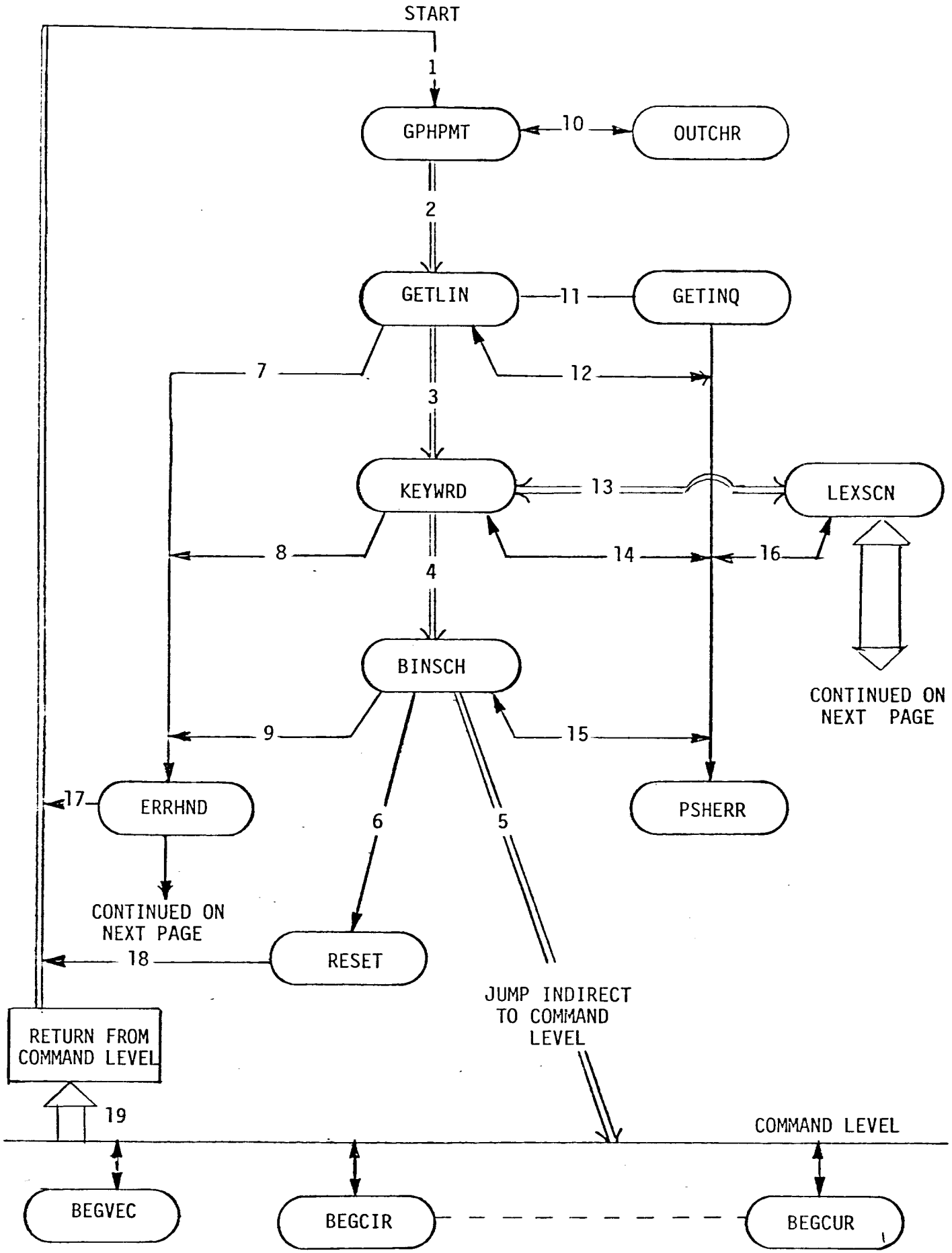
symbols are then placed in a stack. In this implementation of the interpreter, the symbols are placed in a parameter area of variables. The ordering of the symbol "fetches" determines the order inwhich they are placed in the parameter area.

Upon successful scanning of the input line and building of the parameter area, the variables are checked for correct numeric range and interaction with other parameters. Successful completion of this level results in advancing to the specific section of code that executes the specified command. The implementation details of this mechanism are discussed below.

3.2.6. Assembly Language Implementation

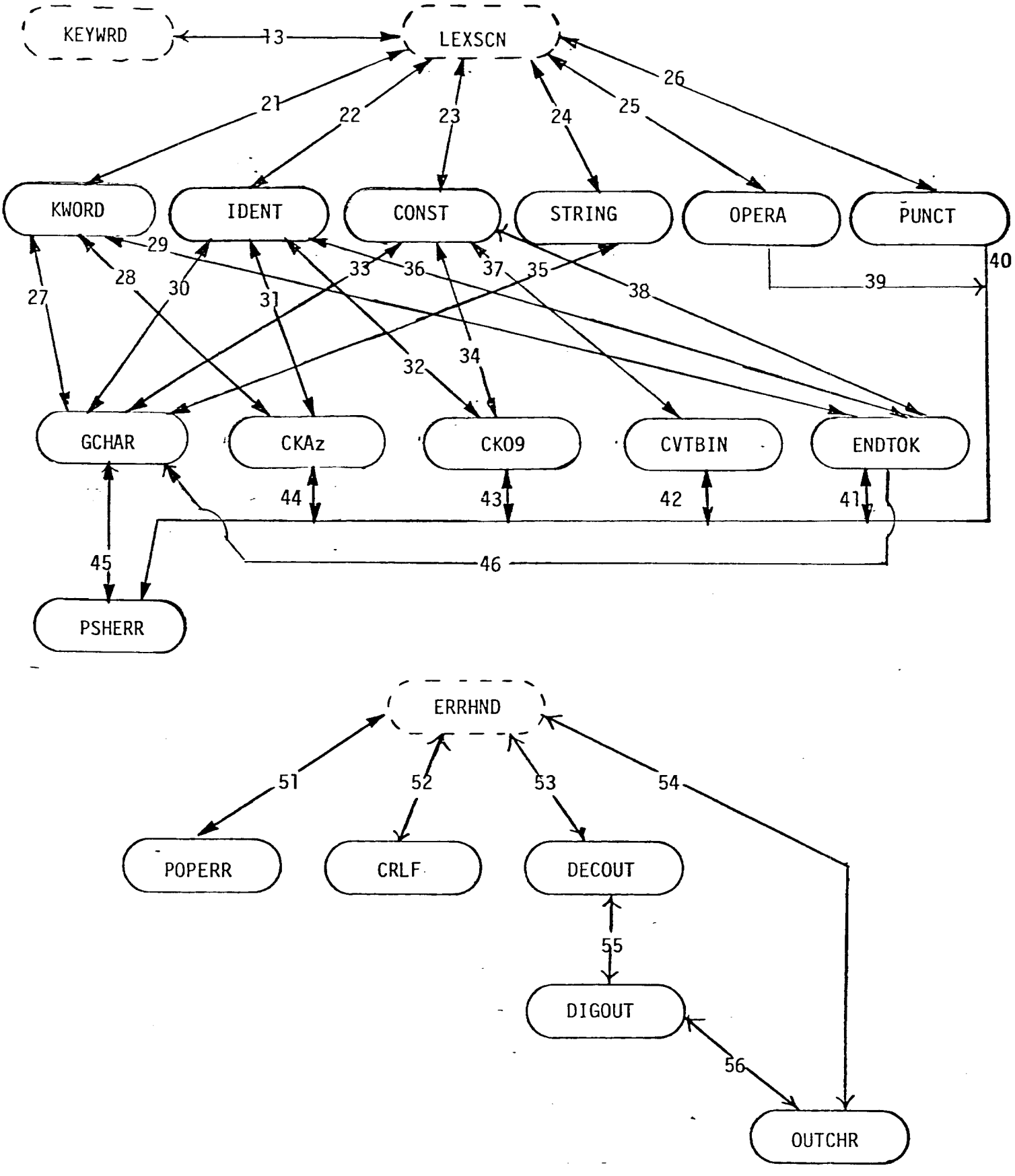
3.2.6.1. Command Line Retrevial

With the methodology for the design of the lexical analyzer discussed, the actual assembly language routine functions and interaction can be described. The reader is referred to Figure 3-9 for the lexical subroutine logic flow. For purposes of discussion, we shall assume that the interrupt driven terminal interface has been putting characters into the input queue. When a character is inserted in the input queue, it is checked to see if it is a carriage return. If it is, a variable called CRGCTR is incremented and the interrupt service routine exits. The variable CRGCTR serves as a line counter and it is used as a flag to start the lexical scanning process. When the processor is idle, it is executing a wait loop checking to see if CRGCTR is greater than zero. When this condition occurs, a complete line that is



LEXICAL ANALYZER SUBROUTINE
FLOW DIAGRAM

FIGURE 3-9



LEXICAL ANALYZER SUBROUTINE
FLOW DIAGRAM

FIGURE 3-9 CONTINUED

terminated by a carriage return is copied, by the subroutine GETLIN, into a lexical scanning buffer called LEXBUF. The subroutine GETLIN also decrements the variable CRGCTR and checks for lexical buffer overflows. Subroutine GETLIN in turn, calls subroutine GETINQ which retrieves a character from the input queue and adjusts the queue pointers accordingly (path 2,11).

If the processor were interpreting a graphics primitive command when CRGCTR was incremented, nothing would happen. The processor would simply finish executing the current command line, return to the wait loop, find CRGCTR not equal to zero, transfer the next line into the lexical buffer, and proceed as indicated above. The system is quadruply buffered: There are two buffers from the hardware standpoint in the ACIA, and two in software via the input queue and lexical buffer. This provides virtually no perceivable lag in system response, even with the fastest typer, and performs well when a display file composed of a number of graphics primitives is downloaded via the serial download port.

3.2.6.2. Keyword Recognition

With a string of characters now in the lexical buffer, subroutine KEYWRD is called (path 3). Due to the design of the language the first type of token that the lexical scanner will search for is a reserved word. To indicate this information to the lexical scanner subroutine (LEXSCN), subroutine KEYWRD sets the variable TOKCLS to $01. This variable is global to the rest of the lexical analyzer and its function is to indicate which type of

token is required. Subroutine LEXSCN (path 13) is then called to evaluate the variable TOKCLS to determine which token type to look for (keywords through punctuation). Since this is the first pass through this line of characters in LEXBUF and the variable TOKCLS contains $01, LEXSCN will call subroutine KWORD (path 21).

The function of KWORD is to scan the lexical buffer from its beginning, transferring each character to KWBUF, the keyword buffer, until a delimiter is found. It should be pointed out that these types of routines were derived from the transition diagrams and regular expression notation mentioned earlier. During the scan of LEXBUF and the transfer of characters into the keyword buffer, subroutine KWORD calls a number of support subroutines to assist in executing the logic of the transition diagrams. These subroutines are: GCHAR, to get a character from the lexical buffer; CKAZ, to check if the characters are between A-Z or a-z; ENDTOK, to determine if a delimiter was encountered and also if it was valid, and PSHERR, to accept an error code generated by one of many subroutines, push it on the error stack and set the error flag (the carry bit) to inform higher level routines that an error of some sort has occurred (path 21,27,28,29,41,43,44,45). It is up to the higher level routines to determine if the error should be considered a fatal or not.

Once the entire reserved word is transferred to the keyword buffer, a reserved or keyword table is searched. If no match is found, an error occurs and the character pointer in the lexical buffer is reset to zero and readied for the next line (path

4,15,9,17). At this point, the lexical buffer could be printed out with the appropriate error messages on the screen. This was not done in this version due to project time requirements.

The keyword table is searched by the subroutine BINSCH (path 4), which performs a binary search of the table to determine if the string of characters in the keyword buffer is really a valid reserved or keyword.

3.2.6.3. Parsing a Command String

Once the existence of a keyword has been verified, the driver program of the parser calls the appropriate token gathering programs to retrieve the tokens required for the keyword and to store the tokens in the correct places. Once all the tokens have been successfully gathered, the execution of the command can begin.

Perhaps the best way to explain the implementation of this mechanism is by following an example. Suppose the command for the generation of a line from point a to point b was issued. According to the definition of the language, the general format of a command line is:

vec M,C,X1,Y1

where M is the coordinate mode specification, either relative or absolute, C is the color of the line, X1 and Y1 define the end point. It is assumed that the beginning point of the vector is the current cursor position. For this discussion, it is assumed that the keyword "vec" has been found and the address of the driver program has been loaded (Figure 3-10a). The processor now jumps to

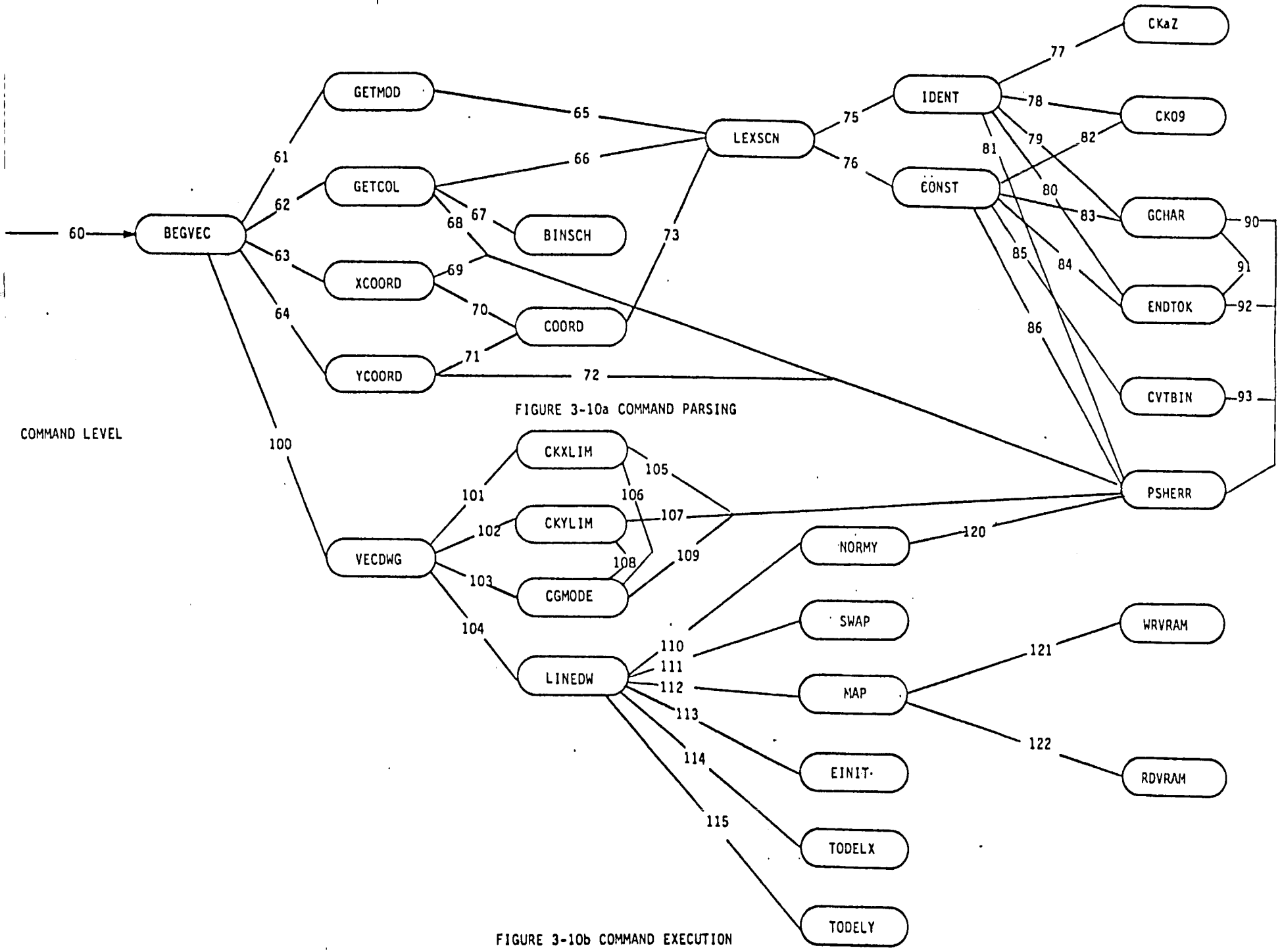


FIGURE 3-10a COMMAND PARSING

FIGURE 3-10b COMMAND EXECUTION

the beginning of that program (path 60). Subroutine BEGVEC is a short driver program that gets the tokens necessary for vector drawing. First, subroutine GETMOD is called to retrieve the next (second) token (path 61). Since the mode token has only alpha attributes, it is of token class "identifier". Subroutine GETMOD sets the variable TOKCLS to a $02, and calls subroutine LEXSCN, the lexical scanner (path 65). Subroutine LEXSCN checks the variable TOKCLS to see what type of token is to be retrieved, and then calls the appropriate subroutine. In this example, subroutine IDENT is called (path 75). Subroutine IDENT calls subroutine GETCHR to get characters from the lexical buffer and place them in the identifier token buffer IDTOK (path 79). In addition the lower level utility subroutines described earlier, CKAZ, ENDTOK, and PSHERR, are called as needed (path 77,78, 79,80,81,91,92). One additional subroutine is called in this pass that was not called before. This subroutine is CK09 and it checks to see if the value passed to it is between zero and nine. Once the token is found, program flow returns to subroutine GETMOD where, if there were no errors, the mode is determined to be either an "r" or an "a". If it was an "r", indicating relative mode, a $00 is loaded into the variable MODE. If it was an "a" indicating absolute, a $01 is put into the variable MODE. If neither one was found, an error is indicated.

If it is assumed that subroutine GETMOD found a valid mode, control is returned to subroutine BEGVEC which then calls the subroutine GETCOL to acquire the color token (path 62). Since the color token is an identifier, the variable TOKCLS is set to a $02 and subroutine LEXSCN is called. (Subroutine LEXSCN is essentially

byte two's complement representation could have been used, giving a range of +32,767/-32768. It was decided that encoding the sign into the least significant bit of a separate byte would have the advantage of faster checking and decision making in the numerical handling routines. Therefore the three byte convention was used.

Subroutine CONST calls almost the same support routines as subroutine IDENT does as well as calling subroutine CVTBIN that takes the ASCII representation of the number and converts it to a three byte binary value (path 82,83,84,85,86,90,91,92,93). Program control then returns back up through the levels until subroutine XCOORD is reached. The binary values of the X coordinate are transferred to the variables XSEN, XVALHI, and XVALLO for later processing.

At this point, the only information still needed to draw the line is the Y coordinate. Subroutine BEGVEC calls subroutine YCOORD for this value (path 64). The program path for this token is the same as described above for the X coordinate. The only difference is that the retrieved value is stored in YSEN, YVALHI, and YVALLO.

At this point in the interpretation of the command line, all the tokens have been obtained. It cannot be assumed however that additional information may not be included in the input line. The valid termination of a command line is a carriage return or a semicolon, indicating that the rest of the line contains a comment. The lexical analyzer continues to search the line for either of these conditions. In the case of a comment, the remainder of the

line is ignored and command processing continues. It should be pointed out that the lexical scanner ignores any form of white space, such as horizontal tabs and spaces, both before and after tokens and comments. In addition, the syntax for argument delimiters can be easily changed, in the assemble code, from commas to spaces, colons or most any other type of punctuation.

3.2.6.4. Command Interpretation

Command interpretation deals with the actual execution of the command once all of the arguments have been obtained by the lexical scanning process. This part of the software development basically replaces the code generator and optimizer portions of a compiler. The strategy used to develop the command execution software was to program the microprocessor and VDP in the most efficient manner to yield the desired result. For example, if a line is desired, a suitable algorithm is coded in assembly language to do this. The necessary data transfers to the VDP were also included in the algorithm. In the case of object generation, each point that is generated is passed to the VDP to illuminate the correct pixel in video memory. If functions such as changing background colors or reversing video are required, various table entries in the VDP would need to be changed, which generally resulted in changing control registers in the VDP or table entries in video RAM.

In the following discussion of the line drawing logic, the reader should refer to Figure 3-10b. At this point in the execution of the line draw command the lexical scanning process has

been completed and the tokens are stored in their appropriate place in memory. Program control has returned to subroutine BEGVEC, which in turn calls subroutine VECDWG (path 100). The function of subroutine VECDWG is to call the appropriate variable limit checking and clipping subroutines before the line is actually "drawn" (path 101,102,103,105,106,107,108,109). If it is necessary, subroutine CGMODE changes the specified user coordinates to internal machine coordinates. The specifed X and Y endpoints are checked againist the limits of the display area. If they fall outside of these limits, an error condition results. It is possible to simply clip the line and inform the operator with a warning message. Once the limits of all the variables are checked, the actual line drawing algorithm can be executed (path 104) provided no error conditions exist. If errors are present, control is returned up through the subroutines to the error handler and eventually back to the start of the graphics prompt program.

The details of the line drawing algorithm will be discussed in detail in the next section, however a short synopsis of the algorithm will be given. Subroutine LINDWG calls lower level subroutines to calculate the X and Y coordinates to be plotted between the endpoints at the line (path 110,111, 112,113,114,115,120). Once the coordinate pairs have been determined, subroutine MAP is called to map these into video RAM. Subroutine MAP invokes lower level utility subroutines to actually plot points into video RAM (path 121,122). When each point is finished being plotted, control is transferred back to subroutine LINDWG and if necessary, another point is computed and plotted.

The mechanism outline here is typical of the command interpretation process for each command of the graphics system. As the reader reviews the subroutine flow diagrams, it becomes apparent that some command interpretation actions are more rigorous then others. The point to be made here is that any number of commands may be added with little or no interaction problems for two reasons. First, the hierarchical design of the software allows command action to be inserted at a specific level and second, the majority of the subroutines required for the new command have been extensively tested.

One drawback in the use of interpreters is the constant addition of assembly code required to "execute" a new command. To reduce this problem, a number of support routines were written to make the system more modular and easily expandable. The more important support subroutines are: WRVRAM, which writes to video RAM indirectly through the VDP, RDVRAM, which reads the contents of video RAM at specified locations, WREG, which writes to the specified (0-6) VDP register; RREG, which reads the specified (0-7) VDP register; BLKMOV, which moves a block of data from one location to another; and MAP, which maps the generated image pixel onto the current display frame. These routines interface the point generation algorithms to the VDP. Collectively, subroutines RDVRAM, WRVRAM, and MAP could be referred to as a raster operation.

3.2.7. Major Algorithm Design

Currently, there are sixteen graphics processor commands

implemented. They range in complexity from turning the cursor on and off, to object painting. The more complicated algorithms will be described here along with reasons for their choice. The less complex algorithms can be analyzed from reading the assembly code and the comments contained therein.

3.2.7.1. Line Drawing Algorithms

Point plotting techniques are based on the use of a Cartesian coordinate system. These points are addressed by their X,Y coordinate pairs. In general graphics displays have three formats for coordinate systems. They are:

a) 0,0 is in the upper left hand corner, X increases from left to right, Y increases from top left to bottom left.

b) 0,0 is in the lower left hand corner, X increases from left to right, Y increases from lower left to top left.

c) 0,0 is in the center of the screen, with the X and Y axis displayed as in a regular 4-quadrant system.

The second format was chosen because it is generally acknowledged that people working with any sort of graph are more used to working in the first quadrant rather than the fourth quadrant or in all four quadrants. Software hooks are available to support either of the other two formats.

Lines generated should have the following characteristics.

1. They should appear straight.

2. They should terminate accurately.

3. They should have constant density.

4. The density should be independent of length and angle.

5. They should be drawn rapidly.

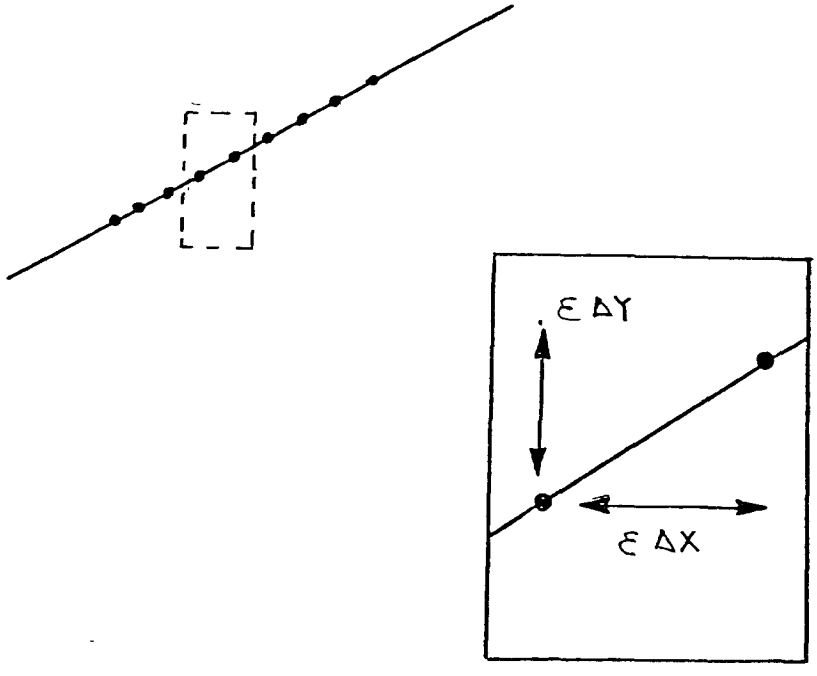
There are two basic line drawing techniques. One method has hardware generate the X,Y coordinate pairs and the other has a processor generate the coordinate pairs via a software algorithm. Since the use of the VDP chip precluded the design of the hardware generator, various software methods were considered. In many cases, hardware generators implement one of the software algorithms to be discussed.

3.2.7.1.1. Symmetrical DDA

Three line generation algorithms that were given the most consideration were the symmetrical digital difference analyzer (DDA), the simple DDA, and Bresenham's algorithm. The symmetrical DDA generates lines from their differential equations. For a straight line the equation is:

$$dy/dx = \Delta y/\Delta x$$

The DDA works on the principle that X and Y are simultaneously incremented by small steps proportional to the first derivative of X and Y (Figure 3-11). In the case of a straight line, the first



Ideal incremental method for straight line generators

Figure 3-11

derivative is constant and proportional to ΔX and ΔY. Thus, in the ideal case of an infinite precision display a line could be generated by incrementing X and Y by $e\Delta X$ and $e\Delta Y$, where e is some small quantity. However, in the real world of limited precision displays only addressable points must be generated, i.e. integer values.

The appearance of lines generated by the DDA depends on the value chosen for e. In the case of the symmetrical DDA, e is chosen to be equal to 2 exp -n, where

$$2^{n-1} \leq \max(|\Delta X|, |\Delta Y|) < 2^{n}$$

In fact e is the reciprocal of the DDA's line-length estimate, in this case, 2 exp n.

The symmetrical DDA generates accurate lines, since the displacement of a displayed dot from the true line is never greater than one-half a screen unit. Logically the symmetrical DDA is rather simple; the use of a negative power of two for e means that the increment of values can be determined by shifting the ΔX and ΔY values rather than by division. Each step in the line is computed with only two additions.

3.2.7.1.2. The Simple DDA

For the simple DDA, a line-length estimate equal to the larger of the magnitudes of ΔX and ΔY is chosen so that either $e\Delta X$ or $e\Delta Y$ is of unit magnitude. This allows one of the DDA's adders to become a counter. The simple DDA therefore generates unit steps in

the direction of greatest motion. A Pascal implementation is shown below.

```
procedure DDA(x1,y1,x2,y2:integer);
     var length,i:integer; x,y,xinc,yinc:real;
begin
     length:= abs(x2-x1);
     if abs(y2-y1) > length then length:=abs(y2-y1);
     xinc:= (x2-x1)/length;
     yinc:= (y2-y1)/length;
     x:= x1+0.5;
     y:= y1+0.5;
     for i:= 1 to length do
          begin
          Plot (trunc(x),trunc(y));
          x:= x+xinc;
          y:= y+yinc;
          end
end;
```

The simple DDA is as accurate as its symmetrical counterpart but generates a different sequence of dots because of its different methods of estimating line length. Logically it is simpler, though it performs an initial division to determine the increment value. The simple DDA is a good choice for a software line generator, but the division logic makes it less suited to hardware implementation. In so far as assembly language programming is concerned, the speed can be significantly increased by keeping the computations limited

to adding and subtracting with multiplication and division being limited to powers of two to accommodate the use of left and right shifts.

3.2.7.1.3. Bresenham's Algorithm

Bresenham's algorithm is an interesting variation of the basic DDA. Like the simple DDA, it is designed so that each iteration changes one of the coordinate values by +/- 1. The other coordinate may or may not change, depending on the value of an error term maintained by the algorithm. This error term records the distance measured perpendicular to the axis of greatest movement, between the exact path of the line and the actual dots generated. The following is a Pascal version of the algorithm. It is assumed that the line to be generated is between 0 to 45 degrees in the first quadrant of a four quadrant coordinate system.

```
deltay = y2 - y1;
deltax = x2 -x1;
e:= (2*deltay)-deltax;
for i:= 1 to deltax do begin
     Plot(x,y);
     if e > 0 then begin
          y:= y + 1;
          e:= e + (2*deltay) - (2*deltax);
          end
          else e:= e + (2*deltay);
     x:= x + 1;
```

```
end;
```

This algorithm fulfills all of the desirable characteristics specified above for line drawing routines. The mathematics used are integer addition and subtraction and any multiplication is by two only. There is no division. Even though the mathematical operations will have to be done in double precision, ie 16 bits, the processor is still fast at this level. The line generated is accurate in terms of beginning and ending points. No estimate of the line length is required as in the simple and symmetrical DDA's, and no point can possibly be plotted twice. This is also of concern, given the mapping algorithms which is used. (This mapping algorithm is described in Section 2.1.3.6). As pointed out earlier, this version is only for 0 to 45 degrees of arc. However, due to symmetry, each 45 degree sweep can be folded about the origin to plot lines with any slope and length. This is precisely what is done in the algorithm used by the graphics processor. A high level pseudo code example of the algorithm designed is as follows.

```
deltax = x2 - x1;
deltay = y2 - y1;
if deltay < 0 then x1,y1 is swapped with x2,y2;
     recompute deltax, deltay;
if deltax < 0 then subflag = 1;
     deltax = abs(deltax);
else subflag = 0;
if deltay => deltax then swapxyflag = 1;
```

```
     deltax is swapped with deltay;
else swapxyflag = 0;

for i= y1,y2 do;

     e:= 2*deltay-deltax;
     for i:= 1 to deltax do begin
          Plot(x,y);
          if e > 0 then begin
               y:= y + 1;
               e:= e + (2*deltay - 2*deltax);
               end
               else e:= e + 2*deltay;
          x:= x + 1;

If swapxyflag = 1, then swap x and y;
If subflag = 1, then xplot = x1 - x;
     else xplot = x1 + x;
yplot = y1 + y;
plot(xplot,yplot);
end;
```

Basically the algorithm takes the desired line and rotates it until it gets into the proper orientation for Bresenham's algorithm to work correctly. That is, the line is rotated to the 0 to 45 degree area in the coordinate plane. The points are calculated and then rotated back to their desired orientation. This reversing procedure is controlled by the variables SWAPXYFLAG and SUBFLAG.

3.2.7.2. Circle Generation

3.2.7.2.1. The Circle DDA

Circle generation can be accomplished by using a DDA. This technique is an extension of the line drawing DDA. Since a circle can be described as a differential equation, a corresponding DDA can be derived. It is possible to construct a DDA that draws an exact circle, using the equations

$$X(n+1) = Xn \cos\theta + Yn \sin\theta$$

$$Y(n+1) = Yn \cos\theta - Xn \sin\theta$$

Since θ is generally small, values of cos θ and sin θ are relatively easy to compute and are then constant for any particular circle radius. These values, once derived, could be stored in a lookup table, excluding the need for a sine and cosine algorithm. These equations can also be used if multiplications on the target system can be performed inexpensively. However, referring back to the rules of good drawing techniques, it was pointed out that objects must be drawn relatively fast. Most eight-bit microprocessors do not have an integer, multi-precision multiply instruction. Doing the computations in software can be efficient only if the multiplication routine is not called repeatedly, and since this is not the case in circle generation, another algorithm is needed.

3.2.7.2.2. Bresenham's Circle Algorithm

Bresenham's line drawing algorithm can be extended for the case of circular arcs. The Pascal equivalent of the assembly language used to draw the circle follows:

```
begin
x = 0;
y = 0;
cdelta = 2 - 2 * radius;
repeat

delta = 2 * cdelta + 2y - 1;
deltap = 2 * cdelta - 2y - 1;
if cdelta <> 0 then
     begin
        if ((cdelta < 0) and (delta > 0)) or ((cdelta > 0)
        and (delta <= 0)) then
             begin
                if (cdelta < 0) and (delta <= 0) then
                  x = x;
                  y = y + 1;
                  cdelta = cdelta - 2y + 1;
                else
                  x = x + 1;
                  y = y;
                  cdelta = cdelta + 2x + 1;
             end;
         else
```

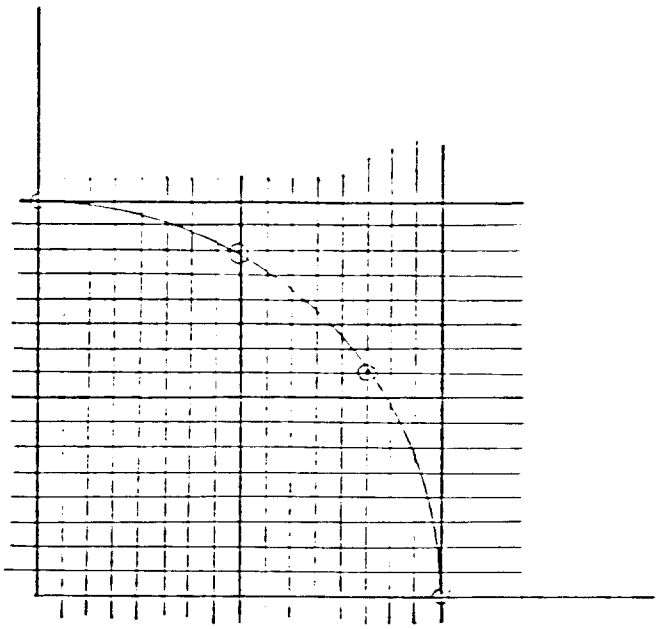
```
            x = x + 1;
            y = y - 1;
            cdelta = cdelta + 2x - 2y + 1;
        end;
else
  x = x + 1;
  y = y - 1;
until y = 0;
  cdelta = cdelta + 2x - 2y + 1;
end;
end;
```

This algorithm draws a quarter of a circle starting at the (0,radius) coordinates and generates points to the (radius-1,0) point. Generation of a complete circle is then just a matter of transforming the generated X,Y pair in the first quadrant to the mirror image of the points in the other three quadrants. Since the circle is always drawn with the radius at the 0,0 point in the coordinate system, the current cursor X and Y coordinates must be added to the generated X,Y pair to have them displayed correctly. This algorithm was used because the multiplication required was one of double precision, single position, left shifting, and double precision addition and subtraction. The algorithm uses only integers, which increases its calculation speed.

The algorithm uses the basic relationship of the points of a circle with respect to its radius. This relationship is defined as:

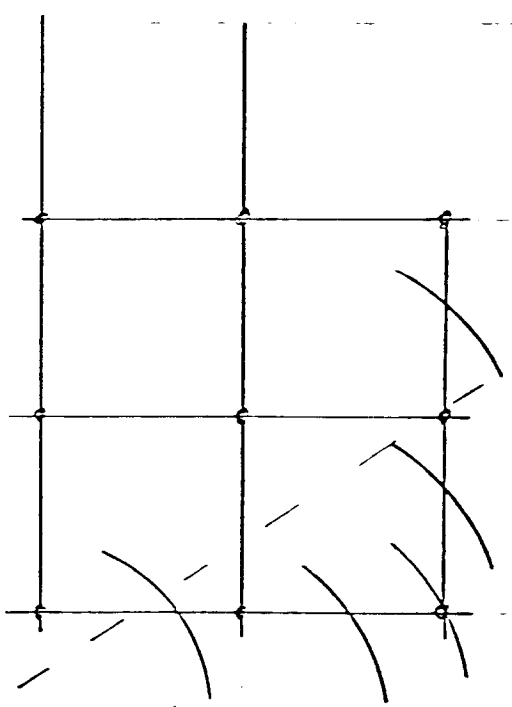
$$X^2 + Y^2 = R^2$$

where the X,Y pair lies on the circumfrence of the circle of radius R and the center of the circle is at (0,0). If this equation is discretizied and superimposed on a grid of integer addressable coordinates, the results can be seen in Figure 3-12. Notice that only a small subset of the circle points map directly into the integer coordinate space. Upon careful inspection of the circle, one can interpolate a set of X,Y pairs that approximate the true circle. There are five possible areas through which the true circle may intersect the grid. These are illustrated in Figure 3-13. If a starting point is known, calculations can be made to compare the actual X,Y coordinates to that of the constrained radii of the circle. Depending on the error between the X,Y pairs, one of three different movements can be made to plot the next point on the circumference. The point that has the least distance from the true circle is plotted and then that new position is used to recompute the next point. The three possible directions of movement are illustrated in Figure 3-14. The calculation of points on the circumference of the circle starts at (0,radius) and continues to (radius,1). Each time a new point is calculated in the first quadrant, its three corresponding quadrant points are then calculated, yielding a complete circle. A complete mathematical derivation of the algorithm can be found in the Appendix.



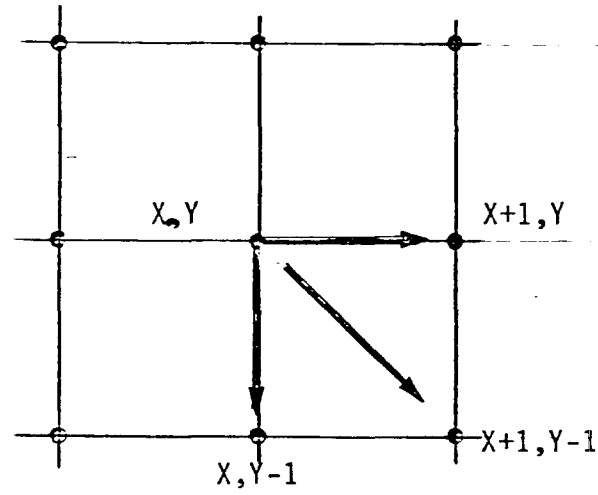
portion of circle overlayed on grid

Figure 3-12



Five possible intersection areas of circle on grid

Figure 3-13



Three possible directions of
movement when circle drawing

Figure 3-14

3.2.7.3. Painting Algorithms

3.2.7.3.1. Recursive Pixel Painting

Painting an object, in its simplest form, is the process of filling an enclosed object with color. In its more enhanced form, the user can paint not only single colors but combinations of colors and patterns. An example of this would be filling a rectangle with a pattern of crosshatches of a blue color. If the processor is not capable of producing colors, various gray scales could possibly be used. Never the less, painting algorithms should produce the same result. A user of a painting program can first draw the outline of the object, and then use the filling function to "spread paint" in the interior of the object. The filling operation starts by replacing the color value of a single pixel, and then spreads throughout the raster, replacing the value of any pixel that contains the old color with the new color. The spreading operation stops whenever it encounters a pixel that does not contain the "old" color. In other words, the algorithm hits the perimeter of the object. The technique is best expressed by the following recursive procedure:

```
procedure Fill(x,y,oldcolor,newcolor: integer);
begin
    if GetPixel(Framebuffer,x,y)= oldcolor then begin
       Setpixel(Framebuffer,x,y,newcolor);
       Fill(x+1,y,oldcolor,newcolor);
       Fill(x-1,y,oldcolor,newcolor);
       Fill(x,y+1,oldcolor,newcolor);
```

```
        Fill(x,y-1,oldcolor,newcolor);
    end
end;
```

The user invokes the filling function by positioning the cursor over a pixel within the region to be filled, and indicating a new color. The fill operation will then spread the color throughout the region. One aspect of the above painting algorithm must be pointed out. If the object is not completely closed, the paint will "spread" outside the object and fill the entire unbounded area.

3.2.7.3.2. Recursive Scan Line Filling

The above filling algorithm has a major problem related to the practically of it. In theory, the algorithm works fine. In practice, it fails because it is highly recursive. Most modern computers have large stacks that can handle recursive events quite well, however the algorithm will quickly use up any realistic stack space if called upon to paint a large object since it recursively calls itself for every pixel it finds that is not of the new color.

Because the target system for this painting algorithm is relatively small, a more efficient algorithm had to be developed. Consider the architecture of the video display processor and video memory. Video memory is arranged as a linear array and the display window is output to the screen as shown in Figure 3-15. Suppose a horizontal line is drawn across the screen from points A to B.

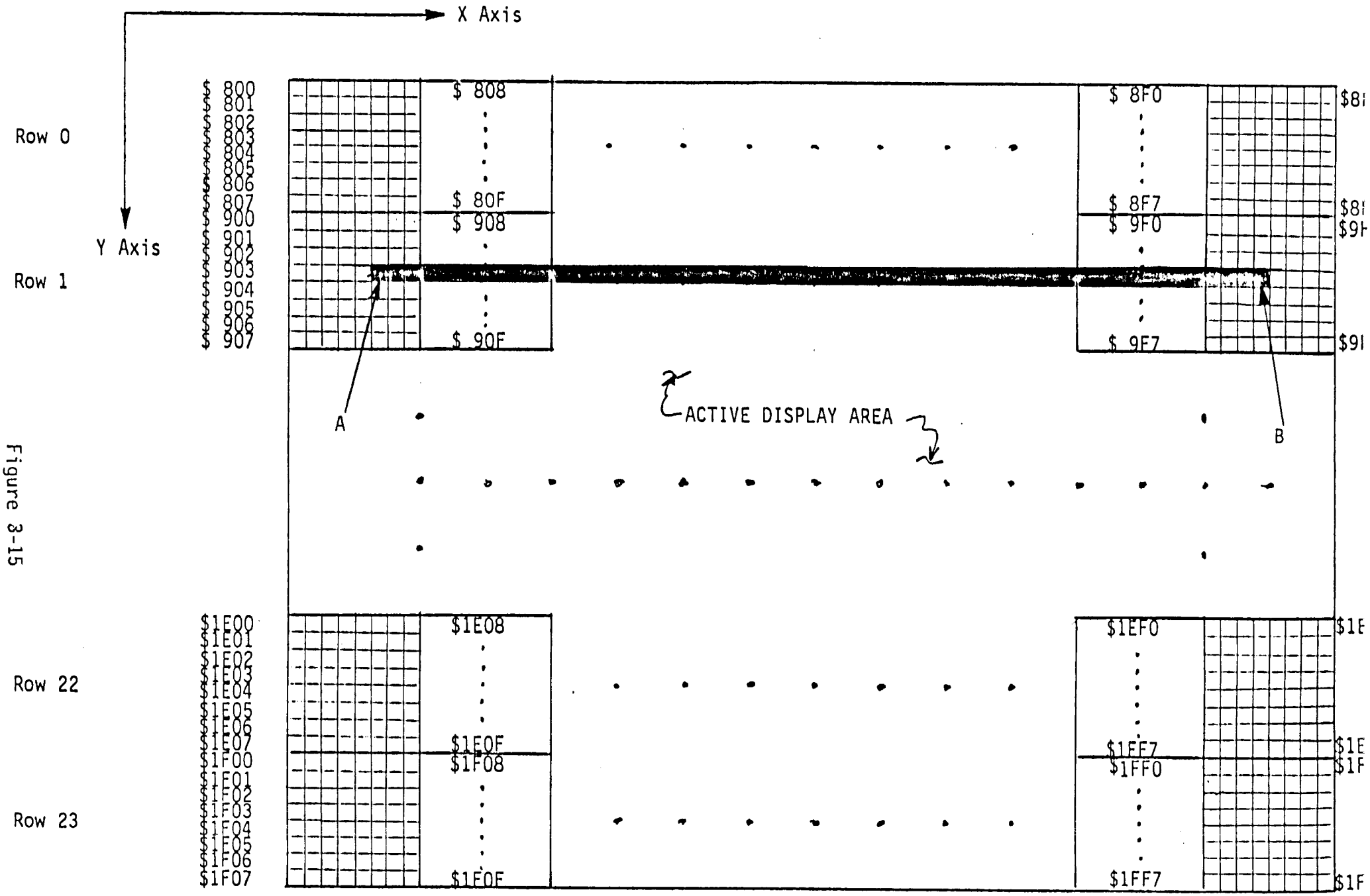


Figure 3-15

Video RAM to Display Screen Mapping

The process to do this is: 1. Find the VRAM address of the starting point, 2. Set the appropriate bits "on", 3. Add eight to the current address to get to the next line segment, 4. Determine if the current byte contains the endpoint and either fill in the bits to that point or turn on all the bits and fetch the next address. Drawing a horizontal line in this manner is very efficient.

Now consider an object to be painted as a collection of horizontal lines with only the endpoints specified. Painting by "lines" instead of pixels would greatly reduce the stack space required. The general algorithm is still recursive in nature, however careful design of the algorithm minimizes the information that needs to be kept on the stack. The algorithm is as follows:

```
Type
   StackRecord = record
                    XMax : integer;
                    y    : integer;
                    ydir : integer;
                    icount : integer;
                 end;
Var
   StackPointer : integer;
      {Current Level of the Stack}
   Stack = Array[1..N] of ^StackRecord;

Procedure PositiveFill(Var XMax:integer);
Var  x:integer;
```

```
Begin
   For x := CurrentPosition to ScreenMax do
      If PixelStatus(x)=Off then
      Begin
         x := TurnOn(x,y);
         XMax := x;
      End; {If}
End;  {PositiveFill}

Procedure NegativeFill(Var XMin:integer);
Var  x:integer;
Begin
   For x := CurrentPosition downto ScreenMin do
      If PixelStatus(x)=Off then
      Begin
         x := TurnOn(x,y);
         XMin := x;
      End; {If}
End;  {NegativeFill}

Procedure Painting;
Begin
            .
            .
            .
LO:
   PositionFill(XMax);
   NegativeFill(XMax);
```

```
    ICount := Xmin;
    ICount := ICount - 1;
    ydir := up;
    Push(XMax,ICount,ydir,yline);
    If StackPtr <= StackMax then
    Begin
       ydir := down;
       ICount := XMin;
L1:    A := yline + ydir;
       If NOT ((A >= 0) AND (A <= 191)) Then
       Begin
L2:       If StackPtr >= 0 then
             Pop(XMax,y,ydir,ICount);
          Else
             Goto L4;
L3:       ICount := ICount + 1;
          If ICount > XMax then
             Goto L1
          Else
             Goto L2
       End Else
          If PixelStatus(ICount,y+ydir) then
          Begin
             XTemp := ICount;
             XTemp := XTemp + 1;
             If XTemp = 0 then
                Goto L2
```

```
            Else
               Goto L3
         End Else Begin
            If (ICount = FF) then
            Begin
               Push(XMax,Icount,ydir,yline);
               If StackPtr <= StackMax then
               Begin
                  yline := yline + ydir
                  Goto L0;
               End Else
                  Goto L4;
            End;  {IF ICount = ff}
   End else
      Goto L4;
L4:
End;
```

The initial values of X and Y are taken from the current cursor position. The user positions the cursor inside the object to be painted. The paint command is typed and the algorithm is entered. Checks are made to determine if the cursor is on an object boundry, or inside or outside of the object. If on a boundry or already inside a painted object, an error condition will result. If not, the current cursor X and Y coordinates of the cursor initialize the XMAX, XMIN and Y variables. The next step is to fill in the line segment to the right of the cursor. This is

done by subroutine POSITIVEFILL which returns an updated value of the variable XMAX. The lefthand part of the line segment is now done by subroutine NEGATIVEFILL and it returns the updated value of the variable XMIN. All the information about the current line now exists and the state is saved on the paint stack by calling subroutine PUSH. This subroutine stores five values, XMAX, Y, YDIR, and ICOUNT, onto the paint stack. Now a direction must be chosen to move along the Y axis to paint the next line. YDIR is arbitrarily set to UP. The line counter is advanced by one, and the new line is filled. If the parameters of XMIN or XMAX do not match the previous values, a new state has been encountered, indicating a "hole" and this condition is saved on the stack. If the object does not change the XMIN or XMAX values, Y is continually incremented by one until the border of the object is encountered.

When the border of an object is encountered, the direction of painting, YDIR, is changed from UP to DOWN. The basic algorithm remains the same but this time through, the pixels are all "painted". When a line is encountered where ICOUNT is different, the paint stack is popped through subroutine POP. The reason this state was saved is that a possibly unpainted area was encountered. The direction is inverted from down to up or vice versa, and the painting continues as described before. When a new border is encountered, the stack is popped and the process continues until the paint stack is empty. It should be pointed out that in this painting algorithm, as with many, if an opening of even one pixel exists in the object, the "paint" will spread outside the object

and consequently "paint" until the border of another object is reached, or the limits of the screen are encountered.

In the actual assembly code implementation of the routine, some efficiency enhancements were incorporated. For example, when the direction of painting along the Y axis is changed, there is no need to repaint already painted pixels. This pixel checking takes time and the results are already known before hand. So, the values of XMIN and XMAX are changed to the new endpoints and painting continues from there. Also, four variables are always stacked or unstacked, making the stack area as small as possible. These four variables are located in a contiguous block so stack transfers can be quick and efficient.

The implemented algorithm is both fairly quick and efficient for moderately sized objects. As the object becomes more complex and very irregular, the stack space and painting time will grow tremendously. For this reason, stack limit checks were incorporated to prevent overwriting system variables. If a stack overflow should be detected during the painting of an object, the user can reposition the cursor to the unpainted section of the object and reissue the command.

It should be pointed out that the current painting command does not support color or gray scale arguments. This is due primarily to the hardware limitations of the VDP. The VDP does not support true bit mapped color graphics. To do this multiple VDP's would be required, each controlling a separate color plane, and while this is very technically possible, it was outside the scope

of the project. Figure 3-16 depicts the architecture of such a design. One VDP can display multiple colors on the screen with certain restrictions that are described in the VDP data sheet in the Appendix. Although one way around these limitations is to include a set of rules to accommodate limitations in the painting and object drawing algorithms, the original scope of the project was to produce a monochrome display. Thus, these limitations were not considered to be necessary to overcome.

The assembly language implementation makes use of a very modular approach to the solution to the problem. Initially, whenever subroutine BEGPAT is called to paint an object, subroutine NORMY is called. Subroutine NORMY changes the user coordinates of the cursor to the VDP coordinates. It changes the X, Y coordinates of the cursor to X = X and Y = 192-Y. Subroutine PIXSTA is then called to check the state of the pixel directly under the cursor. If it is already on, an error condition is reported through subroutine PSHERR. If not, the length of the line segment to the right of the cursor is determined and filled in by subroutine FILLP. This subroutine makes use of subroutine FINDAD to determine the video RAM address that corresponds to the specified X and Y coordinates. For the first time through, this address corresponds to the cursor coordinates. Then the left hand part of the line segment length is determined by subroutine FILLM. Similarly, subroutine FILLM makes use of FINDAD and WRVRAM.

Subroutines PUSH and POP are the paint stack handlers. Subroutine PUSH is called whenever the block of the four painting

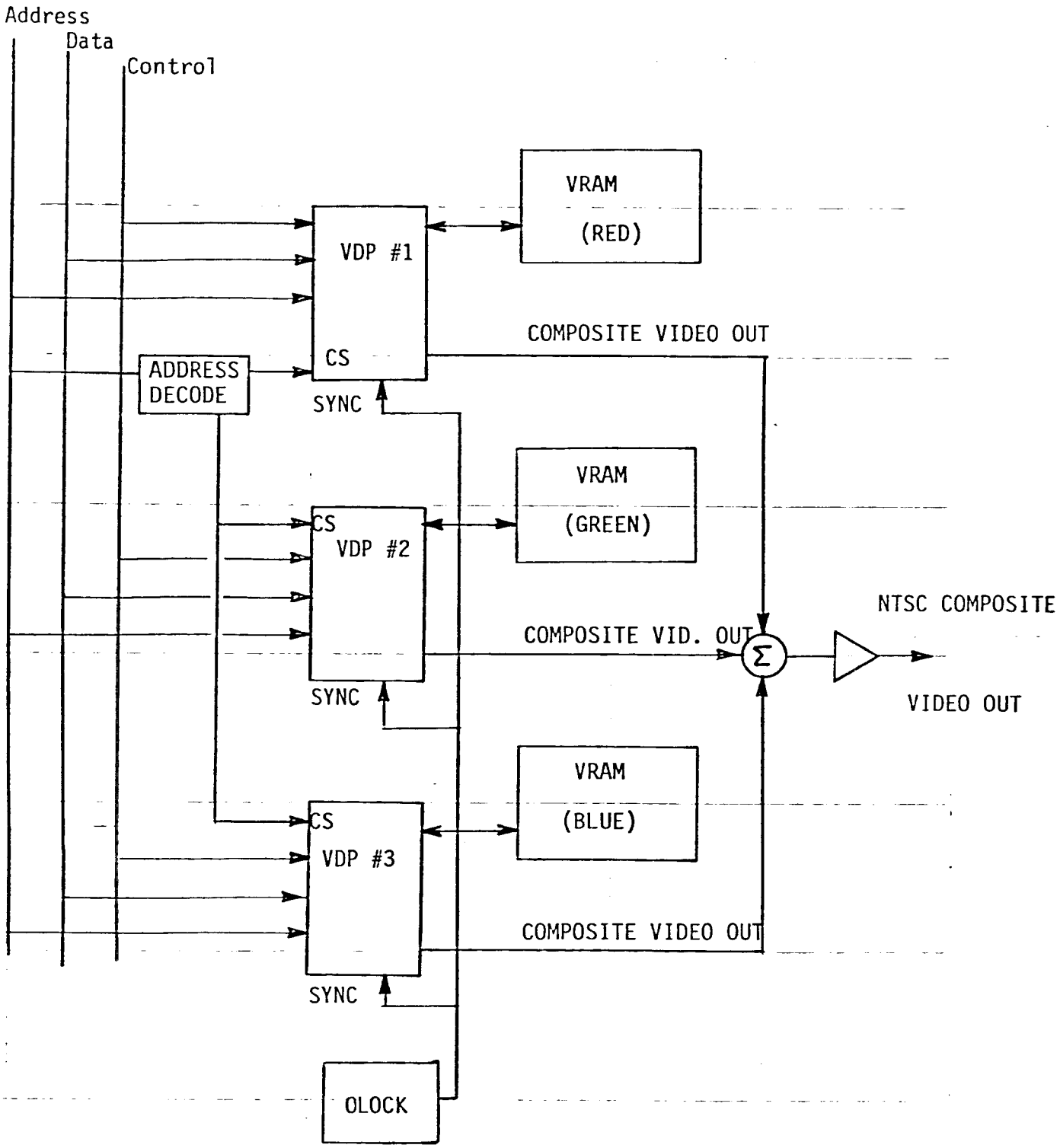

Address
Data
Control
VRAM
(RED)
VDP #1
CS
COMPOSITE VIDEO OUT
ADDRESS
DECODE
SYNC
VRAM
(GREEN)
CS
VDP #2
NTSC COMPOSITE
COMPOSITE VID. OUT
Σ
SYNC
VIDEO OUT
VRAM
(BLUE)
CS
VDP #3
COMPOSITE VIDEO OUT
SYNC
OLOCK

Figure 3-16

state variables need to be saved. Subroutine POP takes care of restoring these variables from the stack to the user area.

3.2.8. Error Handling

In order to provide a good user interface as well as to aid in the debugging of the software, an error handling mechanism was considered essential in the software design. Error handling consists of two functions: detection and reporting. In general, error detection is case dependent while error reporting is case independent and uniform.

3.2.8.1. Error Reporting

The manner in which the overall system reports errors can greatly affect how pleasant and economical it is to use the system and/or the language. Good error diagnostics can significantly reduce the time required for debugging. Well designed error diagnostics should follow these guidelines:

1. The message should pinpoint the errors in terms of the original source statement, rather than in terms of some internal machine representation that is mysterious to the user.
2. The error message(s) should be meaningful and understandable to the user.
3. The message(s) should be specific and localize the problem.
4. The message(s) should not be redundant.

The error diagnostics on this version of the software embody all the above guidelines, but in some cases not as robustly as one would want. However, these shortcomings were realized in the software design, and "hooks" were incorporated wherever possible to embellish the error reporting. They will be highlighted in the following descriptions.

For sake of discussion, assume that an error condition is encountered. As the software is presently designed, an error is reported in the same manner, no matter where or how it is detected. When an error is detected, it is identified with a number. This number is then pushed onto an error stack, identified in the assembly code as ESTACK. Depending upon the type and location of the error, the current task continues or halts immediately. The interpreting of a command line is viewed as a fatal error and processing of the line halts. If a source file is being executed, processing continues with the next line.

Because the software was designed to be hierarchical in nature, error reporting can propagate and be flagged at various levels. This can help both the system user or the software designer. For example suppose the following command line is input:

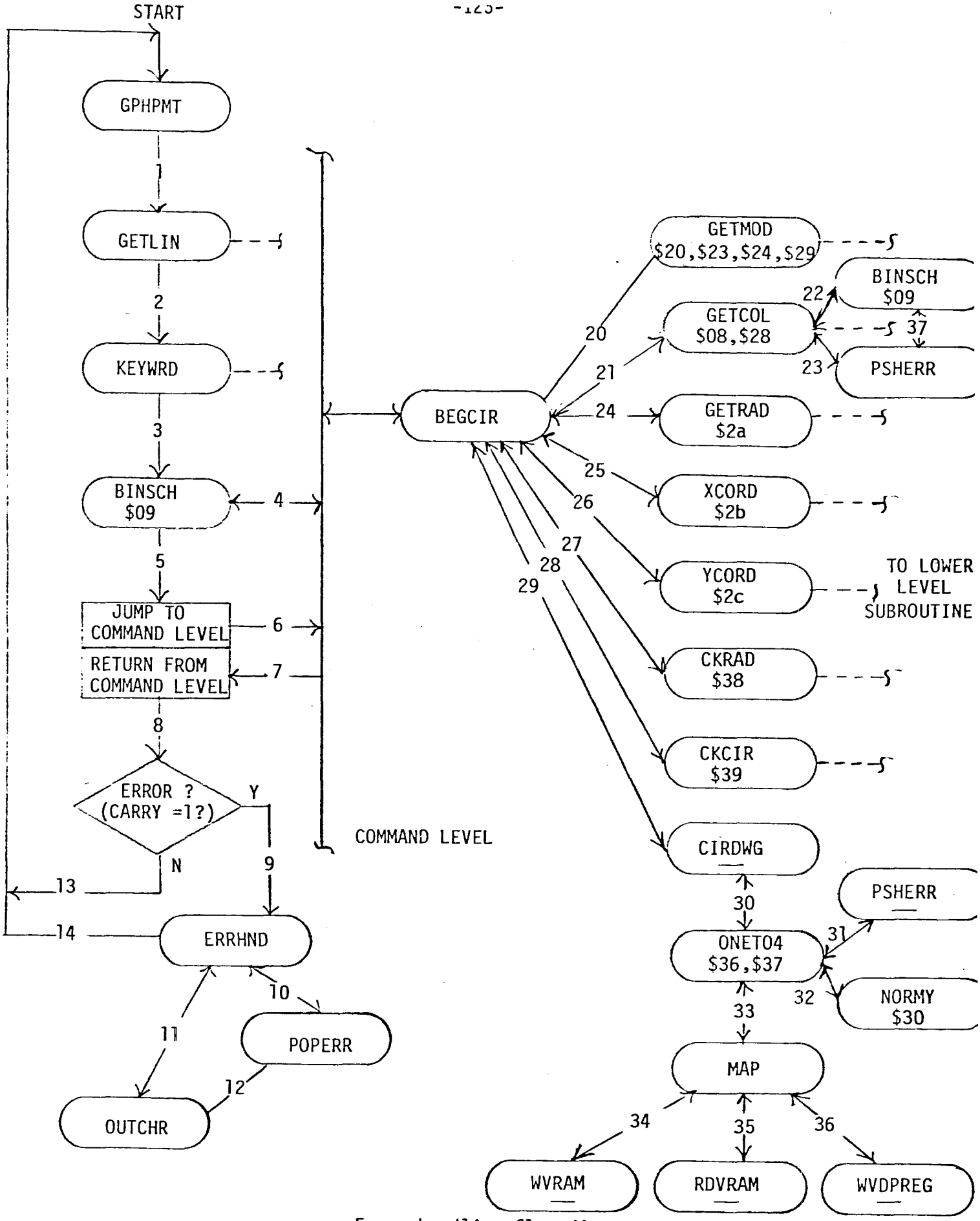
```
cir  a,rd,30,40,40
```

The circle is to be positioned according to absolute coordinates, with a color attribute of red, a radius of 30, and centered at 40,40. The mistake is in the specification of the color. The correct mnemonic for red is "mr". The lexical scanner will

correctly gather the tokens and the syntax will be correct. The error will be flagged at a very low level. To follow the error reporting process refer to the subroutine flow diagram, Figure 3-17. The numbers inside the subroutine blocks refer to the error code that is generated by that routine. The error message produced on the screen will look like:

Error numbers 40, 8, 9

Subroutine BINSCH first identifies the basic error, there is no match in the color lookup table (path 22,37). The error number (9) is placed on the error stack and the carry flag is set. Setting or clearing the carry flag is the universal technique used throughout the software to signal the presence (carry=1) or absence (carry=0) of an error condition. Control returns to the subroutine that called subroutine BINSCH. In this case, it was subroutine GETCOL. At this point, in subroutine GETCOL, the error is further defined as a non-existent color. Therefore, error number eight is placed on the error stack by calling subroutine PSHERR (path 23). In addition, subroutine GETCOL also indicates that an error was made in the color field of this command. As before, the appropriate error code is loaded ($28) and subroutine PSHERR is called. Program control returns to subroutine BEGCIR and since the carry is set the subroutine immediately returns to the main program. The main program interrogates the carry flag, sees it is set, calls subroutine POPERR to print the error message and dump the error stack (path 21,7,8,9,10,11,14). At this point the system prompt is issued and the system is waiting for additional input. The user



Error handling flow diagram

Figure 3-17

can then retype the proper form of the command line.

The information conveyed by the error messages pinpoint exactly where the problem is. The user can refer to the system error messages listed in the Appendix for the meaning of the error numbers. In some cases not all of the displayed error numbers will be of great help to the user. In this case, informing the user that the binary search failed is of questionable value. However, to a person doing additional software development work, this kind of information may prove useful.

Although the use of error numbers is cryptic, a framework does exist for generating complete messages. To generate actual error messages, the error numbers that are popped off the error stack can be used as indices into a table of pointers to stored ASCII messages. These messages can then be placed into the print or output queues and dumped to the terminal. For the initial proof of concept, it was felt that this was not necessary.

A related problem exists for the interpretation of a large source file. The user will not know where the errors are. A possible solution to this problem is to make use of the multiple buffering of the system. Since the command line is moved from the input queue into the 80 character lexical buffer queue, which is not destroyed in the lexical scanning process, the contents of this buffer could be transferred to the print or output queues, dumped out, and then the error numbers or messages could be displayed immediately after that.

3.2.8.2. Error Detection

For error reporting to provide useful information to the user, error detection must be incorporated into each phase of the software and should be as comprehensive as possible. The software developed for this project consists of two major functions: command interpretation and command execution. In the command interpretation phase, many possible failure modes were allowed for. Some of the typical errors identified were:

1. Invalid keywords.
2. Constants beginning with or containing alpha characters.
3. Identifiers consisting of all numbers.
4. Unidentified characters in constants, identifiers, or strings.
5. Invalid token delimiters, end of line characters, etc.

These are generally referred to as syntactical errors as they are generated by the lexical analysis or syntactic phase of the interpreter or compiler. A review of the assembly code comments will reveal what subroutines generate what error codes. Other types of errors generated by the system can be referred to as semantic errors. In general, they are detected at execution or compile time. Examples of these types of errors are:

1. Constants being over or under certain boundary conditions.
2. Invalid or undefined colors or various other object attributes.
3. Incompatible modes with given coordinates.
4. System status messages. ie, paint stack full, message queue full.

These types of errors could occur during a "check" phase of the

command arguments or during the actual execution of the command. As the capability of the software is increased and new commands are added, the amount of code devoted to semantic error checking will increase more quickly than the code devoted to syntactic error checking. This is because the main body of code for lexical scanning is already well defined as are the types of errors to be searched for. The additional error codes will reflect errors in the actual execution of the command rather than in parsing its structure.

3.2.8.3. Error Correction

A third area of error handling capability is that of error correction. This capability allows the interpreter/compiler to repair the error so that processing can resume. Generally, this is a very complex process and a very good argument exists for not incorporating large amounts of time in this area. The basic premise is that in order to correct the code, the interpreter/compiler must know the intent of the programmer. If the intent is obscured by a large number of errors or by vagueness of the program, it becomes difficult or impossible to correct the error(s). The amount of code or execution time that is required for error correction is therefore usually not worth the investment.

4. Language Design

4.1. Introduction

Designing a graphics language consists of defining an instruction set through which the user interacts with the graphics hardware. To some extent, the design of a graphics language depends upon the hardware interface with which the user will be dealing. For example, some systems deal mainly with keyboard input, with the command appearing on the screen directly below the drawing area. Other systems may incorporate the use of a mouse or joystick and objects can be generated by menu selection or direct cursor movement. In order to minimize the cost of this system, a standard terminal was selected as the input device, with the output being delivered to a monochrome or color monitor.

4.2. Design Issues

For any of the interfaces mentioned above, the software design should take into account the following issues:

Simplicity: Features that are too complex for the application programmer to understand will not be used. In the design of a complete graphics system, it is often difficult for the designer to detect this type of problem in the system.

Consistency: A consistent graphics system is one that behaves in a generally predictable manner. Function names,

calling sequences, error handling, and coordinate systems should follow simple and consistent patterns without exception. For example, all relative mode cursor movements should adhere to the same concept of coordinate system designation and direction.

Completeness: There should be no irritating omissions in the set of functions provided by the system. Missing functions will have to be supplied by the programmer, who may not have the necessary access to the computer's resources to be able to develop the functions. Completeness does not imply comprehensiveness; i.e., the system need not provide every imaginable graphics facility.

Robustness: The software should be debugged in all modes of operation, and should provide meaningful error reporting. A graphics language that has a very low tolerance of random white spaces and is very demanding in terms of precise command formatting is not very robust.

Performance: The overall system implementation should stay with its original design objectives. Generally, graphics system users respond well to systems that offer an equally consistent speed of response.

Economy: It is always frustrating to write an application program only to find that it is too bulky or too expensive to use. Well designed software should be

composed of powerful primitives that don't require a string of arguments three lines long.

These issues provide the designer with a number of ground rules for graphics software design. As in any design undertaking, trade-offs are a reality that must be considered. The above guidelines were followed wherever possible during the design of the language for this project. Aspects of the language design that followed the guidelines will be discussed below. It must be pointed out that the initial charter of the project was to implement a few graphics primitives and display the results on a monitor. Since the resulting equipment was intended to be useful, as well as a proof-of-concept, much more time was placed on the software design task than was needed to fulfill the project's initial requirements. This yielded the design of a cohesive set of graphics primitives, from which a subset was chosen for implementation. The primitives that were implemented will be preceded by an asterisk (*) in the following discussions.

4.3. Functional Domains

The commands for the graphics display processor can be grouped into the following six areas:

1. Graphics Primitives: These are high level commands used to generate and display straight lines, text strings, circular arcs, and other graphical elements.

2. Windowing Functions: These commands allow the user to choose the coordinate system for picture definition and to define the boundary of the visible portion of the picture.

3. Display File Manipulation: Allows the user to define, edit, list, and move image files or segment files.

4. Animation File Creation and Manipulation: Enables the system to manipulate or use display image files containing specific attributes for object movement.

5. Control and Miscellaneous: Allows the user to set or reset various system default parameters, as well as make inquiries as to the cursor position, background color, etc.

6. Editor Commands: These may or may not be part of the actual graphics system package. They permit the user to edit display lists, move display lists and change file attributes.

These topics can be subdivided further into their actual command mnemonics. The mnemonics, and their definitions, along with a general description of their argument list and why the structure was adopted are listed below. For the subset of commands actually implemented a set of reference sheets with attributes and usage can be found in the Appendix.

4.3.1. Graphics Primitives

Graphics primitives are the functions that are used to specify the actual lines and characters used to compose a picture. These primitives should be robust enough to allow the user to construct simple, often used objects (cube, square, rectangle, etc.) with little difficulty, and still permit the construction of very complicated objects in a straight forward manner, consistent within the limitations of the overall instruction set. A list of the graphics primitives follows with an asterisk (*) denoting the primitives actually implemented.

* a)	MOV M,X1,Y1	- Move cursor to new location specified by Mode (M), to X1,Y1
* b)	VEC M,C,X1,Y1	- Draw a line from current position to a point specifed by Mode (M), position X1,Y1, with Color (C).
c)	BARC M,C,X1,Y1	- Draw an arc with this point as a defined beginpoint with Mode (M), Color (C), position X1,Y1.
d)	MARC M,Xn,Yn	- Draw an arc with these sets of control points specifed by Mode (M) thru points X,Y.
f)	EARC M,Xn,Yn	- Draw an arc with this point as the defined endpoint specified by Mode (M), to point X,Y.
g)	ARC M,R,C,X,Y,D	- Draw an arc thru a set of defined points specified by Mode (M), with Radius (R), Color (C), Center at (X,Y), thru D Degrees.
h)	DTX C,"xxx"	- Display the string characters with Color (C).
* i)	CIR M,C,R,X,Y	- Draw a circle with Mode (M), Color (C), Radius (R), with center at X,Y.

j) RECT M,C,X1,Y1,X2,Y2 - Draw a rectangle (square) with Mode (M), Color (C), with Diagonal X1,Y1, X2,Y2.

These commands allow the user to construct virtually any object consisting of two dimensional surface descriptions. By various combinations of MOV and the remaining mnemonics, many different shapes can be created. The general format of the command line for the graphics primitives is:

command mode, color, attribute, X1, Y1, X2, Y2

The command is the keyword, mode defines one of two coordinate attributes either relative (R) with respect to the cursor position or absolute (A) with respect to the lower left hand corner of the display, and color specifies one of the sixteen available colors. The attribute argument may vary depending on the keyword. The keyword CIR defines this attribute as the radius, DXT may define it as the text, and the other keywords may not require any attribute at all. The X,Y coordinate pairs are used as begin points, endpoints, or begin-end pairs, according to the keyword selected. For example, CIR would use only one X,Y pair to locate the center of the circle. By comparison, RECT would use two pairs to define the diagonal endpoints of the rectangle.

The philosophy of the structure of the command string needs to be addressed. For example to construct a square somewhere in the middle of the screen, the command sequence would look like this:

```
mov a,100,100          ;position cursor
vec a,ly,150,100       ;construct top line,
```

```
vec a,ly,150,50          ;right hand side
vec a,ly,100,50          ;then bottom
vec a,ly,100,100         ;and finally the left hand side
```

A question that immediately comes to mind deals with the repeated typing of the mode and color values for each line. Two points of view can be expressed here. First, separate commands could be used to set the mode and color. This would eliminate the retyping of the mode and color attributes. The counter argument is that with frequent jumping in and out of these states, the user might assume a certain condition exists when in fact, the existing mode or color was not the desired one. It was decided that the user should have to make a conscious decision in order to minimize unexpected results. Second, this improved the uniformity of the language design, at least in the case of the graphics primitives. And third, since the input device was intended to be a keyboard, some command format was required to convey this information. In addition, since the user is required to type only a single letter for mode, either A or R, and two letters for one of the sixteen colors, it was deemed that the amount of typing overhead was not excessive.

Since cursor movement was to be performed through the keyboard with the system in graphics mode, the MOV command was implemented. It is true that by using selected keys, such as the "h", "j", "k", and "l" keys, moving the cursor by "rubber-banding" lines could be done. Since the project was to implement a proof-of-concept, this was not implemented. However, a joystick/mouse interface is considered to be a better device with which to control the cursor,

and hardware provisions exist on the computer board for interfacing to this type of device.

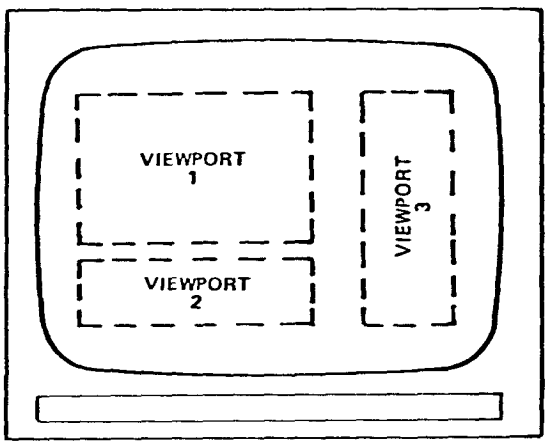
4.3.2. Windowing Functions

Windowing functions allow the user to choose the coordinate system for picture definitions, and to select the boundary of the visible parts of the picture(s). In many graphics systems windowing allows the user to break up the screen into multiple sections to display different areas of information in different parts of the screen. For example, the bottom 10% of active display area could display the status of certain system variables such as mode, cursor coordinates, etc., with the remaining screen serving as the working space. If the user so chose, he could sub-divide the working space into various windows. Figure 4-1. These functions are very valuable in many systems, however, due to the resolution of the selected display and the specifications of this project, the windowing functions were only specified and not implemented. They are shown below.

a) SWD - Set window

b) SVP - Set viewport

It is convenient if window(s) and viewport(s) can be defined separately with a pair of functions. The commands are of the following form:



Multiple viewports, independently scaled and positioned on display surface.

Figure 4-1

keyword xmin,ymin,xmax,ymax

The keywords are SWD and SVP. The arguments xmin, ymin, xmax, ymax define window or viewport positions to extend from (xmin,ymin) at the lower left to (xmax,ymax) at the upper right. A single function could do the work of both of these functions, but it would then be more difficult to change the window independently of the viewport.

4.3.3. Display Files Creation and Manipulation

In graphics systems, the display files consist of sequential graphics primitives that will produce an object when executed. The following commands would allow the user to accomplish these file creations and manipulations.

a) DEFIL - Define an image file with name xxx

b) ENDFL - End image file name xxx

c) LSTFL - List the image file xxx

d) DSFIL - Execute file name xxx

e) RMFIL - Remove image file with name xxx

f) CALLI - Subroutine call to image file xxx

g) RETIF - Return from subroutine call

The above instructions have the basic form of:

keyword filename

As can be seen from the above list of commands, strong similarities exist between manipulating files in this graphics environment and performing typical operations on disk files. This approach to display handling is by no means the only scheme. It is a widely accepted approach that is particularly appropriate to the concepts of the graphics and animation aspects of the language.

To open a display file, one would use the DEFIL command. The user could then compose a file of graphics primitives. To close the file, the user would issue an ENDFIL command. Deleting a file would make use of the RMFIL command. In order to execute the contents of the file, the user would issue the DSFIL. Various files could be chained together by using CALLI and RETIF commands. These files would be executed in the order called. If execution were interpreted, no problems would occur. If, however, the files would need to be compiled, then a link command would have to be designed and used to link the appropriate source display files together.

As was just pointed out, execution can be thought of as either compiling or interpreting. Therefore, a mechanism is required to permit the generation, editing, and execution of these files. There are some aspects of the hardware configuration that must be considered in the design and implementation of such commands. Perhaps the most obvious is that in order to have disk files, disks must exist and so must the software to control and interact with them. These do not currently exist in this system. Thought was given to a method by which one could do a similar operation. The

total addressable memory that the 6502 processor can access is 64K bytes. The language and utilities ROM reside in the top 8K, making the remaining 58K available for user code. If the bottom 2K is used for zero page variables, stack and miscellaneous tables, etc, this results in about 56K of RAM for the user. If this memory were organized similarly to a disk file structure, a limited, but useful system could be designed and implemented. For initial design purposes, suppose that 4K of RAM is viewed as the file name directory block that would contain the file name, a pointer to where it is in the remaining 52K of RAM, and perhaps the size of it. For this somewhat simplistic file arrangement, there would be no need for linked lists of "free blocks" or "used blocks".

The file itself would consist of a beginning-of-file character, a preset number of bytes to act as a file header, the main body of the file, and then an end-of-file marker. Newly created files would be appended to the end of the last file structure. Deleting a file would consist of finding the file in the file directory block, deleting that entry by performing a block move of the remaining file names and attributes over the top of the file name(s) to be deleted. In a similar manner, the file itself would be removed by copying the remaining files over the top of the file to be deleted. The only difference in this operation would be for the last file. It would simply be overwritten with nulls in the file directory portion of RAM.

With the capability of creating and maintaining files, the ability to edit them should also exist. Because of the rather

limited RAM available, a lot of sophisticated editor commands could not be implemented, however an adequate subset of editing functions has been defined and module specifications written. The editor commands will be examined in detail later.

With this somewhat simple but efficient file structure, a number of files could be created, maintained, and executed. In addition, if files were created on another machine, they could be downloaded (in ASCII format) and appended to the current file system.

4.3.4. Animation File Creation and Manipulation

Now that the concept of files and their support has been outlined, the topic of object animation can be addressed. The hardware has been designed with the long term objective of the project to provide a graphics system with a flexible and easy to use language for defining object motion. The choice of this particular VDP was made because of its use of sprites. Five timers were included in order to provide internal clocking capability for various objects. In order to tie this all together, a special file structure, referred to as an object animation file, was conceived. As an example of how an animation file might be used, suppose that five objects have been created, each described in its own separate file. In order to move these objects on the screen, an object animation file would be constructed (using the same graphics primitives as in the original file), with various motion attributes. These attributes would include such parameters as file

name, direction of movement, rate or frequency, repetition of movement, and synchronization. The file attributes commands are listed below.

PATHXY X,Y -This command indicates the points through which the object defined as xxx is to move.

VEC - This is a short hand notation to describe the path of an object between two points in a linear fashion.

DELAY - This command allows the user to program in a delay time for the motion of an object. The delay time becomes effective at the endpoint of an object's path.

REP - Allows the motion of an object to be repeated between the path endpoints, i.e. the movement between endpoints A and B would look like A-B, A-B, A-B....

OSCIL - Permits the motion of an object to be repeated in a back and forth manner, (i.e. A-B-A-B-A-B....).

START - Starts the motion of an object in the object animation file.

STOP - Stops the motion of an object.

TIME - Defines the time in seconds at which an object is moved across the screen.

SYNC - Synchronizes the movement of one object with one or more other objects.

In order to more fully explain the uses of these commands, consider the following specific example. Suppose a circle were to be created with the intent of moving it back and forth across the display screen in an oscillating motion. The user would create the following file:

```
 1  DEFIL 3            ;  Define object named #3.
 2  TIME 5             ;  It will take 5 seconds to go from its
                       ;  beginning point to its end point.
 4  DELAY 10           ;  At each end point it will wait 10 seconds
 5                     ;  before doing anything else.
 6  OSCIL n            ;  The object motion will oscillate.
 7  SYNC n,X,Y,i       ;  Start the movement when object n reaches
 8                     ;  point X,Y in line "i".
 9  VEC X1Y1,X2Y2      ;  This is to be the desired path.
10  MOV a,X,Y          ;  Now define the object.
11  CIR a,ly,30,40,40  ;
12  ENDFL              ;  End the file.
```

The first six lines define the object movement attributes and

the remaining lines define the object. To distinguish this from an ordinary display file, the TIME attribute must follow the DEFIL statement. In order to properly interpret the commands, a file must be defined and built. This file can then be run through the interpreter and a special motion attribute or control block will be built in the processor's memory. In addition, during the actual composition of the object, sprite(s) will be allocated for use in the object's construction, instead of having the object mapped into the "static" video memory plane. This is done because a sprite can be used to move an object more quickly than it can be done by erasing and then redrawing the object.

To see the type of object built by this file, one would use the DSPFL in command. The portion of the file that builds the object would be executed. The object would be constructed in the static video plane. To display the object and its associated motions, one would use the START "n" command where "n"is the object number. Stopping the object would be done by using the STOP "n" command.

Suppose the object's motion were to be repetitive instead of oscillatory; Line six (6) would be changed from OSCIL to REPEAT. If the motion were to be along a curved path, Line nine (9) would be changed from VEC X1Y1,X2Y2 to a series of PATHXY X,Y's. These points would be considered control points for a curve fitting algorithm to calculate the discrete X,Y points of the path to be taken by the object.

In the interest of keeping the scope of the project to a

reasonable size, these commands were not implemented. However, since the ability to provide objects with motion capabilities was an important language goal, a mechanism for doing this was designed and is outlined in the following discussion.

4.3.4.1. Mechanics of Object Animation

4.3.4.1.1. Building the Animation Block

First, the object to be animated must be built in an animation attribute file as described above. This source file is then directed to the interpreter. The effect of the DEFIL 3 command is to start constructing in memory an object animation attribute table whose name is "3". If the next line in the input file were not a TIME command, object 3 table would be "erased". Since it is present in this example, the next byte in the table is the binary representation of the specified time.

The next line to be interpreted contains the DELAY command. This line will cause the interpreter to take the numeric value of the delay time specified, convert it to its binary equivalent, and place that value into object three's animation attribute table. A similar action will take place when the line containing the OSCIL command is encountered. The number of repetitions for the oscillation will be appended to the animation attribute table. The process of building this animation attribute block continues until a keyword for graphics primitives or cursor control is encountered, which signals the loader program that the attribute block for this

particular object is to be closed. Encountering any additional animation commands in the body of this file will result in an error.

4.3.4.1.2. Sprite Allocation

Assuming no errors are encountered, the object is then constructed in the "static" video memory, i.e. no sprites are allocated to compose the movable object. The next step is to determine the optimum pattern for the allocation of the 16 x 16 pixel sprites. (The sprites are magnified to achieve the maximum possible area to assign to an object.) This optimization could be done in one of three ways. The first method uses information supplied by the user in a command, which would specify the width and length of the object. The command could have the form:

BOX X, Y, L, W

where BOX is the keyword, X,Y are the upper left hand coordinates of the "box" to be drawn around the object, L is the length (X direction) in pixels, and W is the width (Y direction) in pixels. This information would then be used by lower level subroutines to calculate an integer number of sprites to be allocated in an end-to-end fashion (side-to-side and top-to- bottom) to completely cover the object. The calculated X,Y coordinates of each separate sprite would then have to be loaded so they would be correctly positioned on the screen, adjacent to each other. A check would have to be incorporated to insure that no more than four sprites are assigned on the same horizontal scan line at the same time. (A

limitation of the VDP; see Section 2.1.3.4 for a discussion of the problem and possible solutions.)

Another method to box an object is to use a variation of the painting algorithm described in Section 3.2.7.3. At the end of the display list the user would position the cursor inside the object to be painted. The object would then be painted with a transparent color using a slightly modified painting algorithm that would save the lowest values of XMIN, YLINE and the largest values of XMAX, YLINE. These two endpoints would be used to construct a box around the object. The sprites would be allocated as described in the above paragraph.

The third method would be to keep a list of the smallest and largest X,Y coordinate pairs when the pixels of the object being generated are plotted. A rectangle could then be constructed using these X,Y pairs as the endpoints of the rectangles diagonal, as discussed in the above paragraph. For most applications, this method is probably the most efficient.

There are other methods that could be used in the boxing of objects. One such method is a variation of the Cohen-Sutherland clipping algorithm. If the capability of the system is expanded in the future to support multiple windowing and viewporting, this algorithm would be a good choice for clipping. Once installed in the system, a small amount of additional work would be required to tailor it to the boxing problem.

4.3.4.1.3. Object Image Transfer

Regardless of which method is selected to box the object, once the sprites have been allocated and positioned, the object in "static" memory must be transferred to the sprite(s). A direct approach to the problem is to start in the upper left hand corner of the box and interrogate each pixel within the boxed area on a line-by-line basis from top to bottom . When a pixel that is "on" is encountered, its X,Y coordinates are used to determine which are the corresponding X,Y coordinates of the sprite pattern table, so that the pixel could be determined and then turned on. Many of the support subroutines, such as RDVRAM, WRVRAM, FINDAD, and MAP already exist to support this function.

4.3.4.1.4. Color Transfer

The remaining task is to transfer the appropriate color of the object to the sprite color block. This could be done in one of two ways. The first method is to look up the current foreground color of the static video memory and transfer it to the correct byte in the sprite attribute table(s) of the sprite(s) that compose the object. The second method is to incorporate a COLOR command and transfer the color specified directly to the sprite(s) attribute table.

4.3.4.1.5. Object Removal

The last task is to remove the object which is now hidden by the sprite(s), from static memory. Since the perimeter of the

object is already known from the previous boxing sequence the video RAM addresses can be determined by using subroutine FINDAD. Once this has been done, using subroutine WRVRAM to set the appropriate locations to $00 would erase the object.

4.3.4.1.6. Object Motion

If the object were to be moved according to the specified motion attributes, timers would then be allocated and initialized to the time values in the animation attribute table. At each timer "tick", the composite object would be moved by updating the X,Y coordinates of each sprite that composed the object. The incremental movement time would be computed by dividing the time from endpoint to endpoint by the total number of pixels along the specified or computed path. To avoid irregular pattern movements, the sprite position should be updated when the end of the active scan portion of the display is reached. This would allow the most time for "glitchless" object position updating.

When the object has reached a defined endpoint, the appropriate delay time would be loaded into a timer, and upon time out, the movement would repeat or oscillate according to the command in the animation file.

The above description illustrates how a single object can be created and moved using the animation command set discussed earlier. Multiple objects would be created and moved in the same manner. In addition, many of the low level support subroutines exist and can be built upon for future enhancements.

4.3.5. Control and Miscellaneous Functions

This section outlines the control and miscellaneous commands that control the display screen attributes and informs the user of the "system status". The commands are self explanatory and are listed below:

* a) CLRSC - Clear the screen
* b) INTIG - Initiate graphics mode
* c) REVID - Reverse video (Reverses foreground and background color)
* d) PAINT - Colors an object with color specifed
* e) CURSO - Turns cursor on
* f) CURSF - Turns cursor off
* g) CURPOS - Displays the absolute cursor coordinates
* h) SETBGC - Sets the background color
* i) SETFGC - Sets the foreground color
* j) SETBDC - Sets the backdrop color
* k) SETCC - Sets the cursor color
  l) STAT - Displays the system status consisting of: cursor X,Y position, window and viewport dimensions, foreground color, number of display files, and memory available

The only commands that require an argument are the ones that deal with painting or coloring. The colors available to the user are listed in Figure 2-10.

One point should be discussed here. The current configuration

of the system only supports two colors; the background color and the foreground color. For example, suppose the current foreground color is blue. If an object is constructed, all the components of that object will be blue. If a SETFGC command is issued and changes the foreground color to red, the object that was blue will change to red. This problem and possible solutions to it are discussed in Section 2.1.3.3.2.

The coloring of sprites is a different matter. Sprites, by their definition can be any of the sixteen colors desired, no matter what their position is on the screen or their level of display priority. To use this feature to its fullest extent, a mechanism to construct small, multiply colored objects from sprites should exist. This mechanism can be patterned after the preceding sections discussion of animation files. That discussion pointed out that in order to have an animation file, the TIME command had to immediatley follow the DEFIL command. The fact that no TIME command was used could signal the interpreter that even though no animation attribute table needs to be constructed, the assembling of the object should be done with sprites instead of keeping it in "static" video RAM.

4.3.6. Editor Commands

Because the concept of a file system has been outlined for the graphics processor, an editor should exist to edit the created files. The basic editor commands are indicated below.

a) ed - Invokes the editor for use with filename xxx.

b) i - Inserts a new line

c) d - Deletes a line

d) w - Writes out the newly edited file

f) q - Quits the editor without saving any changes made

g) a - Append to the end of the file

h) b - Go to the beginning of a file

i) . - Print current line

j) .+n - Go to and print line n lines ahead of current line

k) .-n - Go to and print line n lines before current line

l) cr - Advance thru file line by line

m) - - Go backwards thru a file line by line

n) s - substitute a character or string for a character or string

o) /xxx/ - search for a character or string

Those readers familiar with the UNIX operating system will find a close resemblance between these commands and those used by ED on UNIX. A brief description of the actual mechanics of how the operating system would handle editing follows. When the editor is invoked by typing "ed filename", a search is made in the file sytem directory for the file and a pointer to it. If the named file is not found, the user is informed and the processor returns to the monitor. If the named file is found, the system obtains the address of the file and makes a working copy of the file, starting the copy at the highest RAM location. All modifications are performed on the working copy. Once it has been updated, the new file is appended to the end of the present file structure. The part of the file structure that was below the file being edited,

(including the now-saved new version of that file), is copied over the old file. In addition, the file address pointers in the file directory are updated as well.

In order to accomplish this two efficient routines need to exist. The first one is a block move forward and reverse subroutine. This moves the files on a byte by byte basis either forward or backward in RAM. A fast and efficient subroutine was written that would move 10K bytes in approximately 0.52 seconds. The second subroutine that is needed is a string or character search. This was written and proved to be very quick. These two subroutines can provide the foundation for continued development of the editor. In addition, all the subroutine specifications have been written and the subroutine data flow path has been developed. These have not been included in this thesis because the main thrust of the project centered around the graphics hardware and language. An interim solution is to create display files on a host system and use the editing features contained in the host's operating system. The display files can then be downloaded and executed by the graphics display processor.

5. Contributions

5.1. Summary of the Project

The primary motivation for this project was the need for a low cost graphics system, capable of producing relatively simple objects, which would include both hardware and software support to move or animate the objects without screen flicker. In addition, because of the intended use of the system, a reverse video operation that was also flicker free was required. In reviewing the scope of such a project, many problems needed to be addressed in the "proof of concept" design of this system. At the time of its design, the VDP was the only device of its kind that was available.

The project was divided into two tasks: the hardware design and the software design to support the "proof of concept" of the hardware. The decision to use the VDP over the microprocessor/ bitslice approach was influenced by two factors. The first factor was economics. The VDP and its associated support hardware are much less costly than the microprocessor/bitslice and their associated hardware. Second, the VDP embodied the unique concept of multiple display planes (both pattern planes and sprite planes) that made it easier to meet the animation objective at a reduced cost.

The VDP needed to be matched with a microprocessor that would act as a front end device to effectively manage the VDP. The off loading of functions was done in hardware where it was deemed practical to do so. This resulted in the incorporation of hardware

timers, and additional ACIA's. The additional ACIA allowed the graphics processor to be connected to a host computer. The timers were used to timeslice the video display and provide a time base for the objects in the animation file.

Initially, the software task of the project dealt with the implementation of a few commands to demonstrate the capabilities of the hardware that had been designed. This task became somewhat more complicated as the need was recognized for support software to effectively demonstrate command interpretation. First, the language had to be designed. This language consisted not only of graphics primitives, but also included animation primitives. To support the decoding of the commands, an interpreter had to be designed and written. This led to the design and implementation of the terminal support, lexical analyzer, parser and error handler software. The editor and file handler were conceptually designed and presented in this thesis. This software development increased the design time considerably, but provided a strong foundation for future enhancements.

5.2. Thesis Contributions

The author believes that the following activities constitute the contributions and goals of the project:

(i) In Section 2 the design of a low cost graphics engine consisting of a 6502 microprocessor and TMS 9918 video display controller was presented. An evaluation was performed of

possible alternative architectures that could be used, and reasons were presented for the final choice of the microprocessor/VDP architecture.

(ii) The resultant system produces a low resolution graphics display capable of performing rapid screen updates, displaying multiple planes, and generating composite objects with the additional attributes of continual rapid movement of the object on the screen without flicker.

(iii) Section 3 introduces the design of a graphics language and support software to fully utilize the hardware. The graphics primitives were designed in detail with respect to syntax and semantics. The animation commands and their actions were specified extensively, along with full descriptions of the perceived implementation details.

(iv) The software design and implementation of the interpreter to handle the graphics primitives are discussed in detail in Section 4. This section includes the monitor/debugger, terminal interface, lexical scanner, parser, display file construction and usage, and error handler.

(v) Section 4 also evaluates various algorithms for line and circle generation, and for painting. Novel painting and circle generation algorithms are described along with their actual software implementation details.

5.3. Recommendations for Future Work

In view of the continuing advances in microprocessor and display controller technologies the author proposes the following subjects for future work:

5.3.1. Hardware

a. The incorporation of a second and possibly a third VDP, in parallel with the original, to increase the number of sprites and object planes.

b. Interfacing a joystick/mouse to the system (the interface hardware exists, the software needs to be developed to handle it) to greatly enhance the user interface.

c. The design of a high resolution microcomputer/bitslice display controller. that would use the concept of multiple paralled display planes similar to the VDP.

5.3.2. Software

a. Continue the development of the language as described in this thesis.

b. In addition to incorporating a display list editor, design an object editor that can remove an object from the screen by following outline pixels and delete that portion of the object in the object's display list.

c. Add user friendly features such as on-line help files for the use of various commands.

d. Allow objects to be constructed by rubberbanding.

BIBLIOGRAPHY

Aho, Alfred V., and Jeffery D. Ullman, Principles of Compiler Design, 2nd edition., Addison-Wesley, March,1978.

Beetem,John., "Vector Graphics for Raster Displays," Byte ,Vol. 5, Number 10, (October, 1980), 286-293.

Bresenham,Jack., "A Linear Algorithm for Incremental Digital Display of Circular Arcs", Association for Computing Machinery, February, 1977.

Browne,Jack and Charles Melear, "Simplifying Video - Display Design by Using a Versatile IC Controller," Electronic Design, Vol. 27, Number 13, (June, 1979), 69-80.

Butland, David and Judy, "An Easy-to-Use Graphic Drawing Package," Computer, Vol. 13, Number 7, (July, 1980), 69-80.

Clark,James, "A VLSI Geometery Processor for Graphics," Computer, Vol. 13, Number 7, (July, 1980), 59-68.

Capowski,Joseph J., "The Neuro Science Display Procesor", Computer, Vol. 11, Number 11, (November, 1978), 48-56.

Conrac Division Raster Graphics Handbook, Chapter 4, SIGGRAPH Core Standards, 4-1 - 4-30, Conrac Corporation, 1981.

Jordan,B.W. Jr., and R.C. Barrett, "A Cell Organized Raster Display for Line Drawing," Proceedings of The IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure, (May, 1975), 7.

Laws,B.A., "A Gray Scale Graphic Processor Using Run Length Encoding," Proceedings of The IEEE Conference on Computer Graphics, Pattern Recognition and Data Structure, (May, 1975), 7.

Loceff,Michael, "A New Approach to High-Speed Computer Graphics: The Line," Computer , Vol. 13, Number 6, (June, 1980), 56-65.

Lucido,Anthony P., "An Overview of Directed Beam Graphic Display Hardware," Computer, Vol. 11, Number 11, (November 1978), 29-37.

Machover,Carl, "A Brief, Personal History of Computer Graphics," Computer, Vol. 11, Number 11, (November, 1978), 38-45.

Machover,Carl, and Robert Blauth, ed., The CAD/CAM Handbook , Computer Vision, 1980.

Masscomp, Inc."Masscomp Computer Graphics Users Manual," Masscomp, Inc., 1983.

Mello,James A. and John Greaves, "Multi-Processing Improves Throughput and Response in a Vector to Raster Converter," Computer Design, Vol. 19, Number 3, (March, 1980), 127-133.

Myers,Ware, "Computer Graphics: The Human Intrface," Computer, Vol. 13, Number 6, (June, 1980), 45-54.

Myers,Ware, "Computer Graphics: A Two Way Street," Computer ,Vol. 13, Number 7, (July, 1980), 45-54. 49-57.

Myers,Ware, "Interactive Computr Graphics: Flying High, Parts I & II," Computer, Vol 12, Number 7, (July 1979), Vol. 12, Number 8, (August, 1979), 52-67.

NEC Electronics,"The uPD7220/GDC Design Manual", NEC Electronics USA, Inc., 1983.

Newman,William M. and Robert F. Sproul, Principals of Interactive Computer Graphics, 2nd ed., McGraw-Hill Inc., 1979.

Preiss,Richard B. "Storage CRT Display Terminals : Evolution and Trends," Computer, Vol. 11, Number 11, (November, 1978), 20-27.

Shaw,Alan C., The Logical Design of Operating Systems , Prentice-Hall, 1974.

Sutherland,Ivan E., Edaward Cheadle, James Kajiya, "A Random-Access Video Frame Buffer," Proceedings of The IEEE Conference on Computer Graphics, Pattern Recognition and Data Structure, (May, 1975), 1.

Texas Instruments, "TMS 9918 Video Display Processor Data Manual", November,1980.

Ziebelman,Peter, "Display Chip and Basic Keep Graphic Costs Down," Electronic Design, Vol. 28, Number 22, (October 25, 1980), 135-141.

APPENDIX

Texas Instruments TMS 9918 Data Manual

Mathematical Derivation of the Circle Drawing Algoirthm

Monitor Commands and Usage

Implemented Commands and Usage

System Schematic

Board Photographs

Sample Display Files

Photgraphs of Monitor (Execution of Display Files)

Creating Display Files Downloading to the Graphics Machine Under Unix

Graphics Interperter Source File

Download Program Source File