THE GRAPHICS DISPLAY CONTROLLER BOARD

FOR THE TRS-80 COLOR COMPUTER

APPROVED:

TO AMY AND MY PARENTS

THE GRAPHICS DISPLAY CONTROLLER BOARD

FOR THE TRS-80 COLOR COMPUTER

BY

SEUNGYOON PETER SONG, B.S.E.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

AUGUST, 1986

ACKNOWLEDGEMENTS

ABSTRACT

This thesis describes the design and implementation of a graphics display controller board. The NEC uPD7220 Graphics Display Controller chip is the brain of the system that can execute high-level figure drawing commands. The board is interfaced to a TRS-80 Color Computer running under the OS-9 operating system. The device driver for the board is written in the high-level language C. A set of utility routines is implemented in the device driver to simplify the programming. The design and simulation of the hardware was done on the SCALDsystem, and the board is built with the aid of Merlyn-PCB.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

"A picture is worth a thousand words."

A proverb.

"Computer Graphics" is one of the most fascinating fields in computer applications today. Its use ranges from simple graphics editors to more complex simulators and picture generators, limited only by our imagination. It is also the basis of "friendly" human interface. What can be said in many words, a single picture can say better. A picture is worth a thousand words.

With steady advances in VLSI technology, there emerged several dedicated graphics display controllers such as NEC's uPD7220 and Hitachi's 63484 that can execute high-level commands to draw lines, arcs, rectangles, area fills, as well as managing display memory. These chips greatly simplify the design of both hardware and software, which reduces the system cost considerably. These chips have opened up a new realm in computer applications, mainly with engineering and design workstation, where graphical input and output makes the systems easy to use. They are yet to be found in personal computers, except in Commodore's Amiga, but it is only a matter of time before personal computers will have these dedicated graphics controllers in them. The purpose of this thesis is to present the graphics system developed for today's personal computers.

1

1.1 Design Goals and Specifications

The primary goal of the system is to produce a fairly sophisticated graphics environment that is easy to use. The secondary goal is to implement it as a server in the "Look Ahead" local area network. The purpose of the secondary design goal is to evaluate the robustness of the network by feeding a large amount of bit-mapped data to the server. These two goals translated into the following requirements:

1. The use of NEC uPD7220 GDC chip, the most advanced graphics display controller device available at the time,

2. The resolution of about 800 by 680 so that a page of text can easily be displayed,

3. The use of the 6809E-based processor as the host to function as a server in the network,

4. The capability to handle DMA operations.

These requirements resulted in the following specifications:

1. The display memory must be of at least 64K bytes. Because of the size, dynamic memory should be used to minimize the cost. Since the 7220 GDC is fully capable of handling dynamic memories, this poses no additional problem.

2. The video, or pixel, rate is calculated to be 16M Hz. or higher to support the desired resolution. The detailed calculation is presented in a later section.

3. The device driver must be more intelligent than merely be able to handle basic input/output functions to fully utilize the capabilities of the 7220 GDC.

4. A DMA controller chip should be used in order for the system to be a server of adequate speed. The MC6844 DMAC was chosen for its compatibility with the host processor.

1.2 Thesis Organization

The presentation includes a brief discussion of the NEC uPD7220 Graphics Display Controller (GDC) chip, the work-horse of the system, in relation to the desired capabilities of dedicated graphics controller devices. The design is tested and simulated on the SCALDsystem, implemented with the aid of the Merlyn-PCB system , both of which use computer graphics heavily. The detailed discussion on the uses of these two systems is omitted from the presentation; a user guide is being written for that purpose.

The concept of raster graphics is discussed, along with the 7220 GDC, in chapter 2. The hardware description of the graphics system is presented in chapter 3. A brief discussion on implementation is in chapter 4. The software description is presented in chapter 5. Chapter 6 contains the programming guide to the system. And finally, chapter 7 concludes the presentation with a few recollections and thoughts that have occurred during the implementation of the system, as well as suggestions for further work.

Throughout the discussion, the terms GDC is used to denote the NEC uPD7220 GDC chip and the GDC board to denote the hardware of the implemented system. The hardware implementation was completed in two phases; the first on the printed circuit board produced with the CAD systems in the TRAC laboratory, the second with the additional hardware on the printed circuit board to implement the modifications and expansions of the design. Unless specified otherwise, the term GDC board refers to the modified hardware.
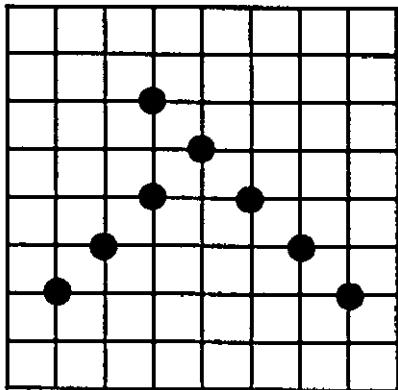
Chapter 2

FUNCTIONS OF GRAPHICS DISPLAY PROCESSORS
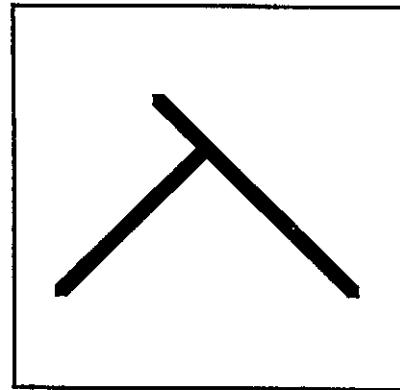
In this section, the capabilities of the GDC are presented to clarify the upcoming discussion. First, the basic concepts of raster graphics are presented, in relation to the capabilities of the GDC.

2.1 Raster Display Process

When we, as human beings, draw images like polygons or circles, we do it with lines and arc segments. This is natural for us. However, it is certainly possible to do the same with numerous dots. We can draw a line with a series of dots placed close to each other. If enough dots are placed close enough, the resulting line could look as if it has been drawn with a single stroke. Furthermore, the dots do not have to be placed in any fixed order in relation to the image. For instance, to draw a circle, the dots do not have to be placed in clockwise nor counter-clockwise fashion. They could be placed in any order, so long as the resulting figure forms a circle. This is the principle of raster graphics. Instead of drawing an image with a series of strokes of lines and arcs, in raster graphics, a series of dots are used to compose the image. This is illustrated in Figure 2-1. If enough dots are placed close to each other, the resulting image of Figure 2-1a would look more like Figure 2-1b.

4

a) Raster image   b) Vector image

Figure 2-1. Raster and vector displays.

In raster display, the screen can be thought of as an uniformly divided array of picture elements, or pixels. As electron beam sweeps across the screen, the beam is turned on and off to create the intended image. The sweep pattern is always the same (left to right, top to bottom), and is independent of the image being displayed. A partially displayed image of Figure 2-1 using scan lines is shown in Figure 2-2. The horizontal retrace is used to bring the beam back to the starting position of the next line, hence the beam is turned off during this period.
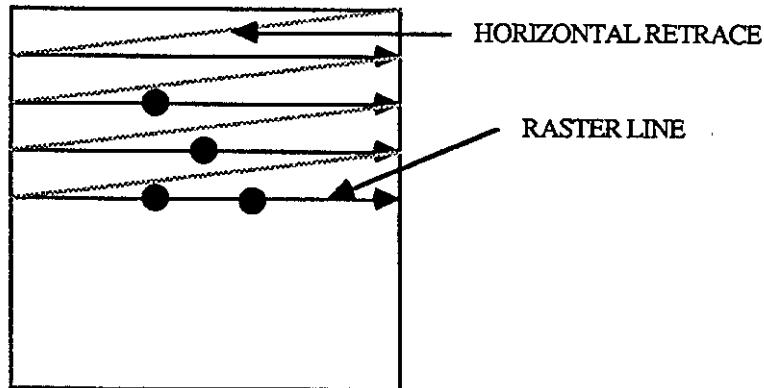
Figure 2-2. Raster scan line.

In vector displays, on the other hand, the beam does not sweep the entire screen. Instead, the beam is controlled to draw only the image. For instance, to display an image composed of a circle only, the electronic beam path will trace out this circle, and the circle only. The majority of electronic oscilloscopes (in X-Y mode of operation) falls into this class. With a vector display device, only two lines are required to generate the picture shown in Figure 2-1. Thus it is faster to draw simple images with vector displays than with raster displays. However, the drawing speed depends heavily on the number of line segments in the image, and with sophisticated images, the drawing speed of raster displays may be faster. Also with Cathod-Ray-Tube screens, the screen refresh interval needed to maintain the flicker-free picture imposes an upper limit on the complexity of the image. For this reason, CRT's are generally less suitable for vector displays than Direct View Storage Tubes (DVST), which does not require refresh. For the same reason, the majority of electronic oscilloscopes are not designed to display more than simple waveforms.

2.2 Synchronization

Images are shown on the screen of a raster-scan CRT monitor as a function of time. In order to obtain a sharp, still picture, the beam must pass over the same trace path, keeping the same on/off sequence, repeatedly. To create a meaningful picture, the beam must be controlled meticulously about the positions of the pixels it project on the screen. To achieve this, horizontal and vertical synchronization signals are supplied, in addition to the pixel information, to the monitor as timing references.

Inside raster-scan CRT monitors are two free-running oscillators that control the horizontal and vertical deflections of the electron beam. These two oscillators operate independently of each other, and even without the synchronization signals. They keep the beam from going off the screen by deflecting it back to the starting point of the next line, or to the top if at the bottom, when it reaches the end of a raster line. The synchronization signals are used only to control the timing of the end of raster lines, not to drive the beam left to right or top to bottom. The sync signals take the form of negative voltage pulses, as shown in Figure 2-3.
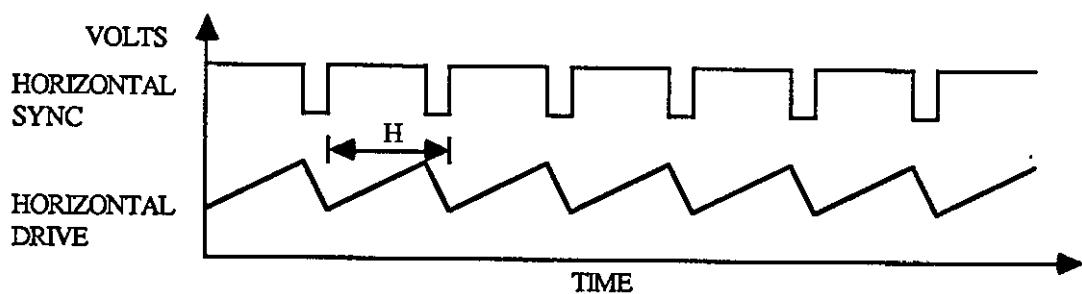
Figure 2-3. Horizontal Sync and Drive signals

The horizontal sync signal is used to control the length of a raster line. During the rising edge of the horizontal sawtooth drive signal, the beam is deflected toward the right of the screen. During the falling edge, it is retraced back toward the left of the

screen. The width of the horizontal sync pulse is in the range of 3 to 6 micro-seconds, and bears no relationship to the length of a line; it just needs to be long enough to be able to discharge the horizontal drive signal. The length of a raster line is determined by the period of the horizontal sync signal. The nominal range is from 33 to 67 micro-seconds, for the frequency of 15 to 30 KHz.

The beam is returned at an accelerated rate to the start of next line during the horizontal retrace period, maximizing the visible portion of the raster lines. This will increase the resolution somewhat. However, the horizontal resolution depends more on the video rate, the speed of which the beam is turned on and off. To maximize the resolution, the fastest possible pixel clock that the monitor will tolerate can be used.

The similar operations occur on a vertical sweep of the screen. The vertical sync and drive signals look much similar to their horizontal counterparts shown in Figure 2-3. A vertical sweep is also composed of two periods; scan and retrace. During the vertical scan, the number of raster lines that is at least equal to the vertical resolution is traced out on the screen. During the retrace period, beam is pulled back to the beginning of the first line on the top of the screen. The period of a vertical sweep, or a frame, must be a multiple of the period of a raster line. The beam path in a frame is shown in Figure 2-4.

a) Vertical sweep   b) Vertical retrace
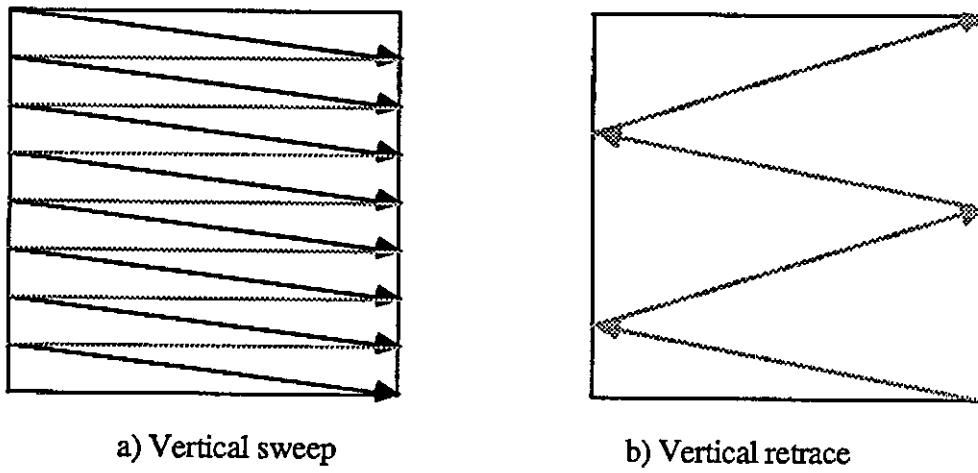
Figure 2-4. Electronic beam path in one frame.

Note that the raster lines are slanted downward while the horizontal retrace lines are nearly horizontal during a vertical scan. This is because the duration for a raster line is about 4 times longer than the horizontal retrace period. The frame rate ranges from 40 to 70 Hz, and even higher on some. The duration of the vertical sync pulse is much longer than the horizontal sync pulse to facilitate generating a composite sync signal [CONR 85].

2.3   Positioning of the Frame on Screen

To position the visible portion of the screen near the center, not all of the pixels in all of the raster lines are used. Rather, the first and the last few lines in each frame, and the first and last few pixels in each line, are always turned off to form the boundaries of the display area. The display area can be moved about the screen somewhat by changing the size of these borders. The names and positions of the borders are shown in Figure 2-5 [NEC 82a].
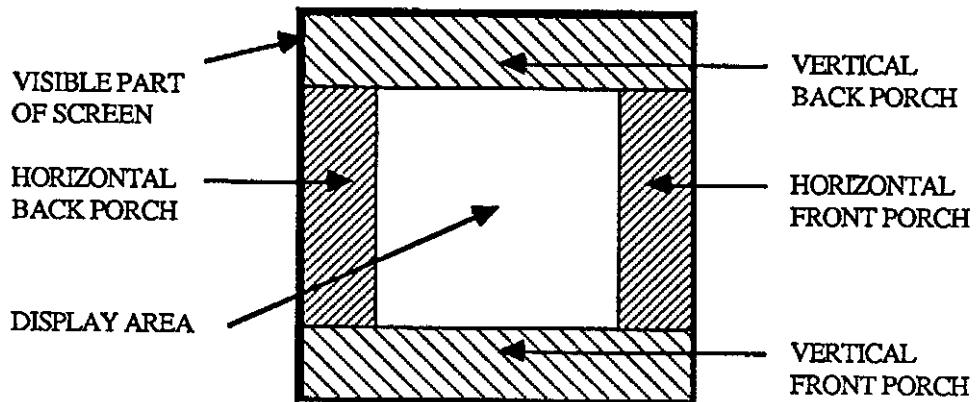
Figure 2-5. Invisible borders around the display area.

To specify the borders, the display processor generates a **blank** signal, which indicates that the beam is outside of the display area. Although it is another synchronization signal, the blank signal is not supplied separately to the monitor . Instead, it is used to mask the video information, resulting in blanked screen outside of the display area. The positions of the horizontal porches in relation to the blank signal are shown in Figure 2-6. Note that the horizontal sync pulse is shown with reverse polarity.
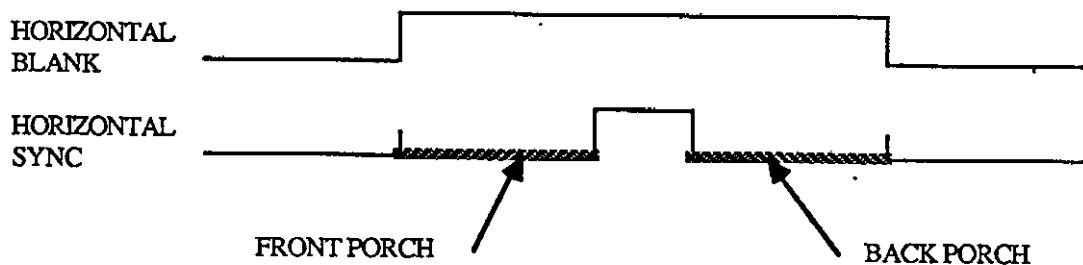
Figure 2-6. Components in a horizontal blank period.

The nominal period of the horizontal blank signal is about 11 micro-seconds. The vertical blank period is similarly organized; the vertical sync pulse is preceded by the front porch and followed by the back porch.

2.4   Interlacing

The luminescence produced by the bombarding electrons on the phosphors on the screen decays quite rapidly, requiring periodic refresh of the screen for a flicker-free picture. To reduce the refresh rate, slow-decaying phosphors are used on low cost monitors. The most commonly used slow-phosphor is the P-39, which gives off a greenish color and has a decay time of 520 milli-seconds. The relatively long decay time of the P-39 phosphor reduces the refresh rate requirement to about 30 Hz., but it also causes a smearing of moving images known as "ghosting" [CHAM 85].

To increase the vertical resolution of low quality monitors, interlacing is often used. With interlacing, two vertical scans are required to display a frame; odd lines in one sweep and even lines in another sweep. In this way, the actual vertical scan rate is not changed, but the effective frame rate is halved so twice as many lines can be displayed without the increase in vertical bandwidth. To achieve proper interlacing, two things must happen. One is that the information on every other line must be sent to the monitor in any one field. The other is that the vertical sync signal for even fields must be delayed by about one-half of a line to properly align two fields. Without proper alignment of odd and even fields, a phenomenon known as "line pairing" can be observed, where lines appear to be paired off because of non-uniform spacing between odd and even lines. This is illustrated in Figure 2-7.

Figure 2-7. Line pairing phenomenon.

2.5 Display Memory

The image seen on the screen is first created on an intermediate storage known as display memory. The design of display memory is critical to the efficiency of the entire system, for this is often the bottleneck of the system. The display memory is subjected to two processes; drawing and display, as shown in Figure 2-8.
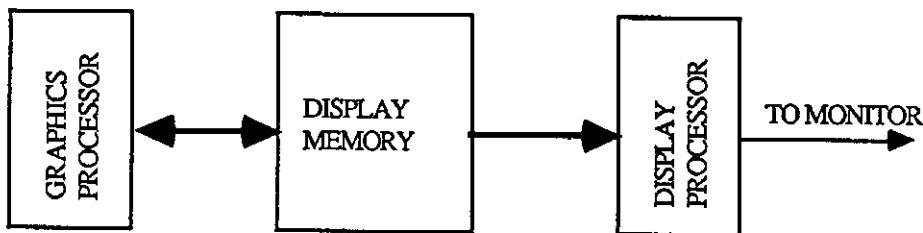
Figure 2-8. Drawing and displaying processes on display memory.

The graphics processor updates the display memory to reflect the changes in the image, while the display processor repeatedly scans the display memory and sends the video information to the monitor. The graphics processor uses the display memory only when there is a change in the image. On the other hand, the display processor constantly accesses the display memory to refresh the screen, irrespective of the changes in the image. Naturally, the display processor has higher priority over the display memory than the graphics processor to maintain a stable picture.

2.5.1 Organization

At one extreme design, the display processor accesses one pixel value at a time. This may be suitable for color or gray-scale displays where a pixel is represented by many bits. However, this severely restricts the memory access by the graphics processor because of the high frequency (at the video rate) of the access by the display processor. For most monitors, the video rate is on the order of tens of mega Hz. The display memory must have very short access time to support this design. To reduce the effective access time, an interleaved memory system would likely be used.

On the other extreme, the display processor can access the entire line at a time, and then send one pixel value at a time using shift registers. This gives plenty of time for the graphics processor to access the display memory and modify it. In addition, slower memory can be used. However, many shift registers are required to achieve this. In practice, a byte, a word, or a few words of pixel values are accessed at one time by the display processor.

2.5.2 Video RAM

To totally eliminate the memory access conflict by two processors, more expensive systems use Video RAM's such as TMS 4161. They have dual ports so that simultaneous read and write access is possible. In addition, there is a built-in shift register to serially read out a row of data at a time with a single access. For instance, TMS 4161 normally operates in 64K-by-1 random access mode. It can also operate in 256-by-256 random access mode where 256 bits can be shifted out at a 25 MHz. rate. The block diagram of TMS 4161 is shown in Figure 2-9.
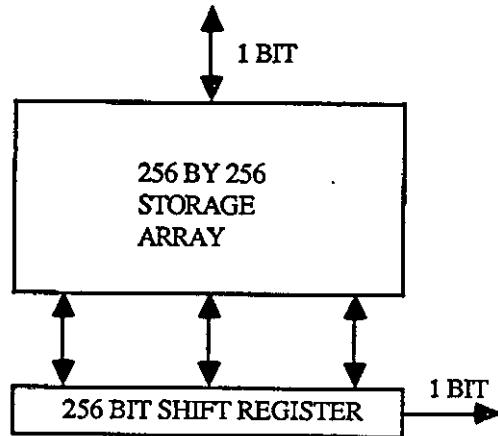
Figure 2-9. Block diagram of TMS 4161.

2.5.3 Integration of Graphics and Display Processors

In less expensive graphics systems, both the drawing and display processes are done by a single processor. The GDC and Hitachi's 63484 are such processors. These control every phase of the operation of the display memory. They can draw primitive figures on the display memory, scan it, and even refresh the display memory. These chips simplify the task of designing a graphics system. However, since one processor has to do the work of two, the drawing speed of the system is greatly reduced. The drawing operation on the display memory is restricted to the blank periods only. Furthermore, if dynamic memory refresh function is enabled, a substantial portion of the blank periods is spent on performing memory refresh. The drawing operation has the lowest priority of all the operations.

2.5.4 Addressing

The display memory is logically addressed by its row and column numbers, or by X and Y coordinates. However, it is impractical to build a display memory that is

physically addressable with row and column using the commercially available memory devices. Instead, the display memory is physically organized to form a contiguous address space with the word length much smaller than the width of a line (i.e. the number of pixels per line). One logical row is, then, composed of many contiguous words similar to the way a two-dimensional array is stored in row-major order in memory, as illustrated in Figure 2-10. This way, the size of the display memory can be adjusted without any hardware modification.
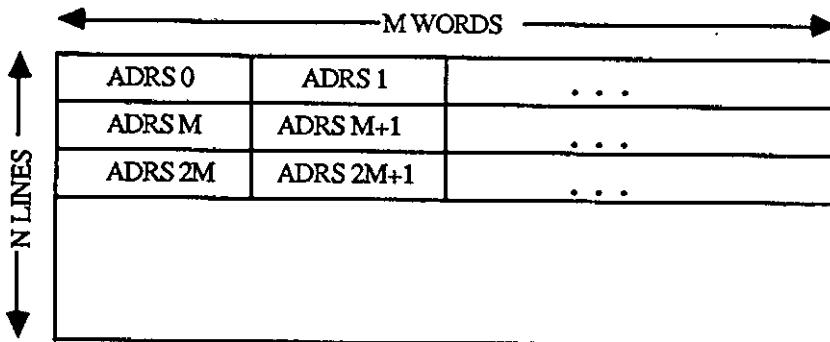
Figure 2-10. Mapping of display memory to physical memory.

If an entire word is made to represent one pixel, it is relatively easy to calculate the physical address of a pixel whose logical address is (X,Y). It would be the address $Y*M + X$. The width of a word can be made to any length to represent multitude of colors or intensity. With 3 bits per word, one of 8 colors can be selected for a pixel.

The vast majority of monochrome monitors use different organization, for only one bit is all that is needed to represent a pixel. So now, one word represents many pixels. If the width of a word is S bits, the physical address of a pixel (X,Y) would be $Y*M + (X \text{ DIV } S)$. Furthermore, it would be the (X MOD S)th bit of the word. To be able to represent colors or gray-scale, planes of the memory are added, providing additional bits to each pixel. For instance, three planes can be used, one for each color,

to represent one of eight colors, as shown in Figure 2-11. Each plane of memory would reside in different memory banks. This organization is easier to construct than designing a 3-bit wide memory system. It also requires less modification to add or delete a plane.
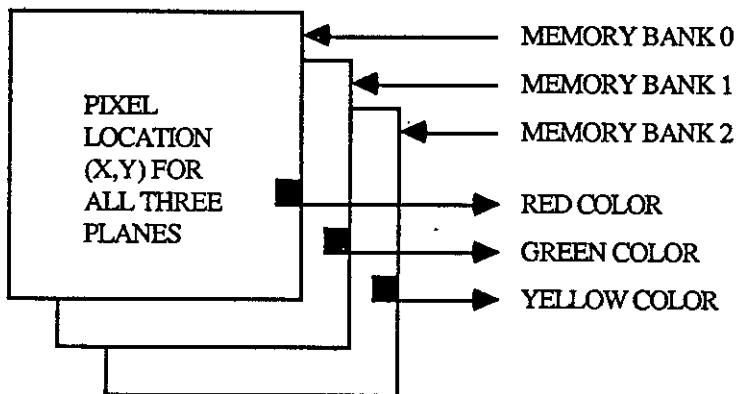
Figure 2-11. Multi-plane display memory organization.

2.6   Pipelined Process

The display cycle is a pipelined process as shown in Figure 2-12. A word of display memory is fetched in one access cycle, and it is sent pixel by pixel to the screen during the next access cycle. Thus the video rate must be equal to the access rate times the number of pixels in a word. For instance, if the display word is 16-bits wide, the video rate must be exactly 16 times the access rate to send 16 bits to the screen during one cycle. At any moment, when the display processor is accessing the N+1th word, the Nth word is being displayed on the screen.
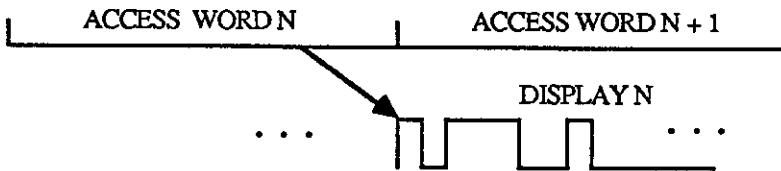
Figure 2-12. Pipelined display process.

The GDC assumes the width of the display memory to be 16 bits. It is possible to design a 32-bit wide memory system so that 32 pixels are displayed in one access cycle. Nevertheless, the memory will still be addressable on 16-bit word boundary. 16 bits would probably be the best choice for a word length, given 18 address pins for the GDC. By multiplexing the data and address bus, additional pins are not needed for the data.

2.7 Coded Character Generation

In bit-mapped graphics, whatever is in the display memory is displayed exactly on the screen. Not only figures, but characters of any size or shape can also be displayed using bit-mapped graphics. However, bit-mapped graphics characters take up much space and a long time to change; a character of 8 by 8 pixels would occupy at least eight words (one in each line), and all eight words must be modified to edit a character. An alternative and more efficient way to generate characters is to use coded characters. Instead of interpreting the contents of display memory as bit-mapped data, it is treated as codes with which characters are generated from a look-up table. This process is illustrated in Figure 2-13.
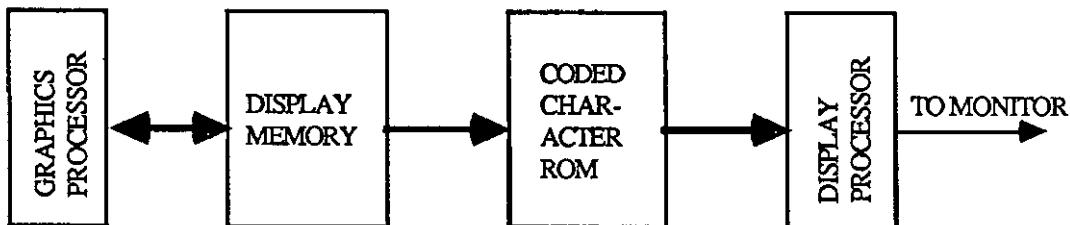
Figure 2-13. Coded character generation from display memory.

With coded characters, the size of the display memory needed to represent a block of text is independent of the font or the size of the characters. Furthermore, the video rate does not have to be equal to the width of the display word times the access rate. An 8-bit wide word can be used to generate a 13-bit wide character, if the video rate is 13 times the access rate. However, to use mix of graphics and coded characters in one frame, it is necessary to set the width of the coded characters equal to the width of the display memory. Otherwise, different pixel clocks must be used for graphics and coded character displays, with the clock switching occurring during the blank period.

To generate a coded character, two memory accesses are required: first, for the display memory and second, for the coded character generator rom. If the combined access time of the memories is too long, the display process may need to be delayed by more than one access cycle. This is shown in Figure 2-14.
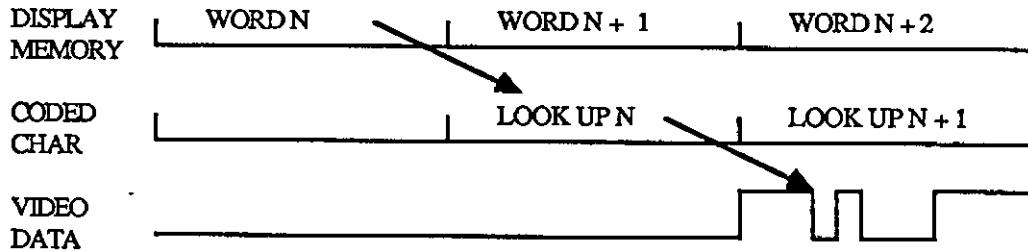
Figure 2-14. Pipelined process of coded character generation.

Since the coded character look-up table is almost always implemented in an external memory, the display processor must scan each line the number of times that is equal to the height of the character font. For instance, if the character font is 10 pixels tall, each line of the display memory must be scanned 10 times. In addition, the display processor must issue the line count, i.e. the number of times that a particular word is being repeated, so the proper row of the look-up table can be read out. Instead of the line count, the GDC issues the line-counter clear pulse for the external counter. A block diagram of coded character generator is shown in Figure 2-15.
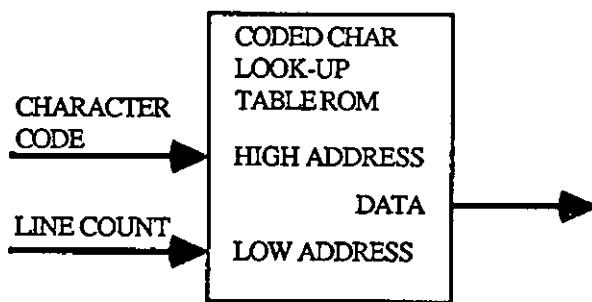
Figure 2-15. Coded character generator

On a 16K-by-8 rom, 2K of 8-by-8 or 1K of 8-by-16 size characters can be coded. Note that the line count controls the low address lines. This organization offers

several advantages over the organization where the line count controls higher address lines. Since a block of 8 (for 8-by-8) or 16 (for 8-by-16) contiguous rows are reserved for a character, it is easy to design, modify the characters and program the table. It is also possible to have different size characters in a rom. For example, an 8-by-16 block can contain an 8-by-9 or an 8-by-15 sized character. If the line counts are connected to higher address lines, the pattern for a character would be distributed all over the address ranges in the rom. This makes character designing and maintenance of the look-up table very difficult.

2.8 Capability of the GDC

In this section, a brief description of the GDC is presented. The detailed description of it is found in the data sheet and the user manual [NEC 85b].

During the visible raster periods of a frame (in other words, active periods), the GDC operates as a display processor. During the blank periods, it operates as a graphics processor and updates the display memory. Normally, a blank period is composed of front porch, sync, and back porch of horizontal and vertical sweeps. If programmed so, the GDC will use the active periods to modify the display memory, instead of scanning it. This will temporarily display unpredictable patterns on the screen, but it will increase the drawing speed considerably. Or, the entire screen can be blanked to hide the disturbances until the figure drawing operation is finished.

2.8.1 Display Cycle

In a display cycle, the GDC generates the address of the pixels that are to be displayed on the screen. Each display cycle takes two clocks (This is not always so). The Address Latch Enable (ALE) signal denotes the beginning of a display cycle, as well as the period when the address is valid on the multiplexed data/address i/o pins.

The timing diagram of the signals the GDC generates in a display cycle is shown in Figure 2-16.
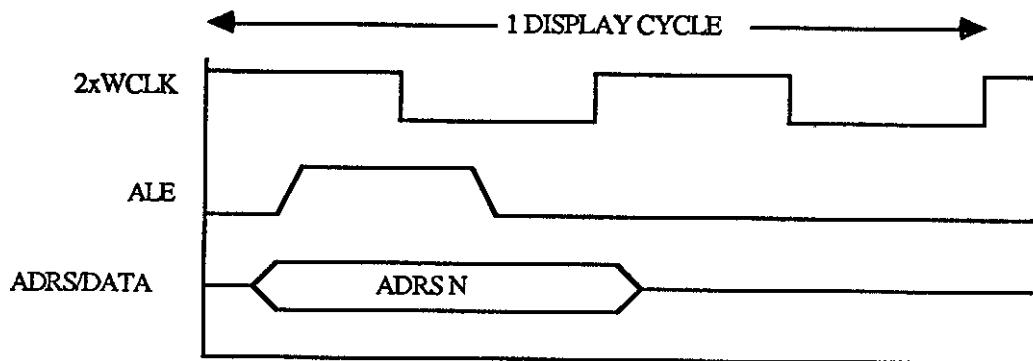
Figure 2-16. Display cycle timing.

Since the display memory is 16 bits wide, 16 pixels must be displayed in one display cycle. For this, the video rate must be 8 times the clock (2xWCLK) rate of the GDC.

2.8.2 Read-Modify-Write Cycle

In this cycle, the GDC is drawing on the display memory. A RMW cycle takes four clocks. Again, the ALE signal indicates the beginning of a RMW cycle. In fact, until the middle of the second clock, there is no way to distinguish between a display cycle and a RMW cycle. During the later half of the second clock, the Data Bus Input Enable (DBIN) signal goes low for a clock period to indicate that a RMW cycle is in progress and that data is expected from the display memory. After the data is input at the falling edge of the third clock, the modified data is output during the fourth clock. The timing diagram is shown in Figure 2-17.
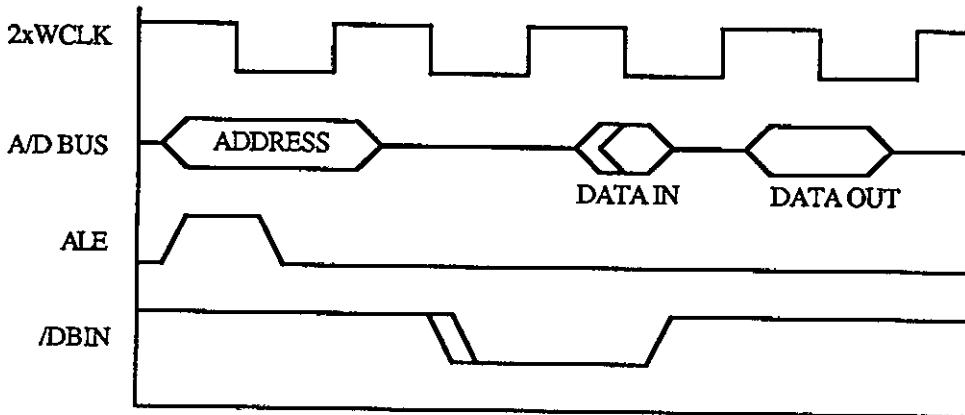
Figure 2-17. Read-Modify-Write cycle timing.

2.8.3 Dynamic Memory Refresh

The nominal frame rate for a raster display device is 40 to 70 Hz. This means that the frame buffer, a portion of the display memory that contains information for one screen, is scanned once each 14 to 25 milli-seconds. This is far too long for dynamic memory to sustain its data without periodic refresh. To simplify the display memory system design, the GDC has built-in refresh circuitry for dynamic memory. During the horizontal sync periods (HS), the GDC generates the addresses of the display memory to be refreshed using an internal refresh counter. Since almost all dynamic memory chips require refresh of 128 rows every 2 milli-seconds, one row must be refreshed every 15 micro-seconds on the average (2 milli-seconds / 128 rows). The HS must be long enough so that all 128 rows can be refreshed at least once in 2 milli-seconds. During a HS, the content of the refresh counter is output on the address bus. In order to generate successive row addresses, the lower address lines should be fed to the memory system as row addresses.

2.8.4 Mixed Graphics and Coded Character Display

The GDC can operate in one of three modes; graphics only, coded character only, and mixed graphics and coded character. In mixed mode, both graphics and coded characters can be displayed on one screen.

During each horizontal blank period, A17 indicates whether the upcoming raster line should be interpreted as bit-mapped or coded character data. To eliminate the need of clock switching between the graphics and coded character data, the width of a coded character is assumed to be 8 pixels so that one display cycle is 8 pixels long. With graphical data, one display cycle is not long enough (it is one-half of the needed) to display 16 pixels. So the GDC scans each word of display memory twice for graphics data to provide more time to display the bit-mapped image. The display cycle timing in mixed mode is shown in Figure 2-18.
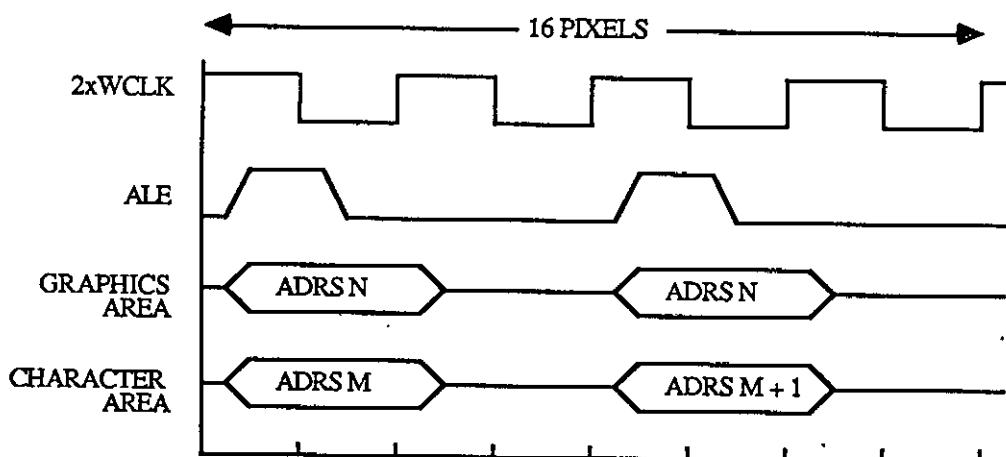
Figure 2-18. Display cycle timing in mixed mode.

With coded character data, a display cycle still takes two clocks. Note that with graphics data, every other access of the display memory must be suppressed. One added advantage of using the GDC in mixed mode is that it can be operated at twice the

clock rate for graphics-only mode without increasing the video rate. In graphics-only mode, if the video rate is 16 MHz., the clock rate is 2 MHz. In mixed mode, a 4 MHz. clock should be used to maintain the same video rate. Since a RMW cycle still takes four clocks, the drawing speed is doubled. Doubling the clock rate of the GDC doubles the speed of DMA operations as well.

2.8.5 Character Cursor

In character area, A16 and A17 supply the position and blinking rate information of the character cursor. During active periods, A17 is asserted high to indicate the cursor position. The A16 is toggled to indicate the blinking rate of the cursor. The blinking rate is programmable.

2.9 Video Timing Calculation

Since the GDC is fully capable of generating all of the necessary control signals for the monitor, it is possible to use it with a variety of resolutions without any modification on the support circuitry. For the benefit of those who want to use the GDC board with a different monitor or different resolution, the detailed video timing calculation used in the project is presented here.

To use the GDC in interlaced, mixed mode with DMA and zoom capabilities, the following must be satisfied:

1. Horizontal Back Porch (HBP) $\geq$ 5 words,
2. Horizontal Sync (HS) $\geq$ 5 words,
3. Horizontal Front Porch (HFP) $\geq$ 3 words,
4. Vertical Back Porch (VBP), Vertical Sync (VS), and Vertical Front Porch (VFP) must each be, at least, greater than the duration of a line.

In addition, the monitor imposes the following constraints:

1. Video Pulse Width ≥ 45 nano-seconds,

2. 15 KHz. ≤ Horizontal Sync ≤ 16.5 KHz.,

3. 47 Hz. ≤ Vertical Sync ≤ 63 Hz.,

4. Minimum Horizontal Retrace Time ≥ 9 micro-seconds,

5. 300 micro-seconds ≤ Vertical Retrace Time ≤ 1400 micro-seconds.

The process of calculating video timing is somewhat arbitrary and involves many trial-and-errors. There are more design parameters than the number of equations, and it simplifies the problem if a few assumptions are made. First, the horizontal resolution is assumed to be 800 pixels, or 50 words. Second, the video rate is assumed to be 16 MHz. so that the video pulse width is 62.5 nano-seconds. This makes the period of a line to be

(50 + 5 + 5 + 3) words * 1 micro-seconds/word = 63 micro-seconds.

The line rate is 15.9 KHz. The period of the active portion of a line is 50 micro-seconds, and that of the blank period is 13 micro-seconds. With 60 Hz. vertical sweep, there can be

(1/60 seconds/vertical sweep) / 63 micro-seconds/line = 264.5 lines/vertical sweep

To minimize the blank portion of the display area, assume the vertical retrace period to be about 300 micro-seconds, or 5 lines. Since there are no restrictions on the length of the vertical blank period, let VBP = 1, VS = 3, and VFP = 1 line. Out of 265 lines, 5 lines are to be set aside for the blank period. With interlacing, there can be at most 521 (260 * 2 + 1) lines per frame, each frame consisting of two vertical sweeps. The vertical blank portion is 315 (5 * 63) micro-seconds. The resolution of the system is 800 by 521. The exact design parameters are re-calculated and shown in appendix C.

The horizontal sync pulse must be long enough so that all the display memory can be refreshed properly. With 64K dynamic RAM, each of 128 rows must be refreshed once every 2 milli-seconds, which means that there must be at least 128 memory accesses during all the HS periods in 2 milli-seconds. There are

(2 milli-seconds) / (63 micro-seconds/HS) * (5 words/HS) = 158 word accesses.

during the HS periods in 2 milli-seconds.

Chapter 3

HARDWARE DESCRIPTION

This section contains the detailed hardware description of the GDC board. The board was initially designed to interface to a processor node in the "Look Ahead" network and function as a server [BALA 83]. This somewhat unusual configuration was called for to test the robustness of the network. However, with the processor in the Look Ahead network being upgraded to the 68000 family of microprocessors, and without a reliable processor running the OS-9 operating system, the board was modified to work with a TRS-80 Color Computer, which is also a 6809 based computer [AHRE 81]. The hardware described here reflects these changes, as well as additions, made to the board. The apparent inconsistencies in design logic is due to these modifications. A simple block diagram of the system is shown in Figure 3-1.
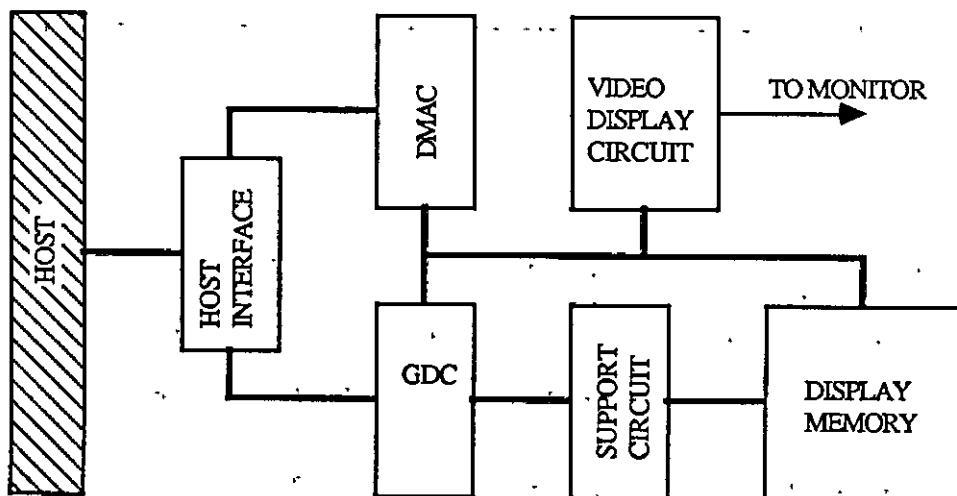
Figure 3-1. A block diagram of the GDC board.

27

3.1 Host Interface

Through this interface, a host computer is in complete control over the operations of the GDC board. Since the interface is implemented with a Programmable Array Logic (PAL) and a few TTL IC chips, it was easy to modify the interface to work with the Color Computer. The modified interface is described below, and the block diagram is shown in Figure 3-2.
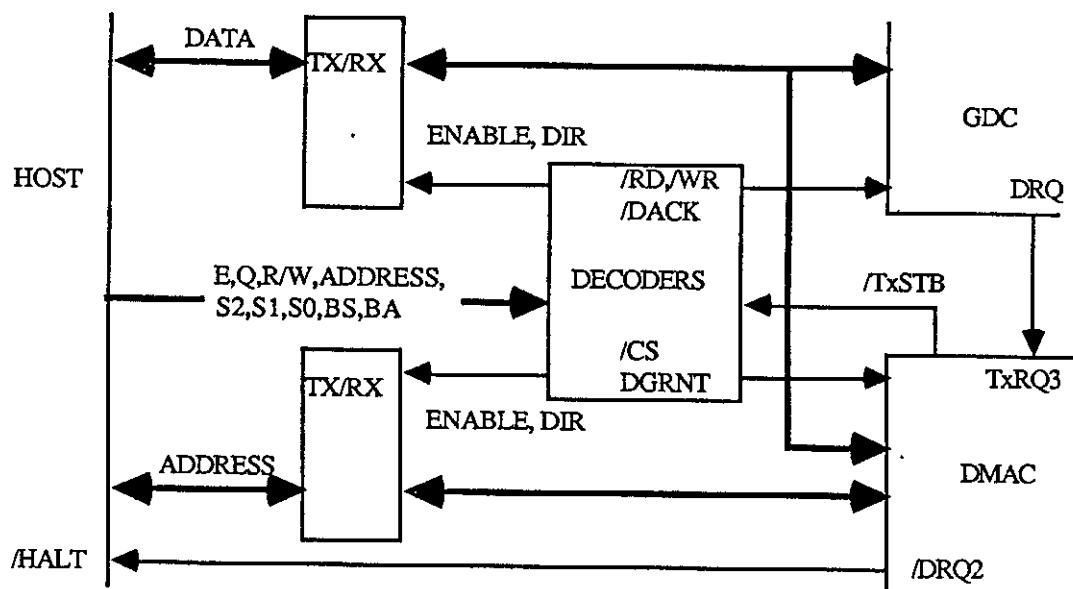
Figure 3-2. Block diagram of the host interface.

3.1.1 Address Space

There are 18 programmable registers on the board, occupying 26 locations. The addresses $FF60 to $FF7A were assigned to them instead of the designated i/o addresses ($FF40-$FF5F) for the following reasons:

1. The DMAC requires 24 contiguous addresses starting at 32-byte boundary,

2. The dual-floppy disk controller is located in the first few addresses of the designated i/o space,

3. The modification needed was simpler than other solutions,

4. It seemed unlikely that the reserved address spaces $FF60 to $FFBF would be used in the future.

An alternative solution is to feed the inverted address lines to the register select pins of the DMAC so that it would respond to addresses $FF5F to $FF48, instead of $FF40 to $FF57. This solution was discarded because of the need to cut numerous traces that are routed on inner layers of the board. Another alternative was to enable/disable the chip-select decoder (IC16, 74ls138) of the Color Computer with a programmable register. Before pending i/o from the GDC board, this register can be set to disable the decoder by asserting low on pin 40 of the expansion port when $S = 6$ is detected. The SAM chip sets S to 6 ($S_2 = 1$, $S_1 = 1$, $S_0 = 0$) for the addresses $FF20 to $FF3F where the second PIA is. This will free these addresses temporarily for other peripherals. This solution was also discarded because the enable/disable control register needs a full decoder to be recognized. The addresses and functions of the registers on the board are listed in Table 3-1.

3.1.2 Board Select Logic

The BOARD_SEL signal represents the addresses $FF60 to $FF7F, and is derived from the following equation:

$$\text{BOARD\_SEL} = S_2 * S_1 * S_0 * A_{15} * A_7{'} * A_6 * A_5$$

ADDRESS	READ	WRITE	DEVICE
FF60,1	system bus address register		DMAC channel 0
FF62,3	byte count register		DMAC channel 0
FF64,5	system bus address register		DMAC channel 1
FF66,7	byte count register		DMAC channel 1
FF68,9	system bus address register		DMAC channel 2
FF6A,B	byte count register		DMAC channel 2
FF6C,D	system bus address register		DMAC channel 3
FF6E,F	byte count register		DMAC channel 3
FF70	channel control register		DMAC channel 0
FF71	channel control register		DMAC channel 1
FF72	channel control register		DMAC channel 2
FF73	channel control register		DMAC channel 3
FF74	priority control register		DMAC
FF75	interrupt control register		DMAC
FF76	data chain register		DMAC
FF78	status reg.	parameter	GDC
FF79	FIFO	command	GDC
FF7A	N/A	zoom	ZOOM prescaler

Table 3-1. Addresses of the programmable registers on the GDC board.

3.1.3 Non-DMA Read/Write Logic to GDC and Zoom Pre-scaler

The host computer controls the read and write operations to the GDC with the /RD and /WR signals. The equations for the read and write enables and zoom pre-scaler register, whose function will be explained shortly, are:

$$\text{/RD} = \text{BOARD\_SEL} * A_4 * A_3 * A_2' * A_1' * E * R/W$$

$$\text{/WR} = \text{BOARD\_SEL} * A_4 * A_3 * A_2' * A_1' * E * R/W'$$

$$\text{/CSZOOM} = \text{BOARD\_SEL} * A_4 * A_3 * A_2' * A_1 * A_0' * E * Q * R/W'$$

Note that "/" is used to indicate negative assertion of a signal and a quote, ', is used to indicate negation. The timing diagram is shown in Figure 3-3. The shaded regions represent the timing requirements for reading from GDC and reading from the Color Computer, which are satisfied.
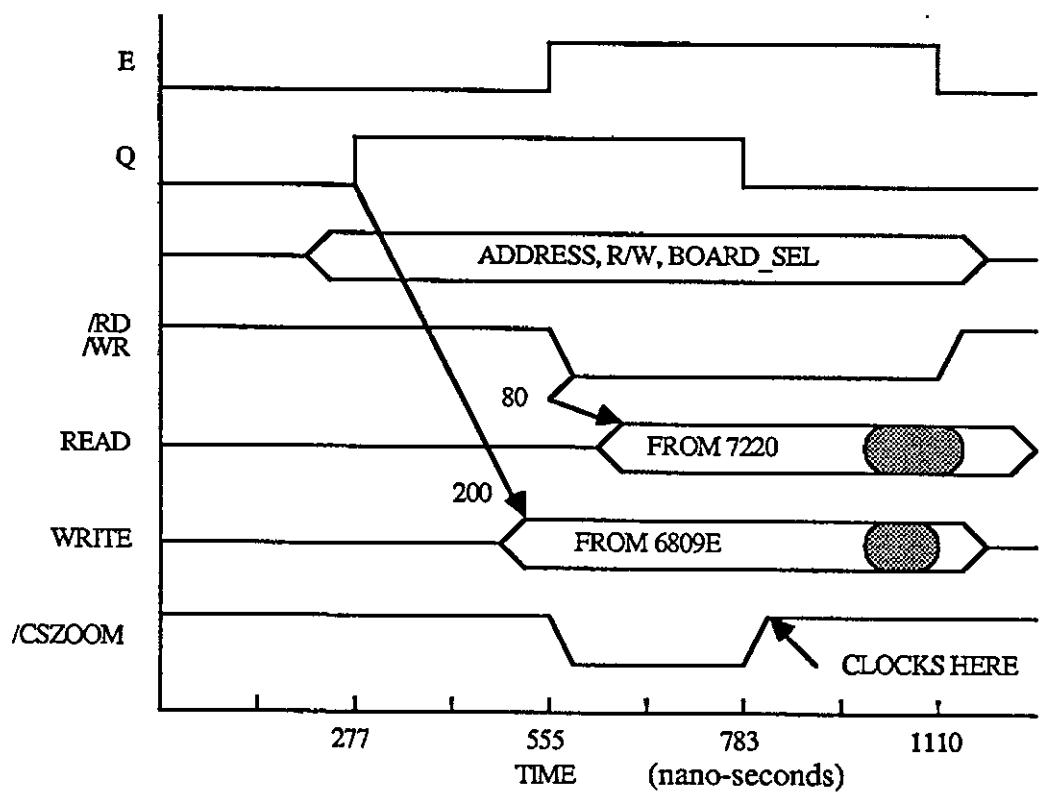
Figure 3-3. Host interface timing diagram.

3.1.4 DMAC Slave Mode Read/Write Logic

Since MC6844 is a member of the 6809 family, the interfacing is simple; only the chip select needs decoding. The equation is:

$$\text{/CSDMAC} = \text{BOARD\_SEL} * A_4' + \text{BOARD\_SEL} * A_4 * A_3'$$

3.1.5 Address and Data Bus Interface

The address and data on the GDC board are connected to the host computer through bus transceivers. These transceivers are enabled only when the board is selected, indicated by BOARD_SEL, or when DMA operation is taking place. For all non-DMA cycles, the direction of the address transceivers is from the host to the board. In DMA cycles, the direction is reversed. For the data bus, the direction is from the board to the host for either non-DMA read or DMA write (to GDC) cycles. The direction is reversed for non-DMA write and DMA read cycles. These are represented in equations as:

$$\text{/ENABLE (for all)} = (\text{BOARD\_SEL} + \text{DGRNT})'$$

$$\text{ADRS\_DIR} = \text{DGRNT}'$$

$$\text{DATA\_DIR} = (\text{DGRNT} * \text{R/W} + \text{DGRNT}' * \text{R/W}')'$$

3.1.6 DMA Read/Write Cycle

In DMA cycles, the DMAC is in control of generating the address and the R/W signals. For the fastest transfer rate, the halt-burst mode of the DMAC is used, where data transfer is continued until completion once initiated. Since the GDC can transfer one byte of data every four clocks (at 4 MHz.), and the DMAC can transfer one byte every clock (at 1 MHz.), the DMA transfer will be able to sustain its peak rate. The sequence of operations that take place in a DMA transfer is described below:

1. Both the GDC and the DMAC are programmed for DMA transfer.

2. The GDC requests data transfer by asserting the DRQ to the DMAC through the TxRQ3 signal.

3. The DMAC requests that the host computer be halted by asserting low on the HALT signal.

4. The host computer finishes the current instruction, releases the control over the address bus and R/W signals. The signals BA and BS indicate that the bus is free on the rising edge of the Q clock. This generates the DGRNT signal.

5. Upon receiving the DGRNT, if the DRQ is still asserted, the DMAC acknowledges the transfer request by asserting the TxSTB low.

6. Now actual data transfer can take place at the second half of every E clock until finished.

The equations for the pertinent signals in DMA transfers are listed below:

/DACK   =   (/TxSTB' * Q + /TxSTB' * E)'

/RD   =   (/DACK' * R/W')'

/WR   =   (/DACK' * R/W)'

The complete equations for /RD and /WR signals are shown in the Appendix B. The DMA read and write cycle timing diagram is shown in Figure 3-4. The DMA request and grant handshake signals are assumed to be already established, and are not shown for clarity. The detailed hardware diagram is presented in Appendix A.
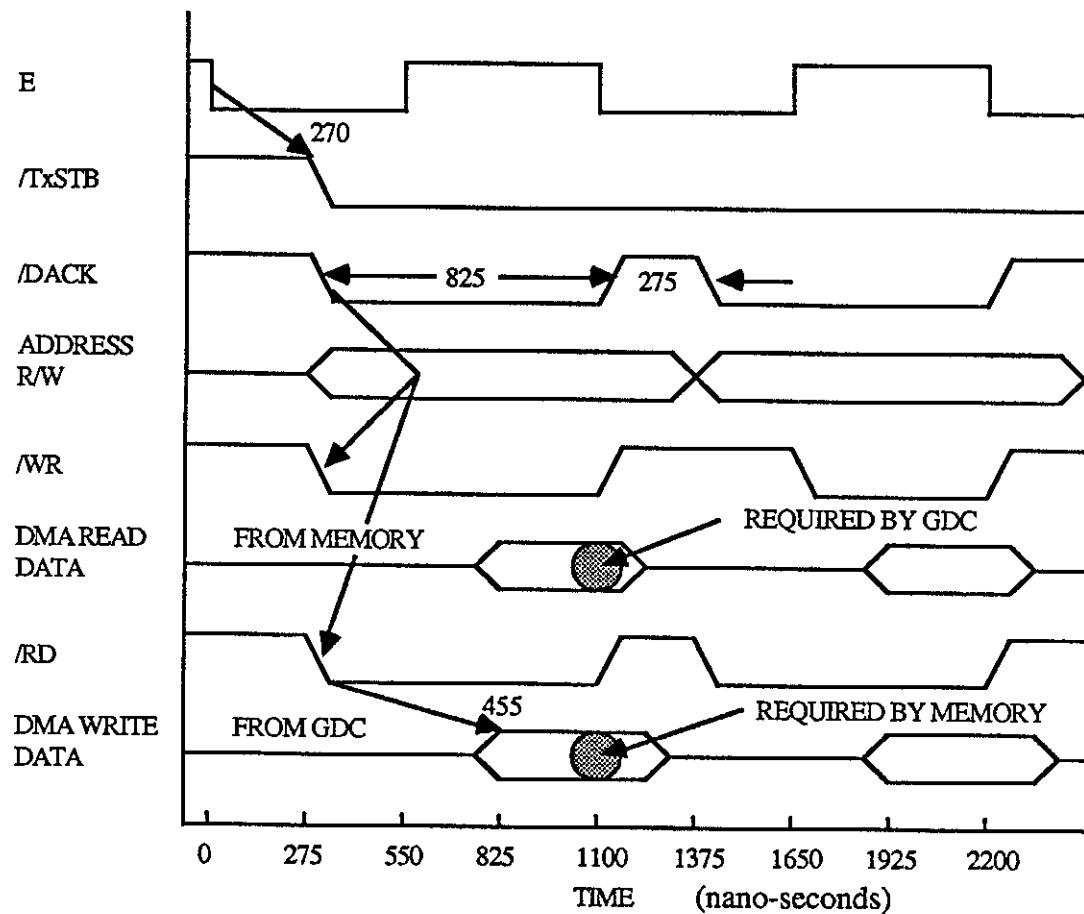
Figure 3-4. DMA read and write cycle timing diagram.

3.2 GDC to Display Memory Interface

The GDC has complete control over the display memory at every phase of its operation. It does so with 20 pins: Address Latch Enable (ALE), Data Bus Input Enable (/DBIN), A17, A16, and a multiplexed address/data bus of 16 bits wide. Through these pins, the GDC provides the necessary signals to scan the display memory, in addition to modifying it. With additional circuitry, the GDC functions as both the display and graphics processor.

3.2.1 2xWCLK Clock Generation

Since the video rate is 16 times, and the clock rate of GDC is 4 times the display memory access rate, the clock for the GDC can be easily derived from the pixel clock. A shift register is used to divide the pixel clock by 4. In fact, this approach is preferred to using two separate clocks for video data and the GDC, for a single clocking source will simplify the task of coordinating the timing of the entire system. The clock input to the GDC is named 2xWCLK to signify that a normal display cycle is composed of 2 clocks. Throughout the discussion, WCLK and 2xWCLK refer to the same signal.

3.2.2 Display Memory Control Logic

It is relatively easy to generate the necessary signals to control the dynamic memories. The Row-Address-Select (RAS*), and Column-Address-Select (CAS*) are generated by stretching the ALE signal, as shown in Figure 3-5. The related timing diagram is shown in Figure 3-6. Note that since the RAS* is asserted 300 nano-seconds, and the CAS* is asserted about 240 nano-seconds before the end of an access cycle, relatively slow memory can be used with ease. It is likely that most of the 250 nano-second access time memory chips can be used in the system.
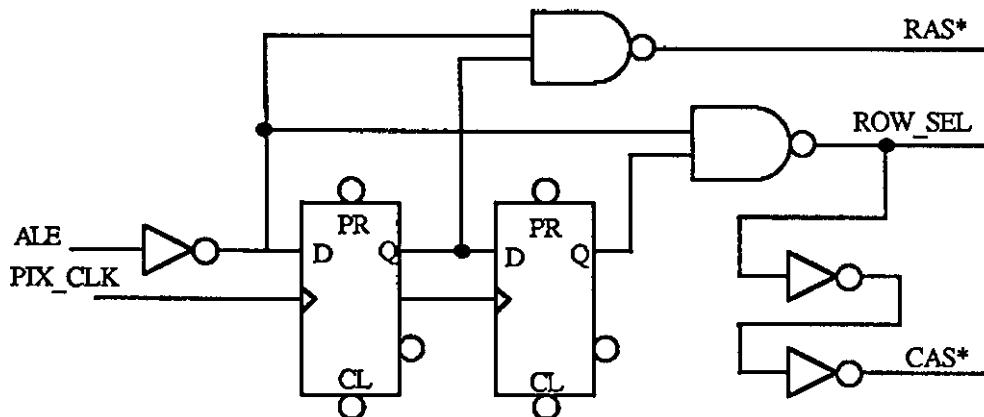
Figure 3-5. RAS* and CAS* generation.
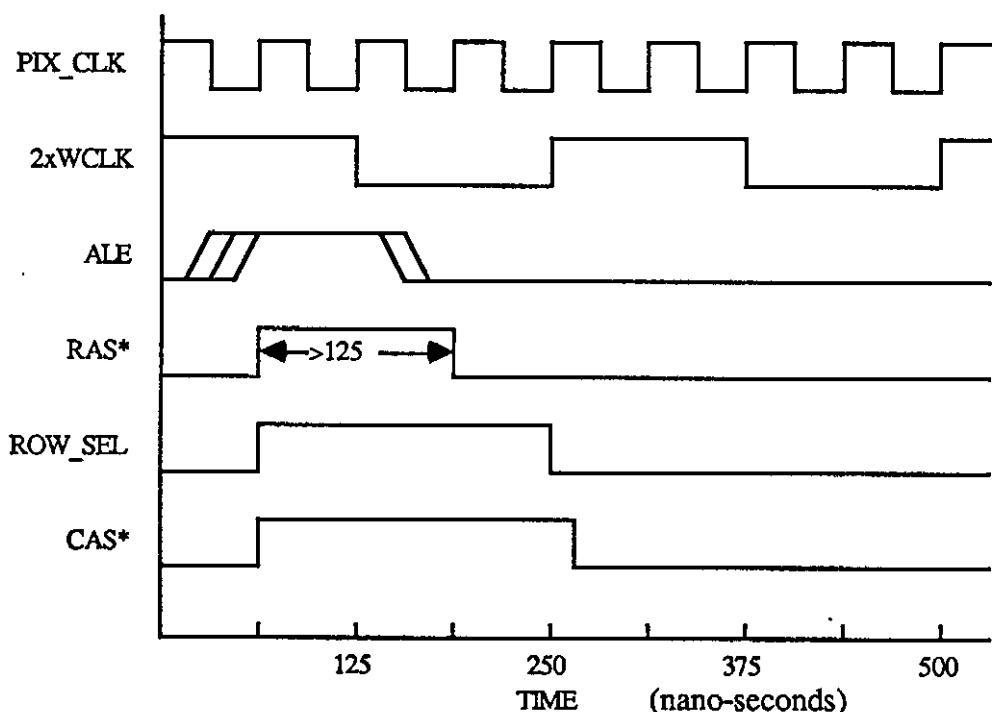
Figure 3-6. Timing diagram for display memory control signals.

3.2.3 Display Memory Write_Enable Signal

The only time the display memory is updated is when the GDC is in a RMW cycle, which is indicated by the assertion of the DBIN signal. Consequently, the Write_Enable (/WE) signal is derived by delaying the DBIN to the second half of the last clock in the RMW cycle. The logic diagram for this is shown in Figure 3-7. The timing diagram is shown in Figure 3-8.
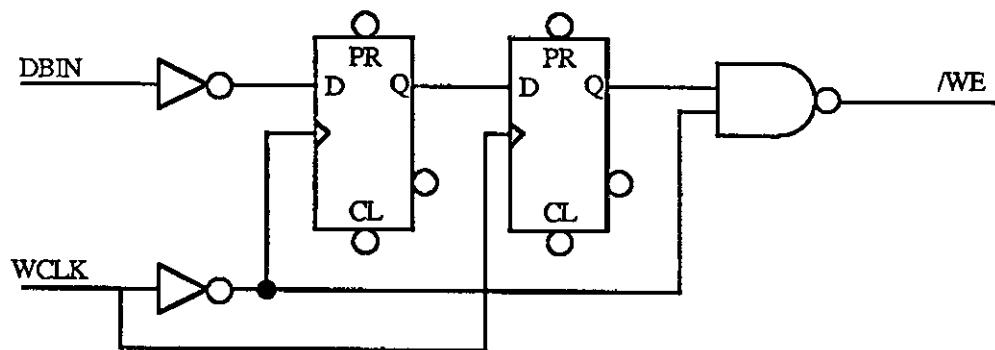
Figure 3-7. Logic diagram for the display memory Write_Enable signal generation.
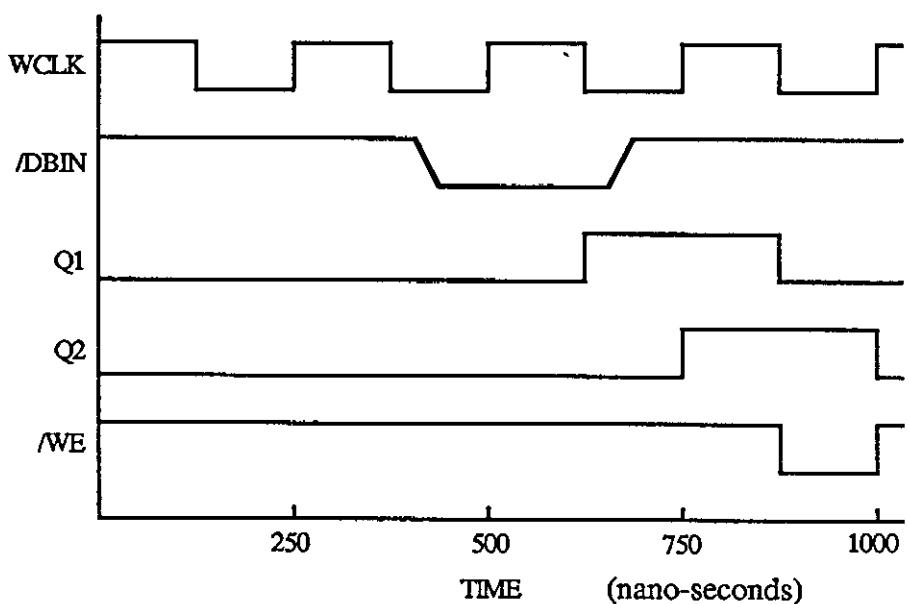
Figure 3-8. Timing diagram for the Write_Enable signal.

3.3 Display Processor Support Interface

There are three parts to the display processor support interface; one for graphics data, one for coded character data, and the other for the character cursor. The video data sent to the monitor must be selected from one of these three sources. A simplified block diagram of the support interface is shown in Figure 3-9.
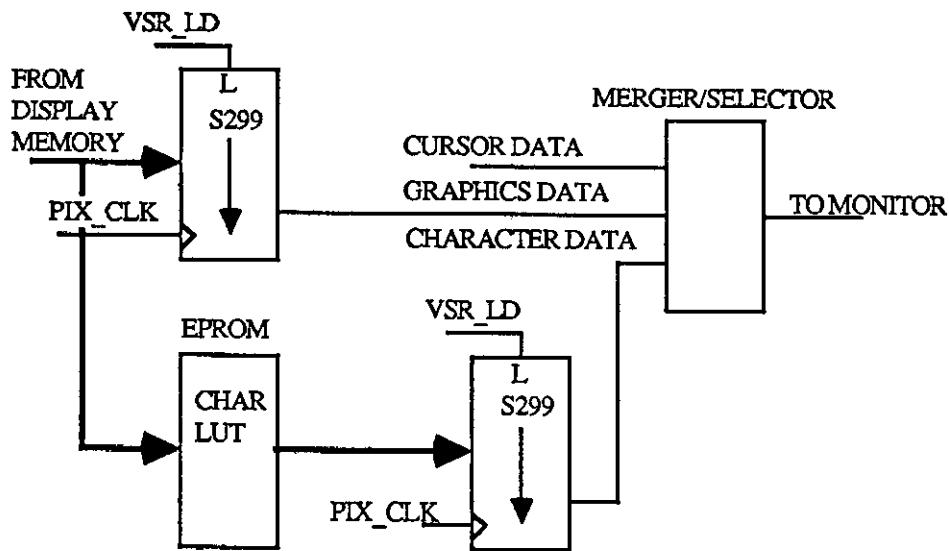
Figure 3-9. Block diagram of the video display processor

When the GDC is scanning the graphics area of the display memory, the 16-bit shift register is used to serialize the graphics data. When the coded character area is being scanned, a separate 8-bit shift register is used to serialize the character data read out from the look-up table. The cursor data is merged with the character data, while the graphics information is mutually exclusive with both. The more detailed design is presented below.

3.3.1 Video Shift Register Load Signal

The Video Shift Register Load (VSR_LD) signal must be asserted at the end of each memory access cycle so that the video information can be sent to the monitor during the next access cycle. A counter is used to create a window at the end of every second clock in which the VSR_LD may be asserted. This window is open when the count is 7 ($Q_2 = Q_1 = Q_0 = 1$). Since the display cycles may be stretched (for zoomed displays), the T10 is used to delay the window for zoomed display cycles. The logic diagram for generating VSR_LD signal is shown in Figure 3-10. Figure 3-10 also shows the logic diagram for generating the WCLK clock. In graphics or character-only modes of operation, the WCLK is generated by the output of Q_2 (divide by 8). In mixed mode, the WCLK is generated by the output of Q_1 (divide by 4). The jumper switches enable the user to select the desired clock. The timing diagram is shown in Figure 3-11.
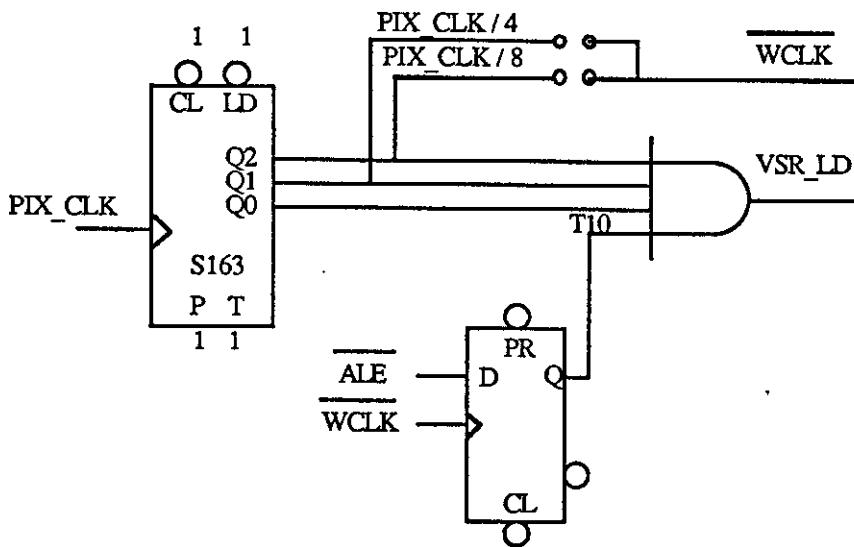
Figure 3-10. Logic diagram for generating VSR_LD and 2xWCLK signals.
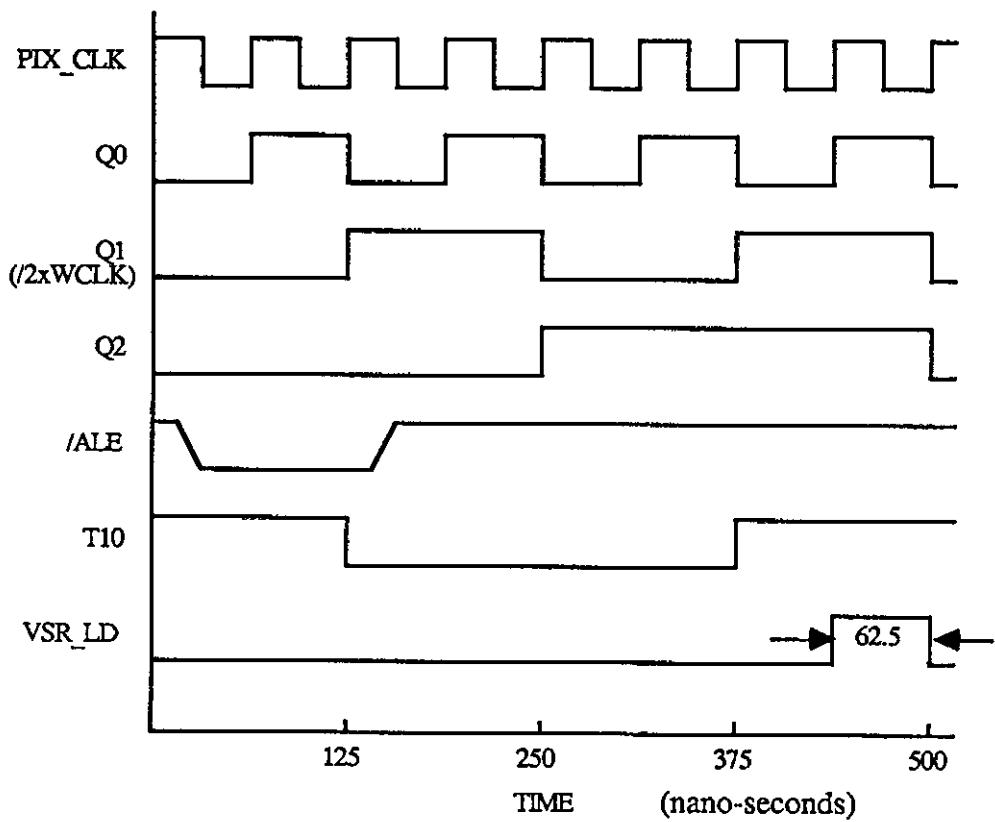
Figure 3-11. Timing diagram for the Video Shift Register Load signal.

3.3.2 Suppressing Every Other Occurrence of VSR_LD Signal

Since in mixed mode of operation, the GDC accesses each address in display memory twice for graphics area, every other access must be suppressed. One way to achieve this is to suppress every other occurrence of the VSR_LD signal. This is shown in Figure 3-12. The modified VSR_LDX2 controls only the shift registers for graphics data. The shift register used for coded character data is still controlled by the VSR_LD signal.
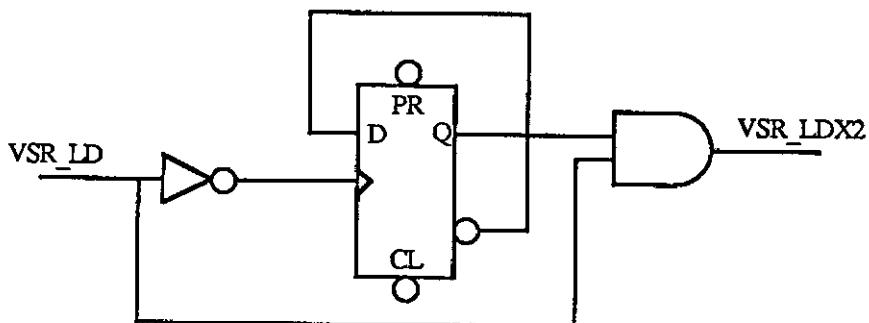
Figure 3-12. Logic diagram for suppressing every other access on graphics data.

3.3.3 Zoom Pre-scaler Logic

With an external pre-scaler, the GDC can create a zooming effect with pixel replication. The GDC will lengthen the display cycle by 2 clocks for each increase of the zoom factor. For instance, a normal display cycle takes 2 clocks to complete while a 2X zoom display cycle takes 4 clocks. A 3X zoom display cycle will take 6 clocks. In a zoomed display cycle, the ALE signal goes high shortly after the 2nd clock, as in normal display cycle. However, it does not fall until the beginning of the next display cycle. It stays high for the duration that is equal to twice of the zoom factor. The beginning of a display cycle is always indicated by the falling edge of the ALE. This is shown in Figure 3-13. Note that the assertion of the VSR_LD signal is missing at the end of the 4th clock due to the stretching of the ALE (high state) signal.
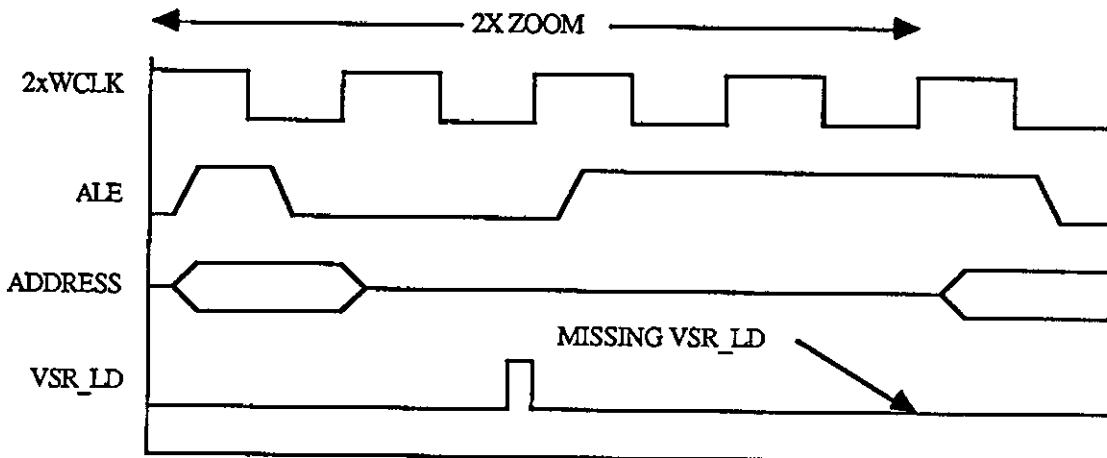
Figure 3-13. Timing diagram of a 2X display cycle.

It is the function of the pre-scaler to slow down the shift register that serially sends the pixel information by the factor that is equal to the zoom factor. This is achieved by temporarily halting the shift operation. The S_0 and S_1 input of the universal shift register (74299) can be programmed to shift left, hold, or load the data. The pre-scaler logic using this feature of 74299 is shown in Figure 3-14.
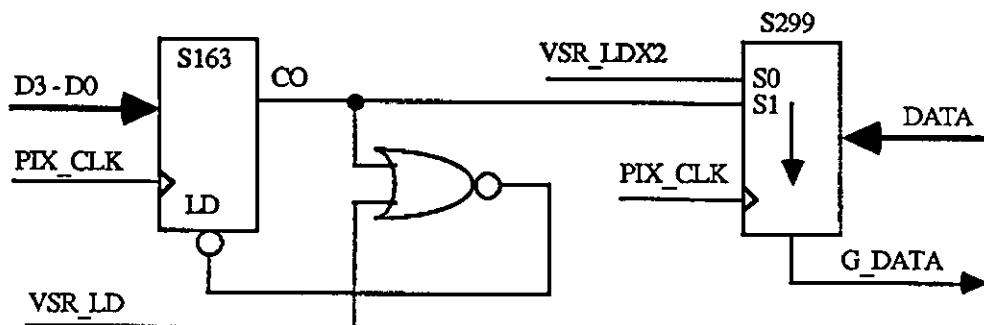
Figure 3-14. Zoom pre-scaler logic diagram.

Before programming the GDC for a zoom operation, the pre-scaler must be set to the desired zoom factor by writing the one's complement of the zoom factor to the address $FF78. The zoom factor is represented by a 4-bit binary number; 0 for normal, 1 for 2X, and F for 16X zoom. The binary counter counts from the one's complement till the carry out is generated. So for the zoom factor of 0 (no zoom), the carry out is always generated, which enables the shift operation of the shift register (S0 = L, S1 = H). A word of display memory is loaded into the shift register when VSR_LDX2 is asserted (S0 = H, S1 = H). The shift operation is halted when CO is low (S0 = L, S1 = L).

3.3.4 Display Cycle Timing for Graphical Data

The complete timing diagram of a display cycle for managing graphical data is shown in Figure 3-15. The GDC scans each word of the display memory twice, so the first access is ignored. However, since the video rate is 4 times the WCLK, four clocks are required to display the entire 16 bit of pixel information.
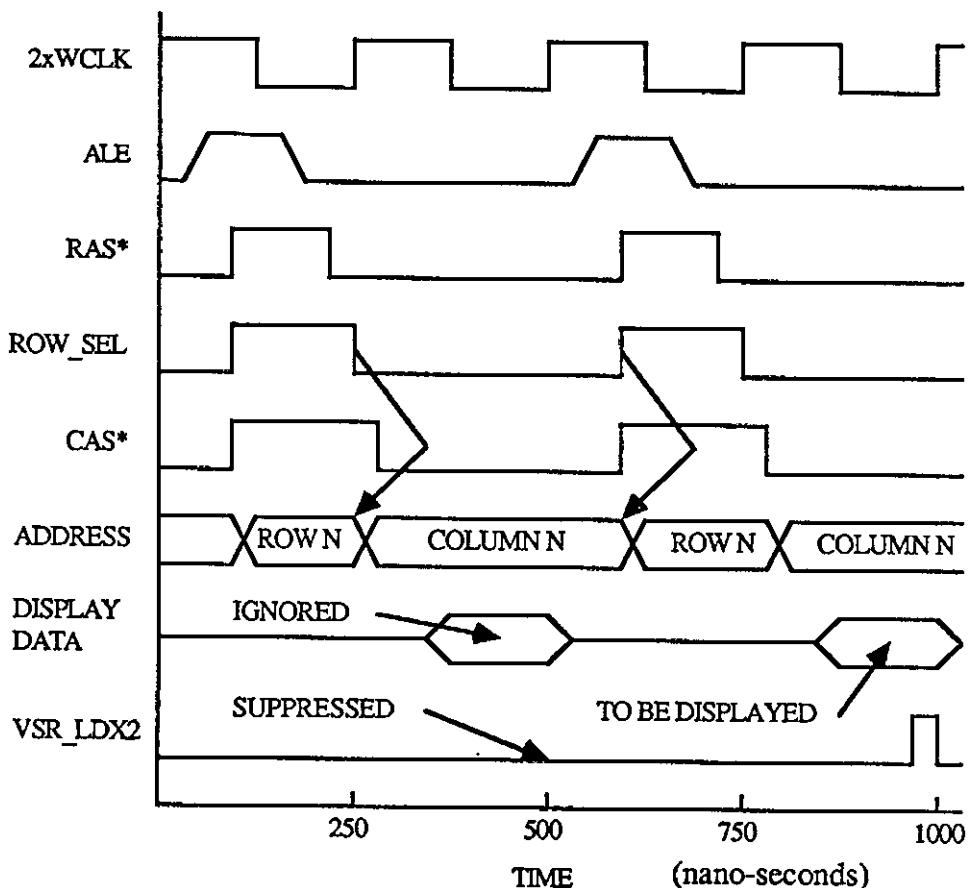
Figure 3-15. Timing diagram of a display cycle for graphics information.

3.3.5 Display Cycle Timing For Coded Character Data

The complete timing diagram of a display cycle for managing coded character data is shown in Figure 3-16. The RAS* and CAS* signals are identical to that of the graphical data, and are not shown for clarity. Note that the video information is sent to the monitor after two stages of conversion, and that each access to a word of display memory results in 8 pixels of data.
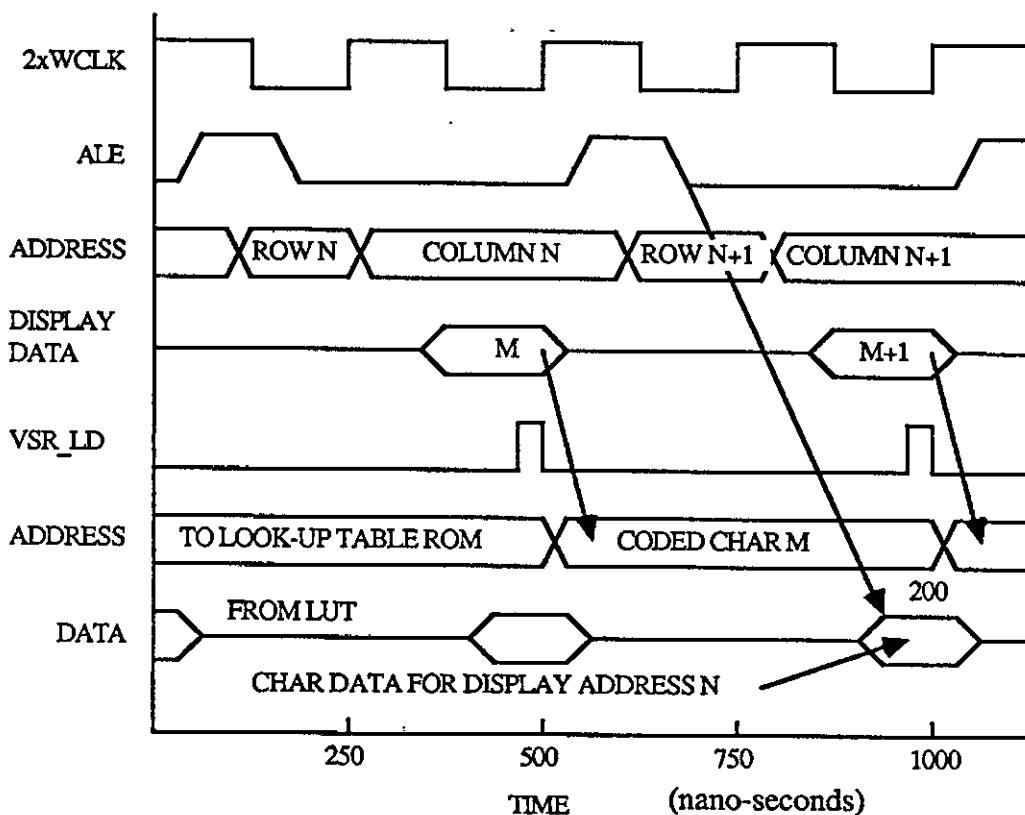
Figure 3-16. Timing diagram for a display cycle for coded character information.

3.3.6 Read-Modify-Write Timing of Display Memory

Although a display cycle in the mixed mode of operation takes twice that of the graphics-only mode of operation, a Read-Modify-Write cycle still takes four clocks. The timing diagram for a RMW cycle is shown in Figure 3-17.
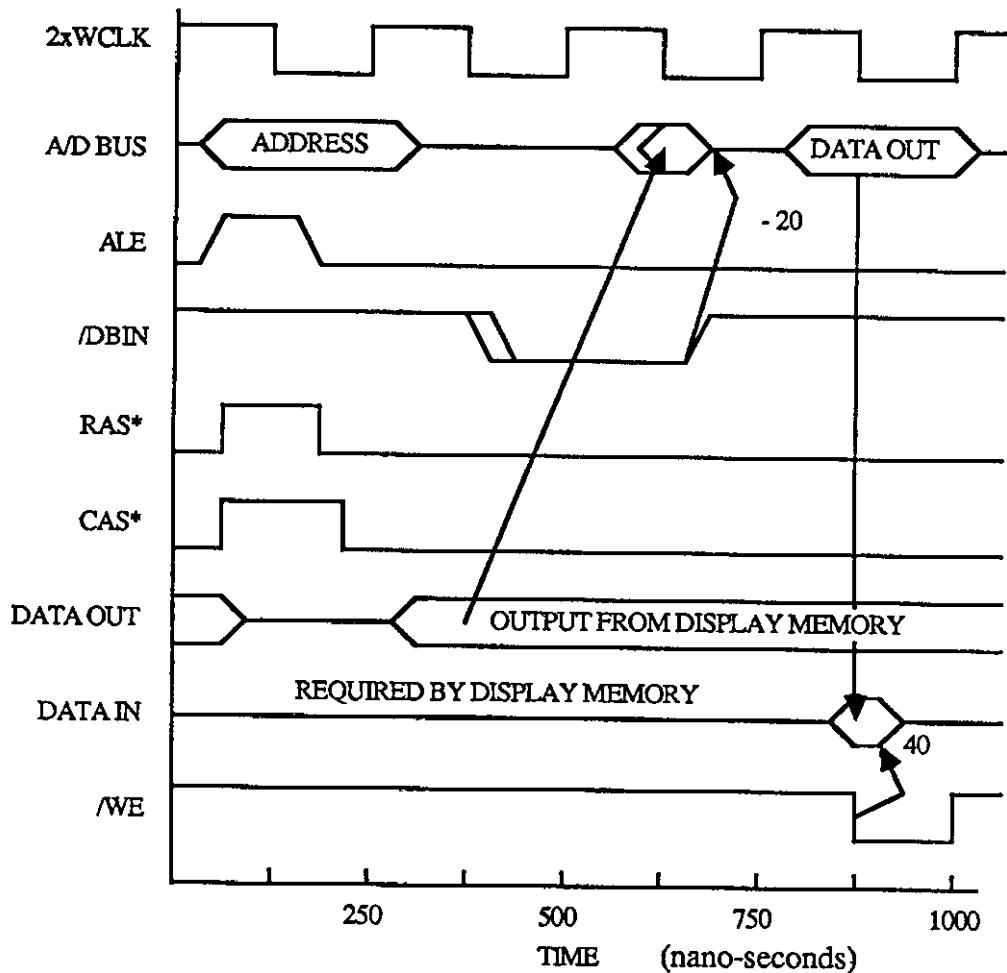
Figure 3-17. Timing diagram of a Read-Modify-Write cycle.

3.4 Multiplexed Signals of A16 and A17

The pins A16 and A17 carry multiplexed signals to the outside of the GDC. In graphics-only mode, they are part of the address bus. In mixed-mode, the following signals are output through them:

1. The value of the **image** bit is output on A17 during the horizontal blank period. The image bit indicates whether the upcoming raster line is to be treated as a part of the bit-mapped graphics area or a part of the coded character area.

2. The external line counter clear pulse is output on A16 during the horizontal blank period. This signal signifies the first time a coded character word is displayed.

3. The cursor position is indicated on A17 during the active period of the raster line in the coded character area. The A17 is asserted high during the time when the GDC sweeps over the cursor position.

4. The cursor blink rate is indicated on A16 during the active period of the raster line.

3.4.1 Image Bit

The image bit is valid after 10 word clocks during the horizontal front porch period. This is why the horizontal front porch must be greater than 5 words for the mixed mode of operation. A binary counter is used to count 10 clocks after the fall of the HS signal. The value of 4 is loaded into the counter on the falling edge of the HS signal. The count is incremented on the leading edge of WCLK clock until the CO is generated, at which time the value of the image bit is latched onto a register. The signal

GMODE is high for graphics area, and CMODE is high for coded character area. The logic diagram is shown in Figure 3-18.
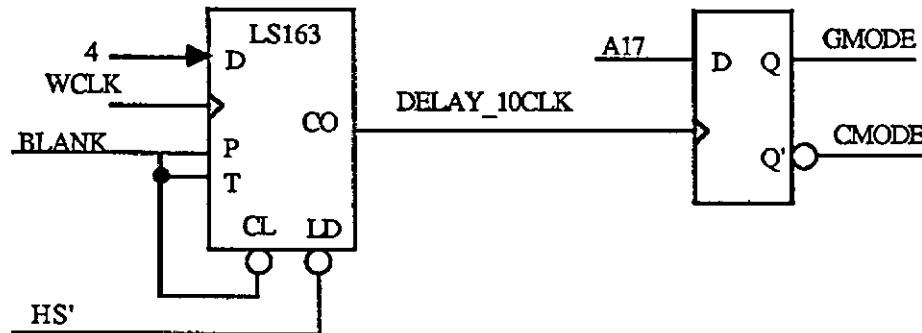
Figure 3-18. Logic diagram for capturing the image bit.

3.4.2 Line Counter Logic

The external line-counter clear pulse is also available after the 10th word clock. The DELAY_10CLK signal is also used to catch this pulse. The line count is fed to the coded character table look-up rom as the 4 lowest address lines. The logic diagram for the external line counter is shown in Figure 3-19.
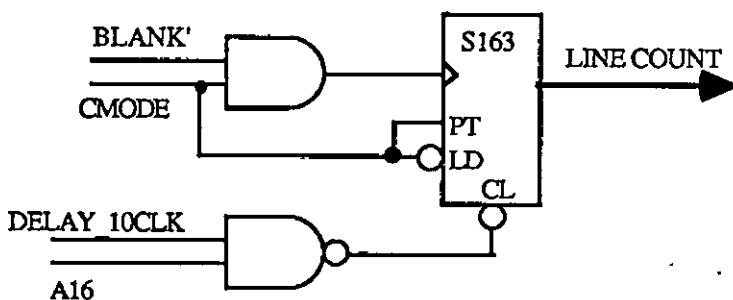
Figure 3-19. Logic diagram for the external line counter.

3.4.3 Character Cursor

The character cursor is generated simply from the following logic:

$$CURSOR = BLANK' * A_{17} * A_{16} * CMODE$$

Since the GDC asserts the cursor and its attribute information at the appropriate time, the CURSOR signal can be merged with the video information without any problem. However, because of the inherent pipe-lined operation of the display process, the cursor must be delayed exactly the same amount as the pixel information. The delay in this system is one display cycle, or four clocks. The HS and VS, as well as BLANK, signals must also go through the same delay. A two stage latch system, clocked by the VSR_LD signal will provide the desired delay.

Chapter 4

HARDWARE IMPLEMENTATION

The logic of the GDC board was designed and simulated in part on the SCALDsystem from Valid Logic Systems Inc. The layout and routing of the board was done on the Merlyn-PCB from VR Information Systems. Since different partnames and netlist formats are used by the two systems, an interface program was written to iron out the differences in transferring data from one system to the other. In this section, the details of the hardware implementation are presented. For more detailed discussion on the use of the CAD systems, refer to the following:

1. An Introductory Guide to the SCALDsystem,

2. SCALDsystem User's Manual,

3. Merlyn-PCB User's Manual.

4.1 Logic Design with the SCALDsystem

There are 4 stages to designing a circuit with the SCALDsystem: schematic capture, timing verification, logic simulation, and packaging. Since the timing verification and the logic simulation stages are optional in SCALDsystem, it is possible to generate the netlist with the packager directly from the schematic capture stage. The omission of the verification and simulation may be tolerable with simple design, but not so in general, for they are a indispensable part of the design process.

4.1.1 Schematic Capture

The entire design of the GDC board is composed of eight pages. Of the eight, the logic diagrams for the functional parts of the GDC board are described in the first four and the last pages. The contents of each page are described below:

page 1: The bus interface between the GDC and display memory is described. The logic for generating multiplexed row/column address is also described.

page 2: The display memory is shown.

page 3: The logic for generating signals to control the display memory and the video shift registers are described. The video monitor interface is also shown.

page 4: The host interface between the GDC, DMAC, and the zoom pre-scaler is described.

page 5: The interface to the floppy and Winchester hard-disk controller is shown. This part of the design in not implemented.

page 6: The capacitors, pull-up resistors, and the edge connector interface is described.

page 7: The numerous resistors used to fill the remainder of the GDC board with plated holes are shown. Each resistor is mapped to a single in-line resistor pack so that it will have 10 holes.

page 8: The logic for coded character generation, suppressing every other VSR_LD signal, and capturing the image bit are described.

4.1.2 The Custom Library

The SCALDsystem has a set of libraries that contain both the logical and physical description of commonly used IC, mostly the TTL series, chips. However, the libraries did not have all of the parts used in the design. There was no part description for memory devices, PAL's, nor microprocessors. In order to fully utilize the capabilities of the SCALDsystem, a library containing the description of the devices that are missing from the existing libraries was created.

For each device, there should be, at least, one body drawing. The body drawing describes the shape of the device, and there may be more than one desired shape (different versions). A trivial example is the logic symbols for a NOR gate, as shown in Figure 4-1.

$$(A + B)' = A' * B'$$
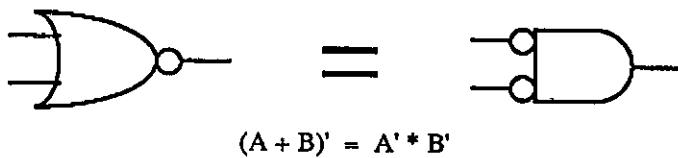
Figure 4-1. Two equivalent representations for a NOR gate.

In addition to the body drawing, the drawings for timing and simulation models should be defined if timing verification and logic simulation of the design are to be carried out. The newly defined timing and simulation model should be individually tested and verified thoroughly before put into use. This is not an easy task, even for simple devices such as dynamic RAM's or PAL's. The problem is far more complex for most VLSI devices, for their functions are in software control. However, there is a way around to the difficulties of defining timing and simulation models, and it is presented along with the discussion on the timing verification and logic simulation processes.

In addition to the logical description, a physical description is also needed to check the fan-in and fan-out requirements of a device, as well as the pin numbers and types (input, output, both, tri-state, or open-collector). This information is used by the packager to verify the physical correctness of the design. The physical description for the devices that are not in the existing libraries, but were used in the design, are collected in the Custom library. The contents of the Custom library is shown in Appendix E.

4.1.3 Timing Verification

Because of the difficulty of defining the timing models for the microprocessors, especially for the GDC, the timing verification for the design was done by part, in steps. The timing verification on the host interface was not necessary because of the relatively long access cycle (at 1.1 MHz.) and the simplicity of the interface. For the similar reason, the timing verification on the bus interface between the GDC and the display memory was not needed. For the remaining portions of the design, timing verification was necessary, and was done.

Instead of creating a timing model that would generate a variety of signals in response to the software control, several models, one for each mode of software control, were envisioned. For instance, the GDC has two modes of operation; display and RMW cycles. The behavior of the signals, and the signals used in the design, are different for each mode. To verify the portion of the design that functions as the display processor, the timing model of a display cycle is used. To verify the portion of the design that functions as the graphics processor, the timing model of a RMW cycle is used. In this approach, the problem of modeling the GDC becomes easier.

An alternative that is even simpler than actually defining a model is to use the assertion property of the signal names to model the behavior of each signal. Instead of verifying every signal between the GDC and the video monitor all at once, the circuit is cut into several sections, and each section is verified individually. To run the verifier on any one section in which the timing models are available for all devices, it is only necessary to model the behavior of the signals that are input to the circuit. The timing behavior of the inputs are described by the assertion property of the signal name. This approach is illustrated in Figure 4-2. The outputs from the previously verified sections can be used as the inputs to other sections of the design.
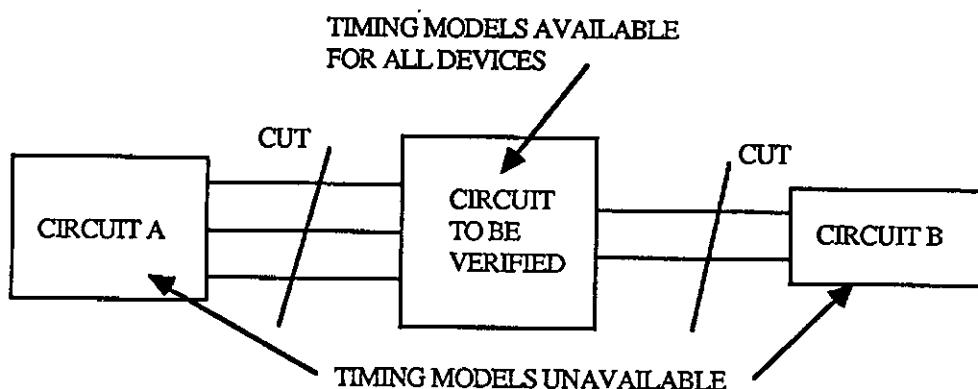
Figure 4-2. An alternative to using a timing model for a device.

The entire design is divided into eight pages, with all the critical sections of the design collected purposely into two pages (pages 3 and 8). These two pages contain the circuits for generating the signals that control the display memory and video shift registers. The design in page 3 was run through the timing verifier with the ALE signal occurring at three different times: at the earliest possible (30 nano-seconds after the beginning of a cycle), at the nominal (at 65 nano-seconds after), and at the latest (at 100 nano-seconds after). The result of the first and the third runs reported the timing

violations, specifically the set-up and hold violations for about 5 nano-seconds, for a flip-flop (U36, 14P on page 3). If the ALE signal is asserted about 5 nano-seconds later than the earliest possible, and about 5 nano-seconds earlier than the latest possible, then the timing violation does not exist. Since there is a very low probability that the ALE signal will be asserted at either end , the design was assumed to be free of timing problems. The actual measurement of the ALE indicated it to be about 60 nano-seconds after the beginning of a cycle. The result of the three timing verification runs is shown in the Appendix F. The timing verification on other parts of the design indicated no violations, mainly due to the relatively long frame of time in which they are subjected.

In the SCALDsystem libraries, the timing models are very simply defined so that many of the devices do not remember the logic values of the previous states. They remember only when a signal is stable and when it is in violation. For example, the simple combinational gates produce the logically correct outputs from the valid inputs. However, the sequential devices such as flip-flops, counters, and shift registers will only indicate whether an output is in a stable or in an unstable state. For this reason, the generated timing waveform is difficult to comprehend, and not of much help to the designer. This separation of timing verification and logic simulation is claimed to shorten the design time.

An attempt was made to define the timing models for the GDC, DMAC, PAL's, and the dynamic RAM, but was later abandoned because of the difficulties involved in handling the software controlled aspect of the processors and the amount of additional work required to validate the models. However, a timing model for a generic 64K dynamic RAM was developed and was partially tested. The model is generic in the sense that most 64K dynamic RAM chips have similar timing specifications. The model checks for the following violations:

1. The set-up and hold times for the address and data input (write cycle) to RAS*
   and CAS* signals,

2. The minimum pulse width time of RAS* and CAS* signals.

The model reflects the random read and write cycles of operation, and not of
read-modify-write cycles nor any of the refresh cycles. The model was tested with
signals that violate the specification, as well as with valid inputs. The model generated
the expected result, reporting violation only when it should. The model is shown in
Appendix G.

4.1.4. Logic Simulation

Due to the difficulty of defining the simulation models for the microprocessors
(GDC, DMAC, and 6809E) that are even more complex than the timing models, the
logic simulation of the design was done in a similar fashion to the timing verification;
by parts and with assertion property of signal names. Again, the portions of the design
which seem simple were analyzed with hand-drawn timing diagrams. Only the design
in pages 3 and 8 were simulated.

Unlike the timing verifier, which runs in batch mode, the interactive logic
simulator proved to be a very helpful tool. Break-points could be set on any signal
with break conditions that depend on other signals. The values of the signals could be
set to any legal value at anytime, including the initial values. Even the logic in the
design could be modified temporarily and simulated. The simulator, in addition,
worked as a debuger. Almost anything was possible, with the exception of producing
a hardcopy of the resulting waveforms it generated on the screen. However, the
upgraded version of the SCALDsystem has the capability to produce a : hardcopy of
the simulation results, identical to that produced by the timing verifier.

4.1.5 Packaging

Once the design is finished, it is packaged to produce the description of it for implementation. The simple gates are mapped to the physical devices, and the net-loading is checked. The drive capability of each device is checked against the total loading imposed on the net by the connected devices. There were no loading violations in the design. Since the realm of the SCALDsystem's functions is the logical aspect of the design, additional parts, such as resistors and capacitors, were added to the design. All signals with the value of logic high were connected to the pull-up resistors. In addition, several connectors were added for the off-the-board interface. All these required design modification, but the timing verification and the logic simulation were not needed on the modified design.

The output from the packager is a partlist and a netlist. Since no one has ever used the SCALDsystem to actually implement a design, and since the design of the GDC board was not fully verified with the SCALDsystem, the netlist was checked against the design manually. Fortunately, there were no discrepancies between the design and the netlist.

4.2 SCALDsystem and Merlyn-PCB Interface

At the time the design of the GDC board was completed, the SCALDsystem did not have the placement nor the routing capabilities. For these the Merlyn-PCB package was used. However, because of the differences in netlist format and partnames, an interface program was developed to automate the conversion process of the data format from one system to the other. The following features were required of the interface program:

1. Generate the master partlist from the parts library in the Merlyn-PCB.

2. Generate a concise partlist from the output of the packager.

3. Check the partlist of the design against the master partlist to make sure that all parts exist in the parts library of the Merlyn-PCB.

4. If a part not in the master partlist is found, search the system-wide name transformation file to see if a different name is used for the part in the Merlyn-PCB. If found, change the name to that which is recognized by the Merlyn-PCB. Make note of the change so that the original name can be restored. If not found, notify the user and ask for a new name.

5. Ask the designer if there are any connectors that were split up into several parts because of the inability of the SCALDsystem to handle connectors with more than 62 connections. If there are, merge them into one.

6. Convert the netlist format to that required by the Merlyn-PCB, using new names for the parts whose names are missing from the master partlist.

7. Transfer to the Merlyn-PCB on TRAC Vax with either Kermit or fast PIB link.

8. If an engineering change has been made on the design while in the Merlyn-PCB, such as pin or gate swap, modify the original design in the SCALDsystem to reflect the changes with the feedback capability of the packager.

There exist additional features of the interface program that facilitate the problem of porting the netlist from one system to the other, but are not discussed here. For the benefit of new users, on-line help files are installed on the system. The following man pages are available on the Unix [tm of Bell Laboratories] system the Valid workstation operates under:

mc   create connectors on-the-fly, without using the GED,

rmec   delete the connectors on-the-fly,

tomerlyn the first half of the interface program that transfers data from the

      SCALDsystem to the Merlyn-PCB,

tovalid   the other half of the interface program that transfers data from

      the Merlyn-PCB to the SCALDsystem,

makemasterpartfile generates the master partlist from the parts library in the

      Merlyn-PCB.

The program listing, written in C-shell script, is shown in Appendix H.

4.3   Board Design with the Merlyn-PCB

The Merlyn-PCB package does have the schematic capture capability, but lacks the user-friendliness of the SCALDsystem's GED. In addition, it does not provide the timing verification nor the logic simulation functions. Fortunately, it has the feature to import the netlist generated by other CAD systems, such as the SCALDsystem. This is exactly how two systems work together as a single CAD system in the TRAC laboratory.

Before the netlist can be imported into the Merlyn-PCB, the project library must have the physical description of all the parts referenced in the netlist. Each part description is composed of two files: symbol and package. The symbol of a part describes the logic function of the device, such as AND, NOR, or NOT. The package of a part describes the physical shape and size of the device, such as the number and location of pins, and whether the pins in a gate or the gates in a chip are swappable. There exist generic figures, such as DIP14, DIP20, DIP40, etc., that specify the exact dimensions for the commonly used device packages. For instance,the figure DIP14 specify the physical dimensions of a 14-pin dual in-line package.

After the netlist is imported, a board is defined. The outline of the board is defined, and the edge connectors, if used, are placed. The exact dimensions, down to 1 mil, of the board and the connectors are needed to define the board. If desired, sections of the board can be designated to be free of vias, traces, or components. Once the board is defined and the components are placed, no further change to the board is possible. A six-layer, S-100 bus compatible board was designed for the graphics system to be implemented. The two inner-most layers are reserved for the power and ground. The outer-most two layers are designated for vertical traces, for on them the edge connectors are placed. The other two layers are reserved for the horizontal traces.

The layer preferences are strongly enforced during the first few passes of the routing. The layer assignment is shown Table 4-1.

Layer	Preferred Direction
1 (top)	vertical
2 (inner 1)	horizontal
3	power
4	ground
5 (inner 2)	horizontal
6 (bottom)	vertical

Table 4-1. Layer assignment.

The components were placed one by one manually from one end to the other end of the board. Since the memory system has a regular interconnect pattern, placement and routing was done on it before other components were even placed. After many trial-and-errors, an acceptable placement was found, and the router was called. Seven passes of the routing, each pass with less. · strict router settings, were required to route all but a dozen traces. The remaining traces were manually routed. After the routing was finished, the engineering optimization was done to remove hangers, stair cases, wells, and to reduce the vias. Many traces were manually relocated to other layers to further reduce the via counts. The finalized design of the GDC board is shown in Appendix I. The detailed hardware diagram of the original design from which the printed circuit board is made is shown in Appendix D. Only the pages showing the differences between the original and the modified design are included.

4.4 Modifications on the GDC Board

After the printed circuit board was produced from the original design, a few modifications on the GDC board was necessary to add more functions and to change the host interface. These late changes required cutting a few traces. The traces that were cut are indicated in the copy of the artwork in Appendix I.

4.4.1 Internal Color Computer Bus

For the new host interface, the signals BS, BA, S_0, S_1, and S_2 were brought out from the Color Computer's internal bus to a female DB15 connector on the back of the computer. The pin assignments to the connector are shown in Figure 4-3.
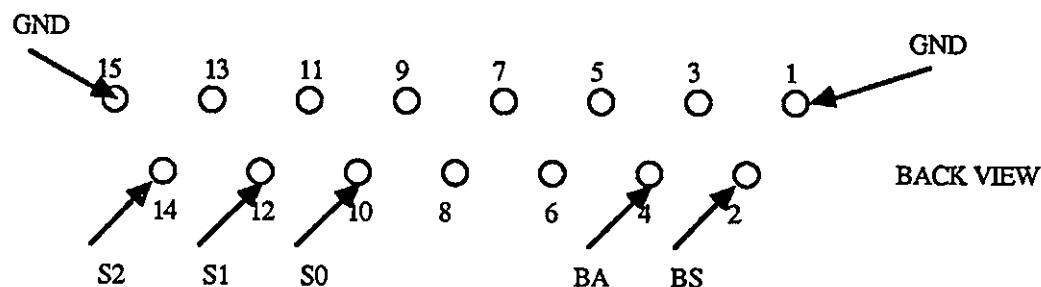
Figure 4-3. Pin assignments on the DB15 on back of the Color Computer.

The above signals are carried through a 16-pin flat cable to a 16-pin dip socket in the GDC board. The pin assignments on the socket are as follow: BS to pin 1, BA to 2, S_0 to 5, S_1 to 6, S_2 to 7, and GND to 8. These signals are, then, connected to the edge connector. The pin assignments for these are shown in Table 4-2.

PIN NUMBER (ZIF)	SIGNAL NAME	PIN NUMBER (EDGE)
	BS	16
	BA	15
	S0	10
	S1	9
	S2	8
3	/HALT	67
5	/RESET	75
6	E	24
7	Q	25
10	D0	95
11	D1	94
12	D2	41
13	D3	42
14	D4	91
15	D5	92
16	D6	93
17	D7	43
18	R/W	17
19	A0	79
20	A1	80
21	A2	81
22	A3	31
23	A4	30
24	A5	29
25	A6	82
26	A7	83
27	A8	84
28	A9	34
29	A10	37
30	A11	87
31	A12	33
33,34	GND	50,100
37	A13	85
38	A14	86
39	A15	32

Table 4-2. Pin assignments on the edge connector on the GDC board.

4.4.2 Color Computer Expansion Bus

The Color Computer expansion bus is brought through a 40-pin flat cable to the ZIF socket on the board. From the ZIF socket, the signals are distributed to the edge connector. The pin assignments and the connections to the edge connector on the board is also shown in Table 4-2.

4.4.3 Added Functions

To the original design, the capability to handle the coded character was added, which resulted in modification to several parts of the design. The modified parts of the design are noted with boxes around them in page 3. The design in page 8 are an addition to the board.

4.5 Parts List

The parts list and summary of the devices used in the GDC board are shown in Appendix L.

Chapter 5

SOFTWARE DESCRIPTION

This section contains the description of the device driver for the GDC board. The device driver is written in the high level language C so that it can be easily ported to other computers running under the OS-9 operating system. Although the efficiency is sacrificed for not being written in the host machine language, the benefit of being easily portable and maintainable seemed worth the sacrifice. With the upgrade of the node processor of the Look Ahead Network to the 68000 family of microprocessors, C seemed to be a wise choice for the language of implementation.

The OS-9 operating system is an optimized version of Unix, written entirely in assembly language. Currently, there exist OS-9 operating systems on 6809 and 68000 based machines. Like Unix, OS-9 supports the unified i/o concept, which makes the task of adding i/o devices easier. The detailed description of the OS-9 is not presented here, except that each device must have a device driver and a device descriptor. For the detailed description, the reader is referred to the System Programmer's Manual for the OS-9 [OS-9S 84].

5.1 C Compiler in OS-9

The OS-9 on the Color Computer is written in 6809 assembly language, and hence does not provide any support for developing device drivers in C. Since the device driver needs to access specific variables in the device descriptor, path descriptor, device static storage, and even in the MPU registers, a detailed study of the C compiler is required. It is necessary to know how the parameters are passed between the functions, which registers are used where and when, how the variables are allocated in memory, how simple and complex data types are treated, and more. It is also necessary to mix the routines written in both C and 6809 assembly languages in the device driver. Fortunately, the C compiler has the capability to accept embedded 6809 assembly codes in the program. The finding is presented below.

5.1.1 Simple Data Type Representation

The internal data type representation of interest is presented in Table 5-1. This information is found in the C Compiler User's Guide, but is presented here for the completeness of the discussion [OS9C 83]. All pointer variables are treated as unsigned type, and all constants are assumed to be of integer type.

DATA TYPE	INTERNAL REPRESENTATION
char	8 bit two's complement binary
int	16 bit two's complement binary
unsigned	16 bit unsigned binary
long	32 bit two's complement binary

Table 5-1. Simple data type representation.

5.1.2 Register Usage

The B accumulator is used on the assignment operations with character variables. The A accumulator is never used by itself, except when the type conversion from character to integer is required. Since all arithmetic operations are carried out with integer variables internally, the character variables are always coerced into integer type with sign extension operation. The D accumulator is used on all comparison operations and on assignment operations with integer or unsigned variables. It is also used to pass the return value from the functions. The register X is used as the index pointer to complex variables such as arrays and structures. The register Y is used only with the **direct** storage class variables. The direct storage class is an extension to the Kernighan and Ritchie's definition of C. The User stack pointer U is reserved for register type variables, and is used only when a register variable is being referenced. There can be only one register variable in a function. The original value of U register is preserved throughout the execution of a program.

5.1.3 Variable Allocation

· All variables are allocated on stack, with the exception of direct storage class variables, which are allocated on page 0 of the system memory. The variables local to a function is pushed on the stack in the order they are declared, one byte for character and two bytes for integer and unsigned variables. The value of the U stack pointer is saved before the local variables, as illustrated in Figure 5-1.
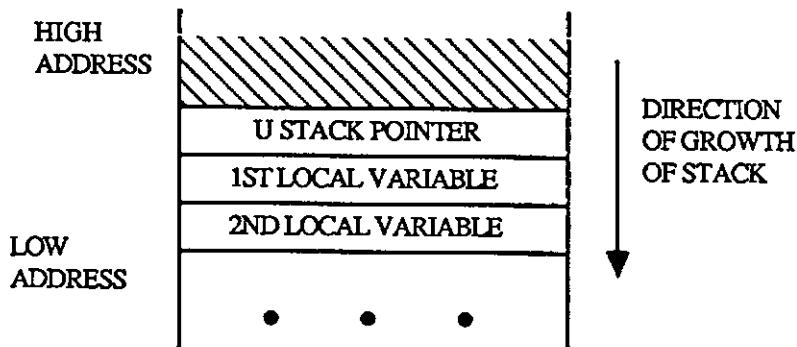
Figure 5-1. Variable allocation on stack.

5.1.4 Parameter Passing and Function Call

Actual parameters, on the other hand, are pushed on stack in reverse order of declaration; that is, the value of the first argument is pushed on the stack after the second argument. This may seem a bit strange, but it simplifies the problem of keeping track of parameters on the stack. The diagram in Figure 5-2 illustrates the contents of the stack during the run time right after the envoked function is entered.
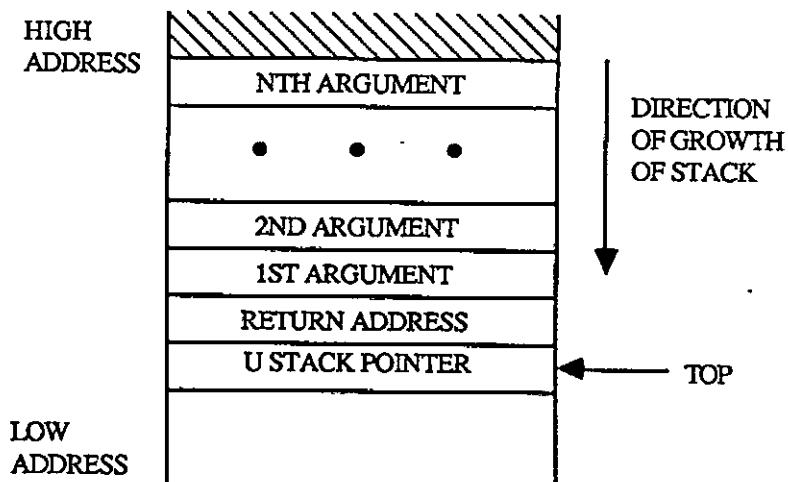

Figure 5-2. Contents of stack showing parameter passing convention.

The physical locations of the arguments to a function start at the 4th byte from the top of the stack. If a function has local variables, they are allocated on top of the U stack pointer, pushing the arguments further down the stack. All, including the character type, arguments occupy two bytes on the stack; the character type arguments are sign extended to form integer arguments. The value of the function is returned through the D accumulator, and not through the stack, which simplifies the problem of handling functions returning no argument.

5.1.5   Pitfall of Coercion

If not careful, a comparison between a character variable and a constant or an integer variable could yield a false result that is difficult to detect. If a constant C is defined to be 0xff and a character variable V is assigned the value of -1, an equality comparison between the two would fail because C has the value of 255 (0x00ff) as an integer constant. It is important to remember the following points:

1. All constants are assumed to be declared as integers, hence sign extension is not performed on them. For example, 0xff is equivalent to 0x00ff, not 0xffff.

2. The character variables are coerced into integer variables for all comparison operations.

3. The coercion from integer or unsigned to character type takes the form of truncation of the leading 8 bits.

4. Use decimal numbers, if possible, to eliminate the confusion arising from the use of more than one data type in a statement.

5.2 Device Driver Design Consideration

Much attention was given to the implementation of the device driver so that it may be easily ported to other OS-9 installations and, possibly, to different operating systems. Because of the relatively simple host interface, porting the device driver to other operating systems did not seem too big a task compared to the task of designing the graphics system. It is obvious that the device driver be structured in such a way that it is easily modifiable and maintainable. The details of the implementation is presented in this section. The GDC board is interfaced to OS-9 as a sequential device.

5.2.1 Structure of the Device Driver

There are three memory modules that warrant close attention; the device descriptor, the device static storage, and the path descriptor. Each are an integral part of the unified i/o concept of OS-9, and has a rigid format for its contents. They play the role of bridges between the i/o manager, device driver, and the device itself, as illustrated in Figure 5-3.
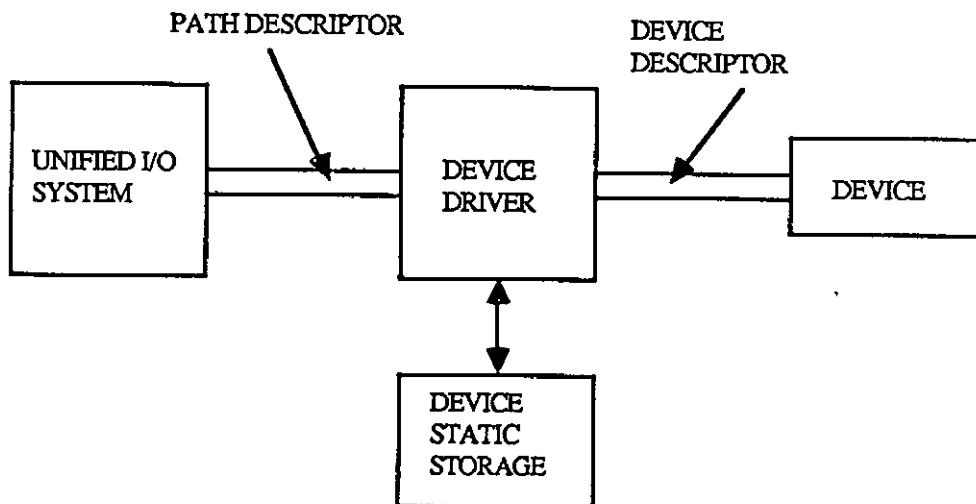

Figure 5-3. Unified i/o concept.

The device driver is organized in such a way that all operations requiring the path descriptor is grouped into one section and all operations requiring the device descriptor is grouped into another section in the program. To port the device driver to other environments, all that is required is to modify these sections. The device descriptor is used only when the device is opened for the first time to initialize the device. For this particular implementation of the device driver, only the MPU register packet address is used from the path descriptor. The access to the path descriptor is limited only to the SETSTAT routine of the device driver. Even there, it is restricted to the first few lines of the routine. Since the purpose of the device static storage is to provide the spaces for the static variables while the device is operating, the access to it is not restricted in any way. Although the format of the static storage may be different for other operating systems, the contents of it will still be the same, for the variables needed to control the device will remain the same.

5.2.2 Data Structures

The format of the path descriptor and device static storage are presented in Figure 5-4. By defining the data structures for these modules, the device driver is independent from the format of the modules, for a change in the format requires a modification only on the structure declaration. This way, the task of keeping track of the variables is left to the compiler. The format of the device descriptor is presented later.

```
struct registers {                              /* 6809 MPU register packet definition */
        char            rg_cc, rg_a, rg_b, rg_dp;
        unsigned        rg_x, rg_y, rg_u;
};

typedef struct {                                /* path descriptor definition */
        char    xx[6];                          /* not interested on these variables */
        struct registers *pd_rgs;               /* pointer to register packet */
} pd_gdc;

typedef struct {                        /* device static storage definition for GDC */
        char    port_ext,               /* extended port address */
                *port;                  /* port address */
                junk[32],               /* area used by OS-9, not of interest */
                bcount;                 /* a variable used by WRITE routine */
        unsigned   wxmin,               /* left edge of window */
                   wxmax,               /* right edge of window */
                   wymin,               /* top edge of window */
                   wymax,               /* bottom edge of window */
                   fxmax,               /* right edge of display memory */
                   fymax,               /* bottom edge of display memory */
                   param[5];            /* figure drawing parameters */
        char    zoomf,                  /* zoom factor */
                table[16],              /* used by line drawing routine */
                mode;                   /* drawing mode */
        unsigned   pattern;             /* drawing pattern */
        char    atable[8];              /* used by arc drawing routine */
} dss_gdc;
```

Figure 5-4. Data structures for path descriptor and static storage.

5.2.3 Device Descriptor

The device driver for the GDC board is written to handle a variety of monochrome monitors. The control over a device is fine-tuned with the specification found in the device descriptor. The contents of the device descriptor used in the project is presented here for the benefit of others who may wish to use the GDC board with different resolutions or with different monitors. The data structure of the descriptor is shown in Figure 5-5.

```
typedef struct {                                  /* device descriptor definition */
        char    sys[0x12];                /* used by the module header */
        char    dev_type,                 /* device type is scf */
                wwdth,                            /* window width in words */
                bwdth;                    /* display area width in words */
        unsigned  wline,                  /* window length in lines */
                  bline;                  /* display area length in lines */
        char    commands[25];                     /* upto 25 bytes of commands */
} dev_gdc;
```

Figure 5-5. Data structure of device descriptor.

Usually the size of the display memory is larger than the size of the screen, as is the case with this project. When the display memory is organized in a way that it is wider than the width of the screen, the GDC must be programmed so that it will correctly calculate the starting address of the next line. The dimensions of the display area is indicated by the constants **bwdth** (in words) and **bline**. The resolution of the screen is given by the constants **wwdth** (also in words) and **wline**, as shown in Figure 5-6.
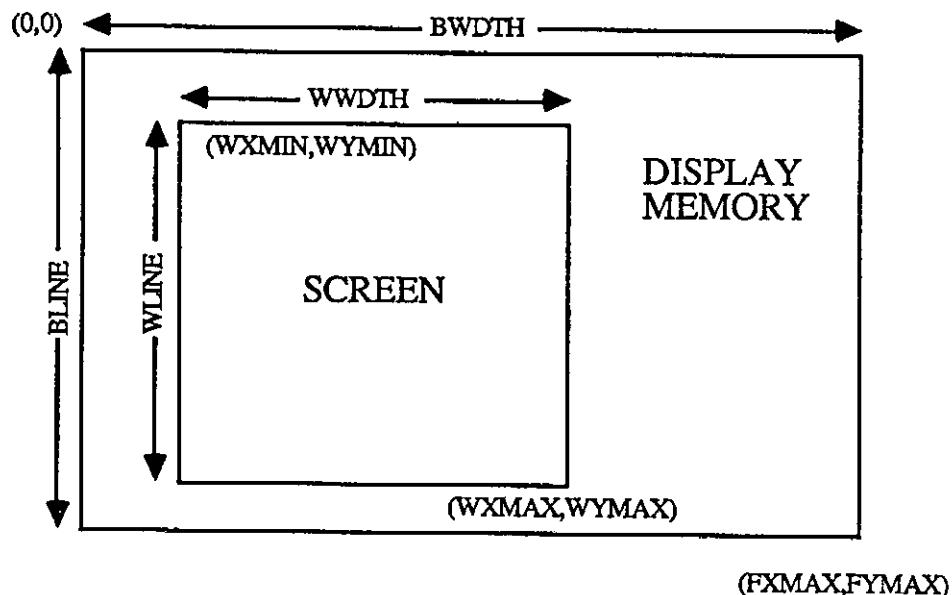
Figure 5-6. Display memory organization.

There are 64K words of display memory, each word being 16 bits, in the GDC board. For this project, it is organized into a rectangular area of 100 words wide and 655 lines long. The remaining 36 words are not used. The resolution of the monitor is 800 by 521 pixels, or 50 words by 521 lines, which occupy approximately half of the display memory. This can be changed with a slight modification on the device descriptor.

The last 25 bytes of the device descriptor is reserved for a series of commands and parameters to configure the GDC to desired operating mode as the device is opened. This is necessary in order to make the device driver capable of controlling a variety of differently configured GDC.

5.2.4   Embedded Assembly Language Codes

In implementing the device driver in C, two problems have to be solved. One problem is generating the proper module type header for the device driver with the C compiler, which generates object files with the **program** module type. The second problem is to access the arguments that are passed to the device driver through the 6809 MPU registers. The embedded assembly language codes provided the solution to both problems.

A file containing the module header of a device driver and the branch table to six device driver subroutines,written in 6809 assembly language, is created. This file is assembled to generate the relocatable header module for device drivers. This module is used as the main module, instead of the **cstart.r** module that the C compiler uses, and is linked with the relocatable module generated by the compiler. A modified version of the C compiler, written by Dr. G.J. Lipovski for this purpose, is used. However, this posed a new problem. The initialized variables are copied from the program area to the data area by the routines in **cstart.r** module. Without it, the variables must be initialized during run time. For this reason, all variables are initialized in the INIT routine during run time.

To access the arguments passed through the registers, several functions are written with embedded 6809 assembly codes that return the values of the registers. A function that returns the value of Y register is declared as **char *get_y()** , whose code is shown below:

```
get_y:
     tfr  y,d
     rts
```

The functions to access the values of the A accumulator and the U register are similarly written. Now the path descriptor, device static storage, and device descriptor can be

accessed in the device driver subroutines in C. Since the D accumulator is used to pass the return value from functions, the function **get_a()** must be the first statement in WRITE and SETSTAT subroutines. The following statements in WRITE subroutine will get the character to be written from the accumulator A and write it to the device base address.

```
#define get_dss() ((dss_gdc *)get_u())
char c, *adrs;
c = get_a();                /* get the character to be written from A acc. */
adrs = (get_dss())->port;   /* point to the device base address */
*adrs = c;                  /* write the character to the device base address
*/
```

The same technique is used to control the B and Condition Code registers on exit from the service request calls. The function **noerr()** clears the B register, which automatically clears the carry bit. The function **error(code)**, where **code** is an integer argument, loads the B accumulator with the value of **code** and sets the carry bit.

5.3 Description of the Device Driver

A brief description of the GDC board device driver is presented here. Since the device driver is written in C, the detailed accounts of the functions of each routine are not given. For that matter, the reader is referred to the program listing, shown in Appendix K.

5.3.1 INIT Routine

The main functions of the INIT routine are to initialize the device static storage and to reset the GDC for desired configuration. Since the commands needed to configure the GDC is supplied by the device descriptor, the device driver has no knowledge of the operating mode of the GDC. Different device descriptors can be used to configure the GDC for other operating modes without further modification. However, the sequence of commands and parameters are limited to a total of 24 bytes because of the physical limitation of the device descriptor file. Each command must be preceded by an opcode **NP+number**, where **number** indicates the number of parameters the command has. Commands without any parameters, such as VSYNC and BLANK, are to be given without the special opcode. Since the GDC has separate input ports for commands and parameters, the use of the special opcode NP is necessary to distinguish between the commands that require parameters and those that do not, so that command bytes are not written to parameter port and vice versa. The value of the special opcode NP is chosen such that it cannot be confused with the existing GDC commands by the device driver.

In addition, since no initialized variables can be used without the **cstart.r** routine, two look-up tables used in the driver are placed in the static storage and are

initialized in the INIT subroutine. The contents of these tables are discussed in the SETSTAT routine.

5.3.2 WRITE Routine

There are two kinds of data for the GDC; commands and parameters. Each has its own address to which the input port is assigned. Not all commands have parameters, and some can even have a variable number of parameters. Since the GDC board is implemented as a sequential device, only one character can be written to the device at a time. This may be a command or a parameter. In order to solve the problem of distinguishing the commands from the parameters, a pre-byte is defined. A study of the command byte patterns revealed that the most significant bit, bit 7, of the command byte is 0 for all with the exception of RDAT (read data), CURD (cursor position read), LPRD (light pen position read), and DMAR (DMA read). It was decided that these four commands will never be written to the GDC through the write routine, and that bit 7 of the pre-byte will be used as the flag to indicate whether it is a command byte or not. The following protocol for using the WRITE routine is defined:

1. If a command does not require parameters, write the command byte to the device.

2. If a command requires a number of parameters, first write the pre-byte consisting of the value $80 (equivalent to setting bit 7) + the number of parameters. Then write the command and the parameter bytes. The pre-byte will set the WRITE routine to treat the next byte as a command and the following bytes as the parameters.

The purpose of the WRITE routine is to provide a simple tool to test the hardware to be sure that it is working perfectly before committing to testing the device driver. Although the WRITE routine by itself provides the sufficient vehicle for controlling the GDC board, it is too primitive to create an environment that is convenient for graphics programming. In order to program the GDC board at the machine level, a very thorough understanding of the GDC is required. To overcome that, more control over the GDC board is provided through the special functions in the SETSTAT routine.

5.3.3 READ Routine

It is possible to envision a read function that will return the contents of the display memory. However, being a sequential device, it seems absurd to return the contents of the display memory one byte at a time to the caller. For this purpose, the DMA read function is provided in the SETSTAT routine.

5.3.4 SETSTAT Routine

The Setstat routine provides a fairly complete control over the operations of the GDC board to facilitate graphics programming. It is composed of many functions to program the GDC for a variety of figure drawing operations. For the details of each function, the reader is referred to the program listing and Section 4 of the 7220 GDC User's Manual. However, the description of the line and arc drawing routines are presented, for the algorithms used in these functions are not presented in the User's Manual. Other functions are, more or less, a straightforward implementation of the algorithms given in the User's Manual. First, the concept of drawing direction used by the GDC is presented.

5.3.4.1   Drawing Direction Definition

The GDC assumes that all pixel addresses are in Cartesian coordinates. The coordinate is divided into eight octants, as shown in figure 5-7.
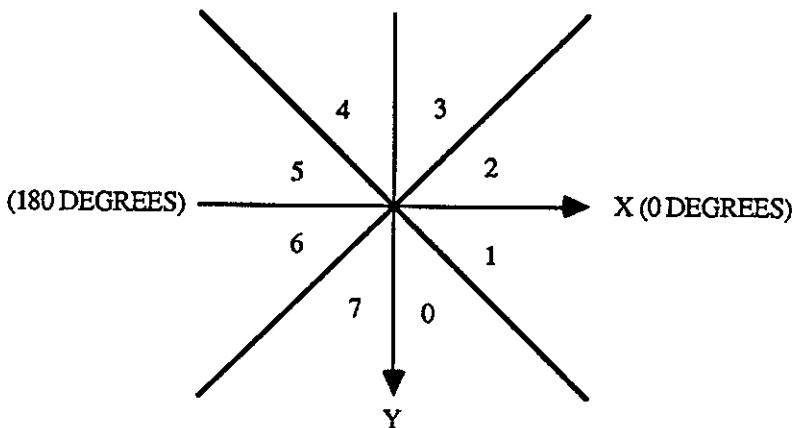
Figure 5-7. Octant Direction Definition.

In octants 1, 2, 5, and 6, X axis is the independent axis, and Y axis is the dependent axis. On the other octants, the reverse is true. Calculating the correct drawing direction and independent/dependent axes is an important part of programming the GDC. The direction parameter is used in all figure drawing operations. In vector drawing, the initial direction of a figure drawing operation is the octant in which the end point of the vector lies when the starting point is placed at the center of the octant diagram shown in Figure 5-7. If the vector lies on a boundary of two octants, the lower octant is taken as the direction. In arc drawing, the initial direction is the octant in which the end of the arc lies when the starting point is placed at the center of the octant diagram. The drawing directions for arcs are shown in Figure 5-8. Since finding the drawing directions by calculating the octant takes much time, look-up tables are used. These tables are presented in the next two sections.
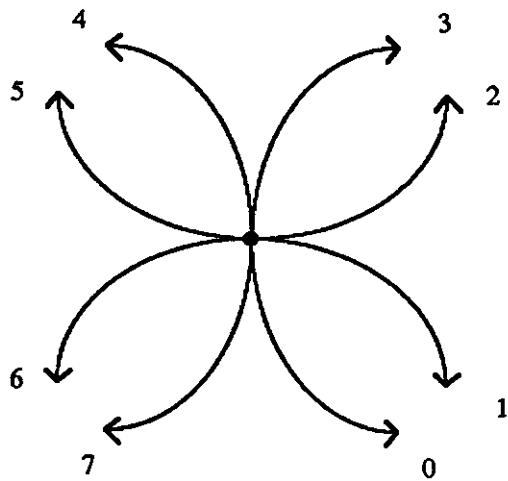
Figure 5-8. Drawing directions for arcs.

5.3.4.2 Direction Parameter Calculation for Vector Drawing

The look-up table **table**, declared in device static storage and initialized in the INIT routine, contains the coded information for calculating the initial drawing direction for vectors (lines). The index into **table** is composed of four concatenated binary variables DCBA whose values are shown below:

$D = 1$ if $dx < 0$. Otherwise it is 0.

$C = 1$ if $dy < 0$. Otherwise it is 0.

$B = 1$ if $dx = dy$. Otherwise it is 0.

$A = 1$ if $dx > dy$. Otherwise it is 0.

For instance, the index value for a vector from coordinates (1,9) to (2,5) would be 4 (DCBA = 0100). The code $17 is read from the 5th entry (index value of 4) of **table**. Since $17 is an odd number, the Y axis is the independent axis for this vector. The direction is found by right shifting the code once and taking the last three bits. In this case, it is octant 3.

By looking at the required parameters, it is obvious that the vector drawing operation in the GDC is a direct implementation of the famous Bresenham's line algorithm [FOLE 82]. The parameter calculations for Bresenham's algorithms and that of GDC is shown side by side for a comparison purpose below:

dx = ABS(x2 - x1)   DC = ABS(DeltaI)

dy = ABS(y2 - y1)

d = 2 * dy - dx   D = 2 * ABS(DeltaD) - ABS(DeltaI)

incr1 = 2 * dy   D1 = 2 * ABS(DeltaD)

incr2 = 2 * (dy - dx)   D2 = 2 * [ABS(DeltaD) - ABS(DeltaI)]

5.3.4.3   Direction Parameter Calculation for Arc Drawing

The second look-up table **atable**, also in the device static storage, supply similar information to the arc drawing routine. It contains the starting directions for the arcs in each octant. The contents of **atable** is illustrated in Figure 5-9. The numbers inside the circle represent the index into the look-up table, and the numbers outside the circle represent the contents of the table, the directions. For instance, the first entry (index 0) of the table is for an arc whose starting angle is in octant 2 (from 0 to 45 degrees). Its direction is 4. The index is calculated with an integer division of the starting angle by 45 degrees.
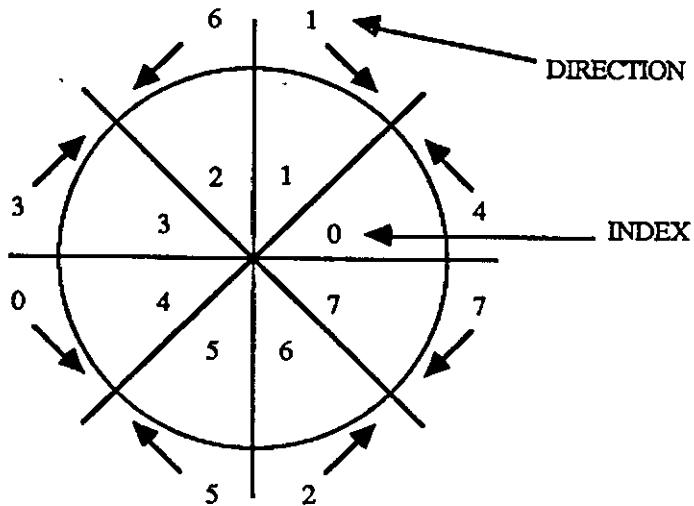
Figure 5-9. Organization of arc direction look-up table **atable**.

The arc drawing function of the GDC cannot draw an arc whose span covers more than one octant at one time. It can only draw arcs whose starting and ending angles lie in the same octant. For example, to draw an arc from 30 degrees to 50 degrees, two arcs must be drawn; from 30 to 45 degrees and from 45 to 50 degrees. Notice that the drawing direction for octant 3 (angles between 45 and 90 degrees) is from 90 to 45 degrees. So, the second arc is drawn from 50 to 45 degrees. This is true for all arcs whose drawing direction is odd.

The function **arc** takes an arc of arbitrary length, divides it into smaller arcs so that both starting and ending points of each arc are in the same octant, and calls upon the **A_arc** function successively to draw one arc at a time until all arcs are drawn. **A_arc** is the function that actually draws an arc. For all arcs whose drawing directions are odd, **A_arc** draws from the ending point to the starting point, starting point being the smaller angle.

5.3.4.4   Sine Function

To calculate an arc length, a parameter required for an arc drawing operation, the trigonometric sine function is needed. Unfortunately, this function is not available from the C compiler that was being used. So, a routine that calculates the sine value of an angle using the angle and a correction factor is developed. The equation

$$\text{SINE(angle)} * 1000 = (\text{angle} * 16 + \text{correction})$$

is used to generate the sine function. The function sin1k returns an integer value that is 1000 times the value of the sine function for an angle to eliminate the necessity of floating-point operations. The equation is designed to have the accuracy of 0.0005, which may produce an error of one pixel for arcs with a radius larger than 2000 pixels. If used with arcs shorter than 1000 pixels, the equation will provide the error-free results. The correction factors for the angles from 0 to 45 degrees are shown in Table 5-2. Using a long integer to store the intermediate result, the arc length is calculated by the equation

$$\text{long\_temp} = \text{sin1k(angle)} * \text{radius} / 1000$$

To preserve the accuracy, the multiplication must be carried out before the division.

Table 5-2. The correction table for SINE function.

	1	2	3	4	5	6
1					16	
2	ANGLE	SINE(ANGLE)	ROUGH VALUE	CORRECTION	ERROR	MARGIN
3	0	0	0	0	0	
4	1	0.0174524	16	1	0.00045	2222
5	2	0.0348995	32	3	-0.0001	-10000
6	3	0.052336	48	4	0.00034	2941
7	4	0.0697565	64	6	-0.00024	-4167
8	5	0.0871557	80	7	0.00016	6250
9	6	0.1045285	96	9	-0.00047	-2128
10	7	0.1218693	112	10	-0.00013	-7692
11	8	0.1391731	128	11	0.00017	5882
12	9	0.1564345	144	12	0.00043	2326
13	10	0.1736482	160	14	-0.00035	-2857
14	11	0.190809	176	15	-0.00019	-5263
15	12	0.2079117	192	16	-0.00009	-11111
16	13	0.2249511	208	17	-0.00005	-20000
17	14	0.2419219	224	18	-0.00008	-12500
18	15	0.258819	240	19	-0.00018	-5556
19	16	0.2756374	256	20	-0.00036	-2778
20	17	0.2923717	272	20	0.00037	2703
21	18	0.309017	288	21	0.00002	50000
22	19	0.3255682	304	22	-0.00043	-2326
23	20	0.3420201	320	22	0.00002	50000
24	21	0.3583679	336	22	0.00037	2703
25	22	0.3746066	352	23	-0.00039	-2564
26	23	0.3907311	368	23	-0.00027	-3704
27	24	0.4067366	384	23	-0.00026	-3846
28	25	0.4226183	400	23	-0.00038	-2632
29	26	0.4383711	416	22	0.00037	2703
30	27	0.4539905	432	22	-0.00001	-100000
31	28	0.4694716	448	21	0.00047	2128
32	29	0.4848096	464	21	-0.00019	-5263
33	30	0.5	480	20	0	*DIV/0!
34	31	0.5150381	496	19	0.00004	25000
35	32	0.5299193	512	18	-0.00008	-12500
36	33	0.544639	528	17	-0.00036	-2778
37	34	0.5591929	544	15	0.00019	5263
38	35	0.5735764	560	14	-0.00042	-2381
39	36	0.5877853	576	12	-0.00021	-4762
40	37	0.601815	592	10	-0.00018	-5556
41	38	0.6156615	608	8	-0.00034	-2941
42	39	0.6293204	624	5	0.00032	3125
43	40	0.6427876	640	3	-0.00021	-4762
44	41	0.656059	656	0	0.00006	16667
45	42	0.6691306	672	-3	0.00013	7692
46	43	0.6819984	688	-6	0	*DIV/0!
47	44	0.6946584	704	-9	-0.00034	-2941
48	45	0.7071068	720	-13	0.00011	9091

Chapter 6

PROGRAMMING GUIDE

This section contains the programming guide to the GDC board. To provide a fairly sophisticated environment for graphics programming, a package of utility routines are developed so that a user does not have to know a great deal about programming the GDC. The package is implemented as service functions in the SETSTAT routine. The function codes are defined in the gkssvc.h file, shown in Appendix K. The users are encouraged to use the constants defined for the codes in this file rather than the codes themselves. First, the programmer's view of the GDC board graphics system is presented.

6.1 Programmer's View of the System

The GDC board has a total of 64K words of display memory. Since the GDC is capable of handling two separate display areas, the memory system can be partitioned into two areas. The size and organization of the partitions, as well as the nature of the information stored in them, are independent from each other. The data in each partition can be thought of bit-mapped (for graphics) or coded (for coded characters). It is certainly possible to have two separate graphics areas, two separate coded characters areas, or one of each in the system. Of course, the display memory does not have to be partitioned at all. Not only the display memory, but the screen can be partitioned vertically into two regions, with each region displaying the data from the partitioned display areas. This is illustrated in Figure 6-1.
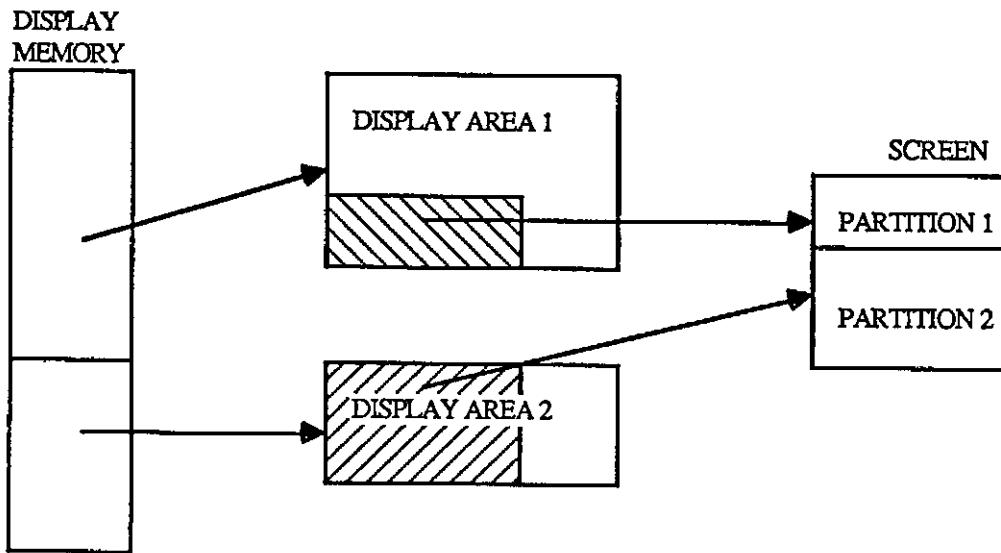
Figure 6-1. The partitioned display areas.

Note from the above diagram that the size of the partitions in the display memory and the screen are not related to each other. It is possible to allocate a larger portion of the screen to a smaller sized display area. However, the pitch, or the width of the display area, must be same for both partitions. The mapping of each partition of the screen to its respective display area is also done separately. In effect, two separate screens, each capable of panning and scrolling within its display area, are projected onto one display device.

The origin is positioned at the top left corner of the display area, so the direction of increasing pixel value for Y axis is downward. For X axis, it is toward right, as in Cartesian coordinates. If viewed as a two-dimensional array of words, the display area is organized in row-major order; that is, the lower addressed locations are used to fill a row before starting the next row. Within each word, the lesser significant bits are assigned to lower addressed pixels; that is bit 0 is assigned to a pixel having a lower coordinate than bit 1.

6.2 Figure Drawing

A brief description of how figures are drawn in the display memory is presented here. Although this information is contained in the 7220 GDC User's Manual in detail, it is summarized here because a full understanding of it is crucial for using the graphics package.

There are four modes to the figure drawing operations; set, reset, complement, and replace. In set mode, a drawing operation modifies the pixel that is part of the figure to the logic 1 state so that the pixel is visible. In reset mode, a figure drawing operation modifies the pixel to the logic 0 state, regardless of its previous state, so that the pixel is invisible. The reset mode is useful for erasing figures or drawing figures in reverse video background. In complement mode, the new state of the pixel that is modified is the complement of its previous state. This mode is also useful for erasing figures; a figure drawn twice, on top of the previous one, will erase it. In many graphics systems, the cursor is drawn in this fashion instead of using separate cursor generating circuitry. In replace mode, the pixels under modification are replaced with the bit pattern in the PATTERN register.

The PATTERN register is a 16-bit wide register that specifies which bits are to be modified during the figure drawing operations. The bits in the registers are used one at a time in round-robin fashion. If one bit of this register is 0, then one out of every 16 pixels will not be modified by the figure drawing operations, regardless of the mode. For instance, in set mode, figures made of dotted lines can be drawn if the pattern register has every other bit cleared. In reset mode, every other pixel will be erased with the same pattern. With the right combination of the mode and pattern, a variety of figures can be created. Once set, they affect all subsequent figure drawing operations until reset to other settings.

6.3 Service Function Description

In order to provide a consistent and uniform parameter passing convention to the functions, a structure **ppacket** is defined as

```
typedef struct {                /* parameter packet */
        unsigned   _p1, _p2, _p3, _p4, _p5;
} ppacket;
```

Instead of passing the parameters themselves, a pointer to the parameter packet is passed to the functions. The number of parameters declared in ppacket is arbitrary; some functions may use only one while others may use all five parameters. The pointer to the packet is passed through the X register. In assembly language programs, the functions are called with the following calling sequence:

```
            LEAX  PPACKET,pcr
            LDA  #function_code
            OS9  I$SETSTT
            . . .

PPACKET     FDB  _p1,_p2,_p3,_p4,_p5
```

In C programs, they are called with the following sequence:

```
struct  registers       {
        char   rg_cc,rg_a,rg_b,rg_dp;
        unsigned  rg_x,rg_y,rg_u;
} reg;
ppacket *parameter;

reg.rg_a = function_code;
reg.rg_x = (unsigned) parameter;
_os9(I_SETSTT, &reg);
```

The functions do not return any error conditions, unless undefined service request is made, nor do they generate any outputs. The description of each function is presented below. The name of the device driver is GDC.DD.

6.3.1 Blank

Function Code   _blank

Input   none

Description

    This function blanks the screen. Since all blanked periods of a frame is used by RMW cycles, this function is normally used before extensive modification to the display memory is done to speed up the modification. The display memory is not affected by this function alone.

6.3.2 Unblank

Function Code   _display

Input   none

Description

This function un-blanks the screen.

6.3.3 Set Background On

Function Code   _bkgndon

Input   none

Description

This function sets entire display memory to on state.

6.3.4   Set Background Off

Function Code   _bkgndoff

Input   none

Description

This function sets entire display memory to off state.

6.3.5 Zoom Display

Function Code	_dzoom
Input	_p1 = zoom factor

Description

This function changes the zoom factor of the display operation of the GDC. The display zoom operation does not affect the contents of the display memory, only the way it is displayed on the screen. In a sense, the effect of this zoom operation is temporary. The numbers 0 to 15 is assigned to the zoom factors for the zoom operations of 1X to 16X.

6.3.6 Character Zoom

Function Code	_czoom
Input	_p1 = zoom factor

Description

This function sets the zoom factor for graphics character writing operation. The shape of a graphics character is defined by a box of 8 by 8 pixels. The actual size of the character is defined by this zoom factor at the time the character is being written to the display memory. The effect of this zoom operation is permanent; resetting the zoom factor to other value will not change the characters already written to the display memory.

6.3.7   Point

Function Code   _dot

Input   _p1 = x coordinate of the point
                                 _p2 = y coordinate of the point

Description

This function plots a point at location (_p1, _p2) on the display memory.

6.3.8 Cursor Move

Function Code   _move

Input   _p1 = x coordinate of the new cursor position
                        _p2 = y coordinate of the new cursor position

Description

This function moves the cursor position to a new location (_p1, _p2).

6.3.9 Rectangle Draw

Function Code   _rect

Input   _p1 = x coordinate of top left corner
                         _p2 = y coordinate of top left corner
                         _p3 = width along the X axis
                         _p4 = length along the Y axis

Description

This function draw a rectangle whose top left corner is specified by (_p1,_p2), width by _p3, and length by _p4. Notice that the top left corner has a lower Y coordinate than the bottom left corner.

6.3.10   Diamond Draw

Function Code   _dia

Input   _p1 = x coordinate of top corner
                                 _p2 = y coordinate of top corner
                                 _p3 = length of left edge
                                 _p4 = length of right edge

Description

This function draws a diamond whose top corner is at the point (_p1,_p2) and the length of the left and right edges from this corner are given by _p3 and _p4, respectively.

6.3.11   Line Draw

Function Code	_line

Input   _p1 = x coordinate of first point
                         _p2 = y coordinate of first point
                         _p3 = x coordinate of second point
                         _p4 = y coordinate of second point

Description

This function draws a line between points (_p1,_p2) and (_p3,_p4).

6.3.12   Drawing Mode

Function Code   _mode

Input   _p1 = mode (mod_rpl, mod_com, mod_reset, mod_set)

Description

This function sets the drawing mode of the GDC. The modes are defined in
gksop.h file.

6.3.13   Set Pattern

Function Code	_pattern
Input	_p1 = 16-bit pattern

Description

This function sets the pattern register with the value in _p1.

6.3.14   Arc Draw

Function Code   _arc

Input   _p1 = x coordinate of the center of curvature
                              _p2 = y coordinate of the center of curvature
                              _p3 = radius in pixels
                              _p4 = starting angle in degrees
                              _p5 = ending angle in degrees

Description

This function draw an arc whose center of curvature is at (_p1,_p2) and radius of _p3 pixels long. The starting and ending angle of the arc is given by _p4 and _p5 in degrees, respectively. The 0 degree angle is at the positive X axis, and the arcs are always drawn in counter-clockwise direction. To draw a circle, an arc from 0 to 360 degrees is drawn.

6.3.15 Set Area Fill Pattern

Function Code   _sfill

Input   8 by 8 bit pattern in _p1 , _p2, _p3, and _p4

Description

This function loads the 8 bytes of parameter RAM in the GDC for future uses in area-fill operation. The eight bytes of fill pattern is indicated through the four parameters _p1, _p2, _p3, and _p4, as shown in Figure 6-2. This command does not actually write the pattern to the display memory; only to the GDC. The **draw_fill** function, function code _dfill, does.
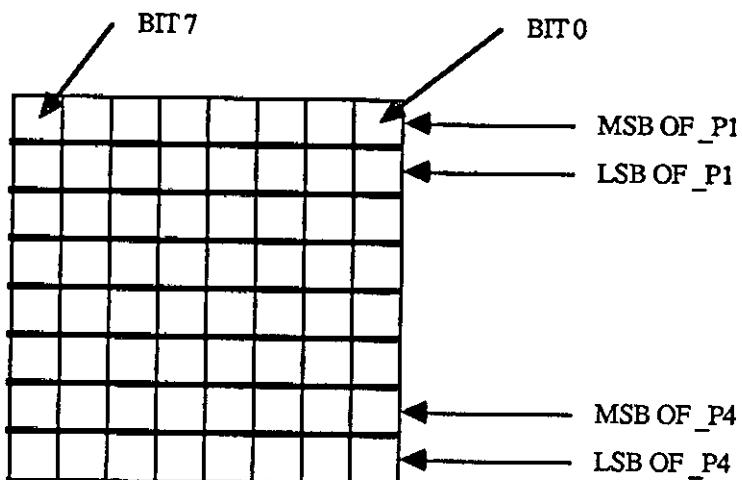
Figure 6-2. The organization of the area fill pattern.

6.3.16 Draw Area Fill

Function Code   _dfill

Input   _p1 = x coordinate of top left corner
                          _p2 = y coordinate of top left corner
                          _p3 = x coordinate of bottom right corner
                          _p4 = y coordinate of bottom right corner
                          _p5 = 0x80 for slanted area-fill; 0 otherwise

Description

This function fills the area bounded by the rectangle specified by _p1, _p2, _p3, and _p4. The pattern in which to fill the area should be specified by the Set Area Fill Pattern function before this function is invoked. Note that this command is used to write graphics characters to the display memory. The slanted area-fill can be used to write italicized characters.

6.3.17 Set Character Height

Function Code   _ccc

Input   _p1 = height of the coded character

Description

This function sets the height of the coded character. The minimum possible height of the character is 8 pixels, and the maximum is 16 pixels. A blinking block cursor is displayed at the character position. The cursor is programmed to be 7 pixels tall, 8 pixels wide, and has the blinking rate of 1 Hz with a 3/4 on-1/4 off duty cycle.

6.3.18 Set Display Area Partition

Function Code	_part1 (for area 1), _part2 (for area 2)
Input	_p1 = starting address of the partition (either 1 or 2) _p2 = length of the partition (the number of lines) _p3 = type (0 for coded character, 1 for graphics)

Description

This function sets the location and the size of a partition of the display area. If the screen is not partitioned, the entire screen will display the first display area. The display partition can only be at a word boundary. This function is used to move the screen around the display area in both horizontal and vertical directions.

6.3.19 DMA Write

Function Code	_dmaw
Input	_p1 = x coordinate of top left corner
	_p2 = y coordinate of top left corner
	_p3 = width of the bounding box (in words)
	_p4 = length of the bounding box (in lines)
	_p5 = starting address of the system memory

Description

This function moves the contents of the system memory, whose starting address is specified by _p5, to the rectangular region in the display memory whose boundary is specified by _p1, _p2, _p3, and _p4. The coordinate (_p1,_p2) specifies the top left corner of the box, _p3 specifies the width of the box in words, and _p4 specifies the length of the box in lines. The bounding box should be positioned on word boundaries only. In moving the byte-sized data from the system memory to the word-sized display memory, the lower addressed byte is packed to the LSB of a word.

6.3.20 DMA Read

Function Code	_dmar

Input

_p1:	x coordinate of top left corner,
_p2:	y coordinate of top left corner,
_p3:	width of the bounding box in words,
_p4:	length of the bounding box in lines,
_p5:	starting address in system memory.

Description

This function moves the contents of a rectangular box, defined by _p1, _p2, _p3, and _p4 parameters, to the system memory starting from the location specified by _p5. The LSB of a word in display memory is moved to a lower addressed location than the MSB.

Chapter 7

CONCLUSION

Looking back at the project, many things pop up in my mind; the feeling of excitement, awe, joy, impatience, despair, and more joy and despair. It would be impossible, certainly inappropriate, to describe every experience encountered during the project. However, a few are presented here for the benefit of those who, undoubtedly, will run into similar experiences, doing similar work.

7.1 Mistakes

The biggest mistake made during the project was the decision to use the SCALDsystem, and subsequently the Merlyn-PCB, CAD system. At the time of the design, it appeared to be a good idea to learn to use the system. Its user-friendly Graphics Editor, Timing Verifier, Logic Simulator, and Packager all looked fascinating, wonderous. However, after about six months of use, still trying to learn and figure out error messages, I realized that there was much more to be learned than it initially appeared. Much time was spent on trying to build the custom library for the devices which were not in the existing libraries. The most difficult problem was defining timing and simulation models for the devices, without which timing verification and logic simulation of the entire circuit are impossible. After much tinkering with the models, the idea of full simulation was abandoned, and an alternative solution was devised. Because of relative simplicity of the circuits involved, the partial verification and simulation was acceptable. Despite all the troubles, a valuable experience was gained.

After spending much time with the SCALDsystem, it did not seem to be much more trouble to use the Merlyn-PCB to complete the placement and routing of the GDC board. It even seemed wasteful, and wrong, to have spent so much time designing the GDC board and not completing it with the Merlyn-PCB. With the current configuration of 2 Mega-words of memory in the TRAC Vax, a session with the Merlyn-PCB was a test of patience and endurance. Each interaction with it was measurable in minutes, not in seconds. Using the User Manual that is not worth even the paper it is printed on, I do not wish to use it ever again.

The second mistake was using the edge card connector as the host interface. With a little more thought, a 40-pin dip connector should have been used. With it, the interface signals could be brought to the board with a 40-pin flat cable. This was eventually implemented on the GDC board. Incidentally, the decision to cover the remainder of the board with plated holes was a wise one, for the holes provide room for modifications and expansions.

The third mistake is connecting an inverter to the input of a PAL, shown in page 5 of the hardware diagram. I do not remember how that happened, but the mistake was detected after the GDC board was made.

7.2 Indispensable Tools

The testing of the GDC board was not possible without the TRACE program and the Tektronix 9100 Series DAS . Since the OS-9 for the Color Computer does not have a source code debugger, the device driver was compiled to generate the assembly listing, which was used with the TRACE program for debugging [LIPO 82]. Although the compiler generated quite efficient assembly codes, debugging was difficult due to

the pure size of the codes. The TRACE program proved to be the most valuable debugging tool. A careful study of the compiler also helped.
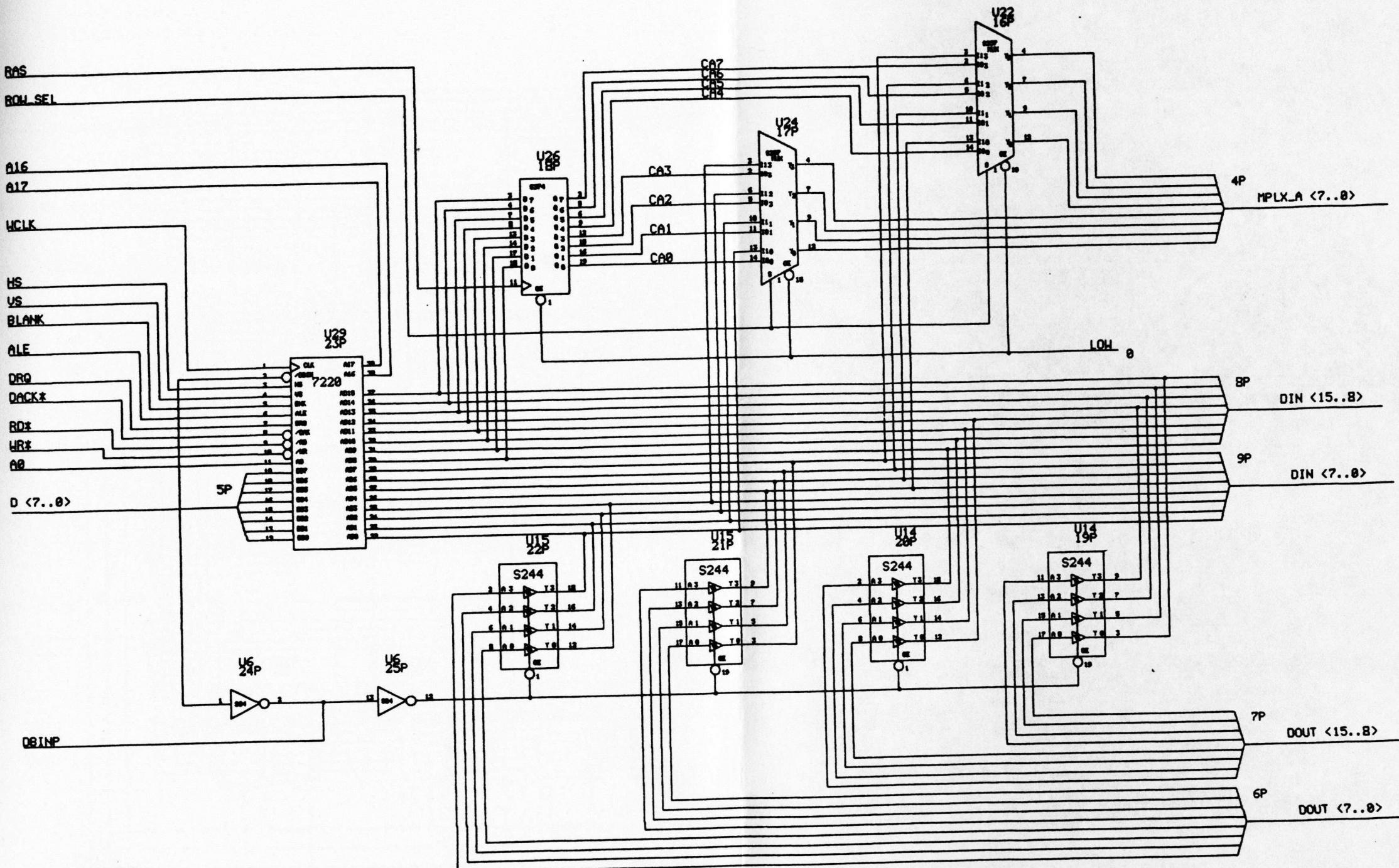
For hardware testing, the DAS was indispensable. It proved to be a very powerful and, yet, easy-to-use tool. With it, it was possible to measure the precise timing of the signals in the system. Although no hardware design errors were discovered, the DAS made it possible to feel confident about the hardware.

7.3 Suggestions for Further Work

It may be desirable to add a light pen to the system since it has no graphical input device. Since the GDC has the provision for it, and is capable of detecting and deglitching the light pulse, the required hardware addition would be simple. A composite video monitor interface may be desirable. Aside from that, there is not much left to be done on the hardware of the system; most of the desired functions are implemented in the GDC board for monochrome monitor interface. It may be desirable to modify the host interface for more powerful host processors, since the 6809 is somewhat small, and slow, for the host processor. An ideal choice would be a 68000 based computer system running OS-9. Currently there are three such systems; Atari's ST, Apple's Macintosh, and Commodore 's Amiga.

What is needed more is the convenience and the friendliness of a higher level graphics package. More functions can be added to the current device driver with ease. The way it is set up, adding a new function is just a matter of adding a case statement and the function itself. Although doubtful on the Color Computer, the Graphical Kernel System (GKS) can be implemented, taking advantage of the figure drawing functions. The GKS and OS-9 is a good combination, for both use the concept of device independent i/o and have similar internal structures [ENDE 84].

APPENDIX A

HARDWARE SCHEMATIC
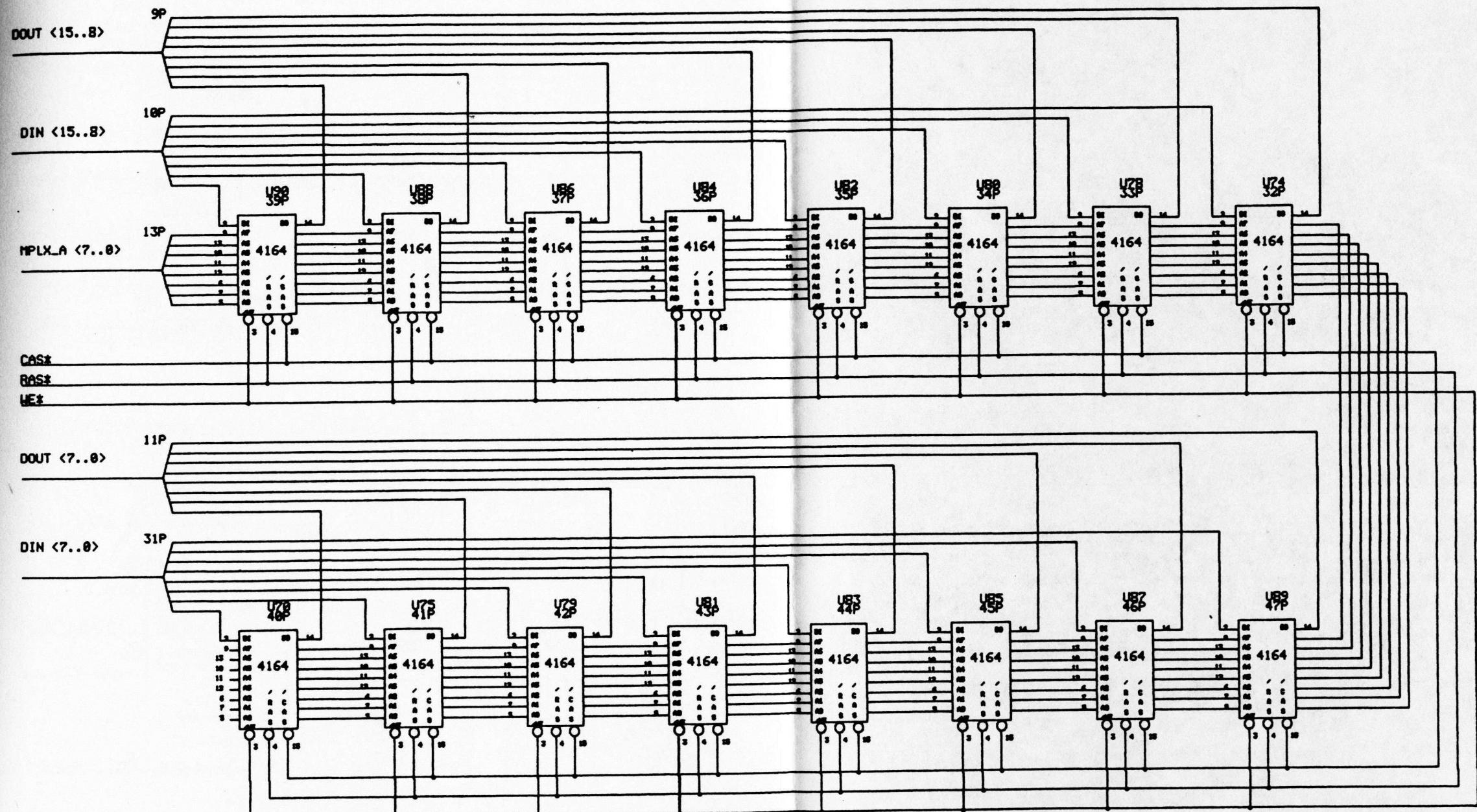
MPLX_A <7..0>

DIN <15..8>

DIN <7..0>

DOUT <15..8>

DOUT <7..0>

TITLE:	DATE:
GDC.LOGIC.1	3-29-85
ENGINEER:	PAGE:
SPS	1 OF 6

DOUT <15..8>   9P

DIN <15..8>   18P

MPLX_A <7..0>   13P

CAS*
RAS*
WE*

DOUT <7..0>   11P

DIN <7..0>   31P
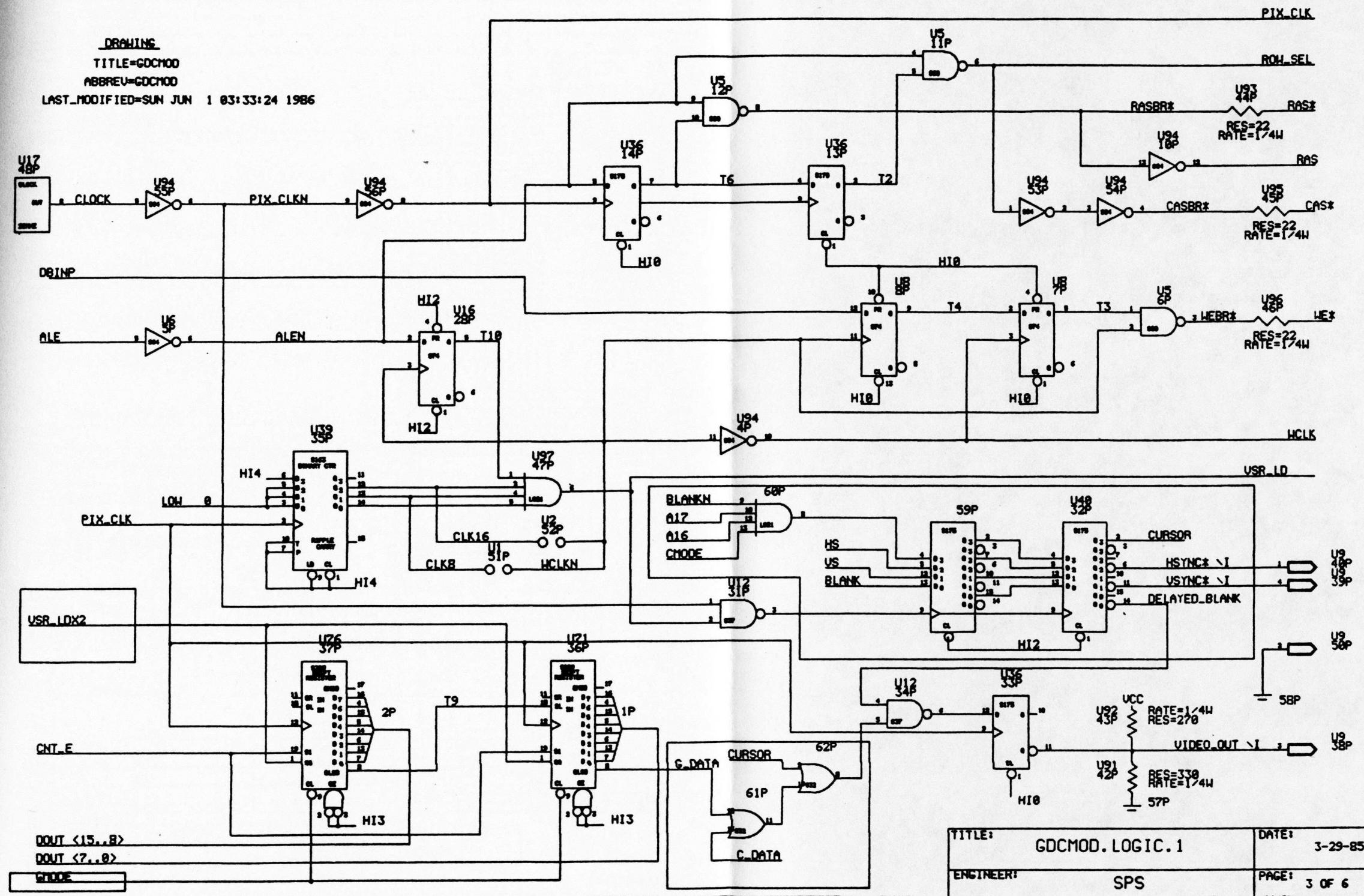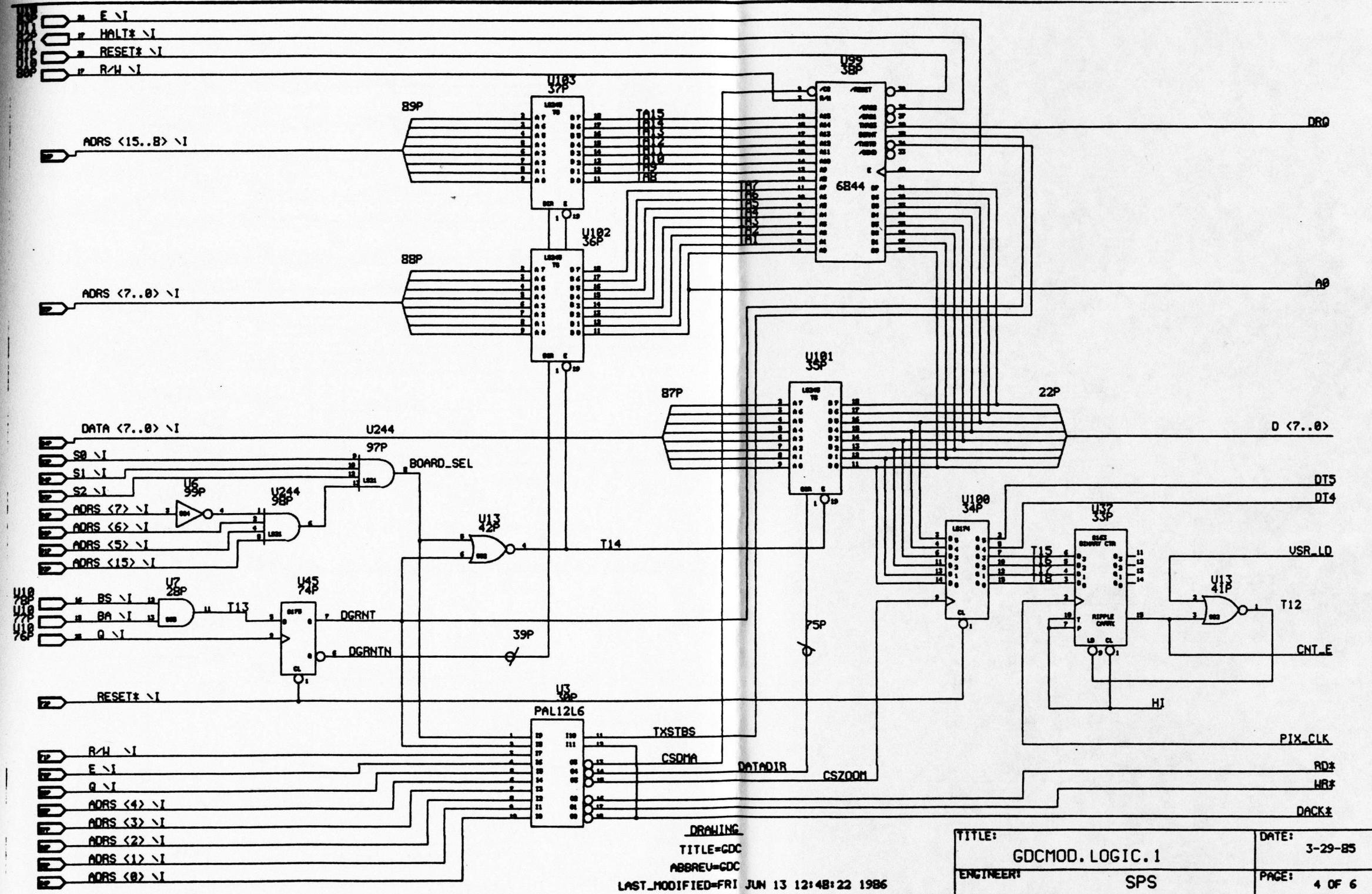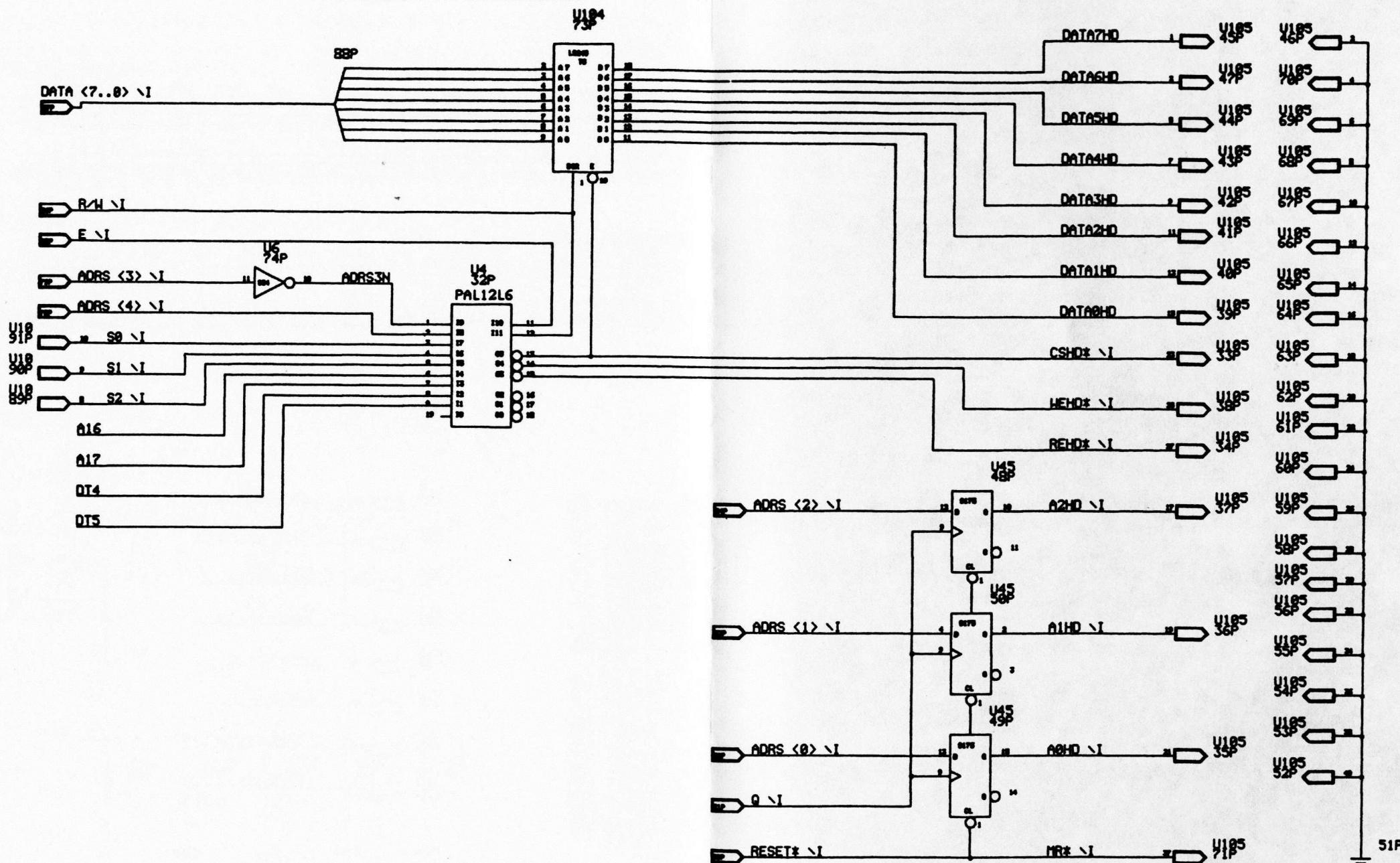
DRAWING
TITLE=GDC
ABBREV=GDC
LAST_MODIFIED=NOT WRITTEN

TITLE:	DATE:
GDC.LOGIC.1	1-13-85
ENGINEER: SPS	PAGE: 2 OF 6

DRAWING
TITLE=GDCMOD
ABBREV=GDCMOD
LAST_MODIFIED=SUN JUN 1 03:33:24 1986
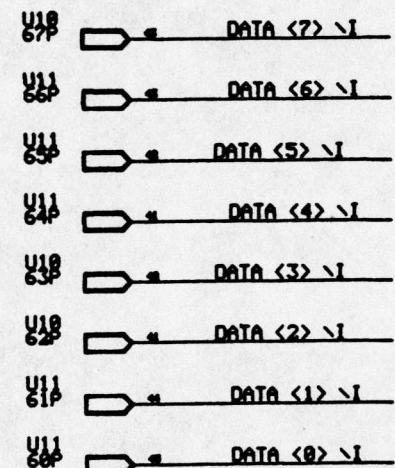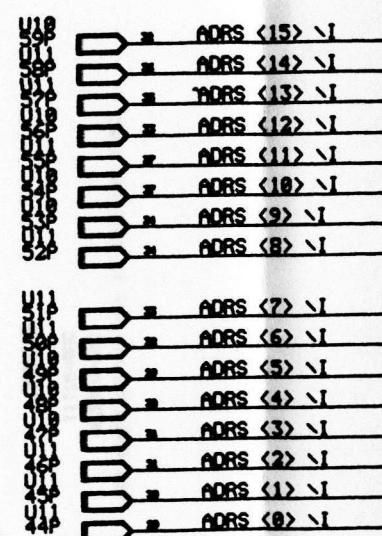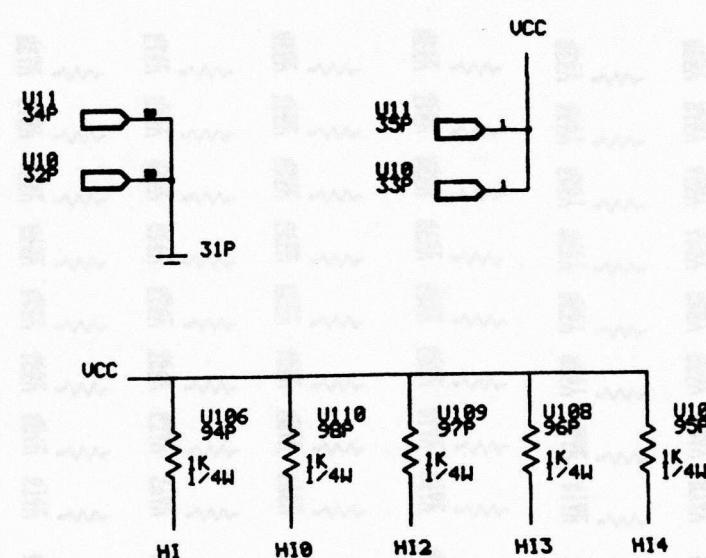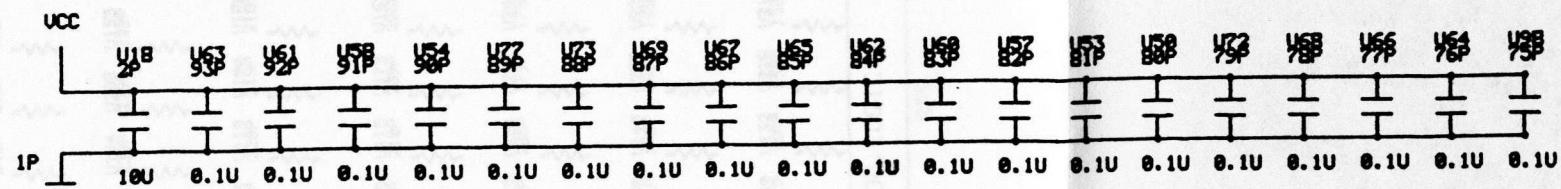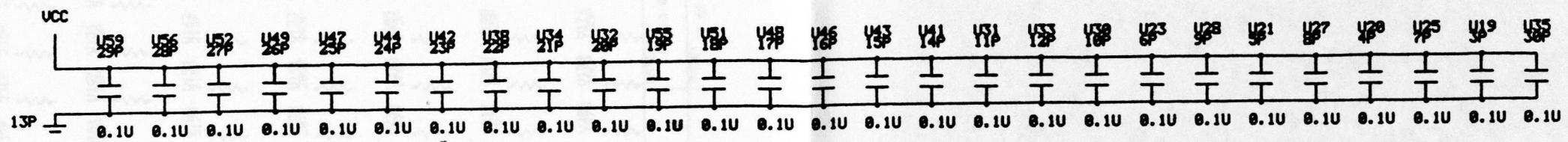
TITLE: GDCMOD.LOGIC.1
DATE: 3-29-85
ENGINEER: SPS
PAGE: 3 OF 6

TITLE: GDC.LOGIC.1	DATE: 6-10-85
ENGINEER: SPS	PAGE: 5 OF 6

VCC

U59 25P	U56 28P	U52 27P	U49 26P	U47 25P	U44 24P	U42 23P	U38 22P	U34 21P	U32 20P	U55 19P	U51 18P	U48 17P	U46 16P	U43 15P	U41 14P	U51 11P	U53 12P	U38 10P	U23 6P	U28 9P	U21 3P	U27 4P	U20 7P	U25 5P	U19 2P	U55 38P

13P

0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U

VCC

U18 2P	U63 93P	U61 92P	U58 91P	U54 96P	U57 85P	U23 89P	U69 88P	U67 87P	U65 86P	U62 85P	U48 84P	U57 83P	U81 82P	U58 89P	U52 79P	U68 78P	U56 77P	U64 76P	U58 75P

1P

10U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U 0.1U

VCC

U11 34P

U18 32P

31P

U11 35P

U11 33P

VCC

U106 34P	U110 96P	U109 97P	U108 96P	U107 95P
1K 1/4W	1K 1/4W	1K 1/4W	1K 1/4W	1K 1/4W

HI   HI8   HI2   HI3   HI4

U18 98P — ADRS <15> \I
U11 59P — ADRS <14> \I
U11 57P — ADRS <13> \I
U18 56P — ADRS <12> \I
U11 55P — ADRS <11> \I
U18 54P — ADRS <10> \I
U18 53P — ADRS <9> \I
U18 52P — ADRS <8> \I

U11 51P — ADRS <7> \I
U11 50P — ADRS <6> \I
U18 49P — ADRS <5> \I
U18 48P — ADRS <4> \I
U18 47P — ADRS <3> \I
U18 46P — ADRS <2> \I
U18 45P — ADRS <1> \I
U11 44P — ADRS <0> \I

U18 67P — DATA <7> \I
U11 66P — DATA <6> \I
U11 65P — DATA <5> \I
U11 64P — DATA <4> \I
U18 63P — DATA <3> \I
U18 62P — DATA <2> \I
U11 61P — DATA <1> \I
U11 60P — DATA <0> \I

153 1 OHM, 0 WATT RESISTORS FOR 153 SINGLE-IN-LINE RESISTOR NETWORKS

TITLE:	GDC.LOGIC.1	DATE: 9-13-85
ENGINEER:	SPS	PAGE: 7 OF 8

DRAWING

TITLE=GDC
ABBREV=GDC
LAST_MODIFIED=THU JUL 17 02:13:53 1985

RES=1
RATE=0

HI2
0
HCLK

HS
BLANK

1P
3P

A17
GMODE
CMODE
HI2
2P
DELAY_10CLK

HI7
6P
5P
4P
VSR_LD
VSR_LDX2
HI7

DOUT <15..8>
9P
16P
20P
27256
17P
VSR_LD
18P
HI2
15P
21P
DOUT <7..0>
10P
19P
VSR_LD
GMODE
HI2
ALE

8P
7P
PIX_CLK
HI2
VSR_LD
C_DATA
CMODE
HI2

BLANK
12P
BLANKN
13P
14P
22P
CMODE
CMODE
DELAY_10CLK
11P
A16

APPENDIX B

HOST INTERFACE PAL LISTING

```
PAL12L6
PAL FOR GRAPHICS WORKSTATION
PETER SONG 4/10/85

BS DGRNT RW E Q A4 A3 A2 A1 GND
/TXSTB /DAK /CS6844 /DATADIR /CSZOOM /RD /WR /DACK A0 VCC

DACK    = TXSTB * Q    +    TXSTB * E
WR      = BS * A4 * A3 * /A2 * /A1 * E * /RW    +    DAK * RW
RD      = BS * A4 * A3 * /A2 * /A1 * E * RW    +    DAK * /RW
CSZOOM  = BS * A4 * A3 * /A2 * A1 * /A0 * E * Q * /RW
DATADIR = DGRNT * /RW    +    /DGRNT * RW
CS6844  = BS * /A4    +    BS * A4 * /A3


FUNCTION TABLE

BS A4 A3 A2 A1 A0 E Q RW DGRNT /TXSTB /DAK
/DACK /WR /RD /CSZOOM /DATADIR /CS6844

:-------INPUTS----------        --OUTPUTS--
1B A A A A A E Q R D / /        / / / / / /
1S 4 3 2 1 0     W G T D        D W R C D C
:               R X A           A R D S A S
:               N S K           C     Z T 6
:               T T             K     0 A 8
:               B                     0 D 4
:                                     M 1 4
:                                     R
-----------------------------------------------------------------
H X X X X X X X H L H H         H H H H L L   READ DMA CONTROLLER
H X X X X X X X L L H H         H H H H H L   WRITE DMA CONTROLLER
H H H L L X H X L L H H         H L H H H H   NON-DMA WRITE GDC
H H H L L X H X H L H H         H H L H L H   NON-DMA READ GDC
H H H L H L H L H H L L H H     H H H L H H   CLOCK/WRITE ZOOM PRE-SCALER
-----------------------------------------------------------------


DESCRIPTION

DACK = DMA acknlwledge to GDC.
WR = low when writing to $(BS + 18) or $(BS + 19).
RD = low when reading from $(BS + 18) or $(BS + 19).
CSZOOM = low during third quarter of E when writing to $(BS + 1A).
DATADIR = low during non-DMA read or DMA write.
CS6844 = low when referencing address from $(BS + 0) to $(BS + 17).
```
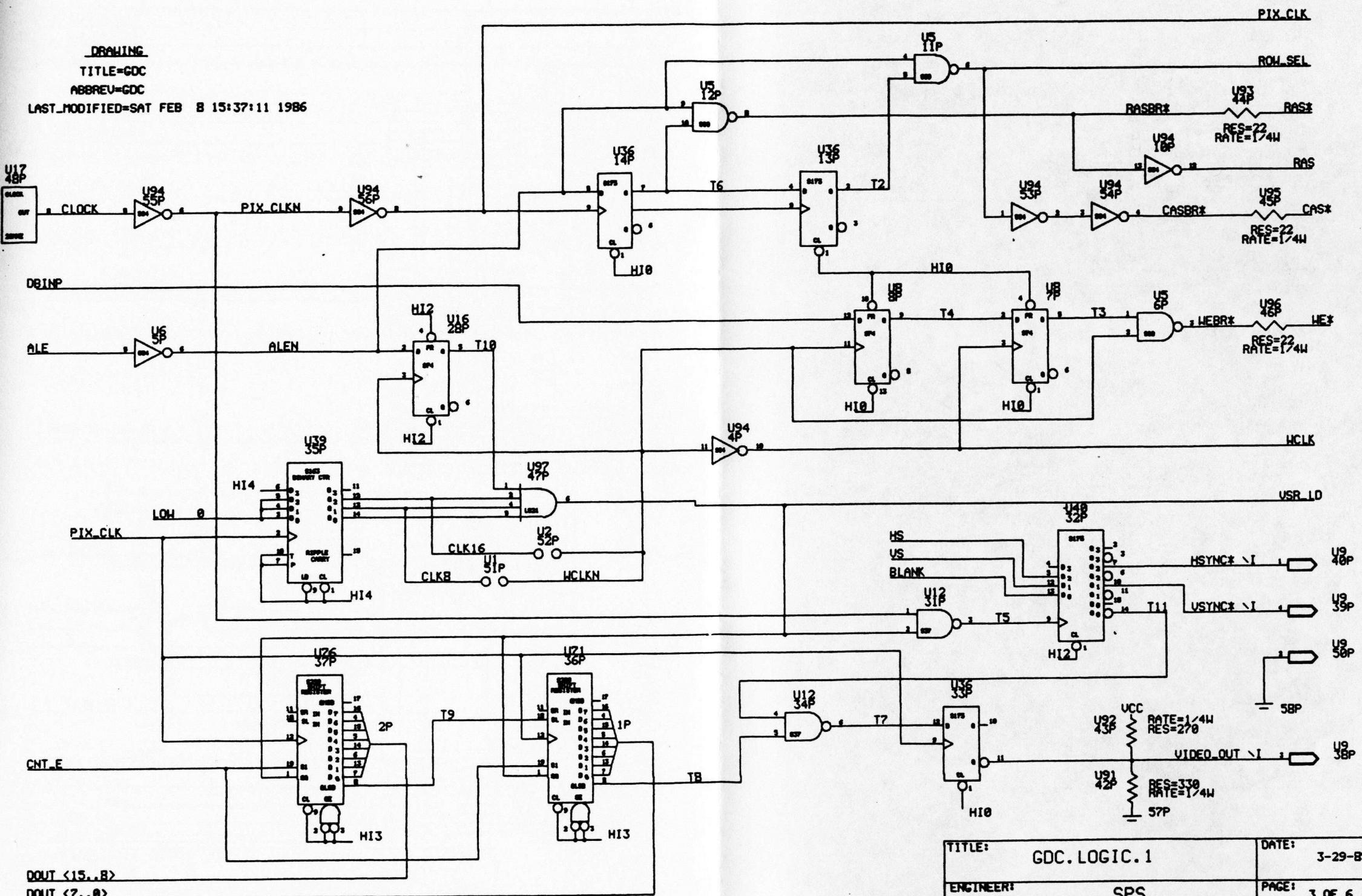
124
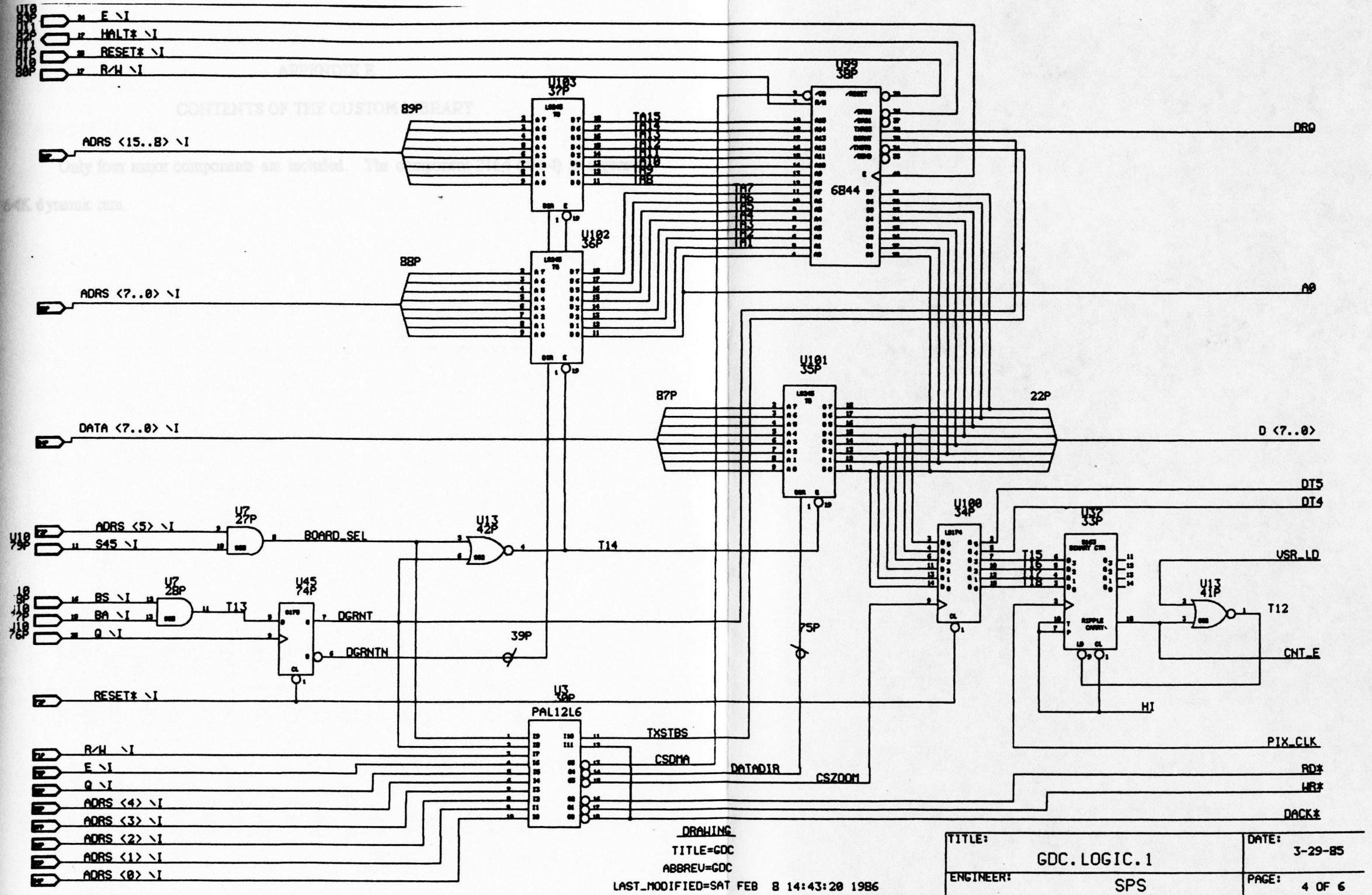
APPENDIX C

VIDEO TIMING CALCULATION

RESOLUTION	800 by 521, interlaced
VIDEO RATE	16 MHz.
PIXEL TIME	62.5 nano-seconds

HORIZONTAL TIMING

ACTIVE LINE	50 words, for 50 micro-seconds
FRONT PORCH	3 words, for 3 micro-seconds
SYNC	5 words, for 5 micro-seconds
BACK PORCH	5 words, for 5 micro-seconds
LINE TIME	63 words, for 63 micro-seconds
LINE RATE	15.873 KHz.

VERTICAL TIMING

ACTIVE FRAME	260 lines, for 16.38 mili-seconds
FRONT PORCH	1 line, for 63 micro-seconds
SYNC	3 lines, for 189 micro-seconds
BACK PORCH	1 line, for 63 micro-seconds
VERTICAL SCAN TIME	265 lines, for 16.695 mili-seconds
VERTICAL SCAN RATE	59.89817 Hz.
FRAME RATE	29.94908 Hz.

APPENDIX D

ORIGINAL DESIGN SCHEMATIC

This is the original design of the board, from which the printed circuit board is made.

GDC.LOGIC.1

DATE: 3-29-85

ENGINEER: SPS

PAGE: 3 OF 6

DRAWING
TITLE=GDC
ABBREV=GDC
LAST_MODIFIED=SAT FEB 8 14:43:20 1986

TITLE:		DATE:
	GDC.LOGIC.1	3-29-85
ENGINEER:	SPS	PAGE: 4 OF 6
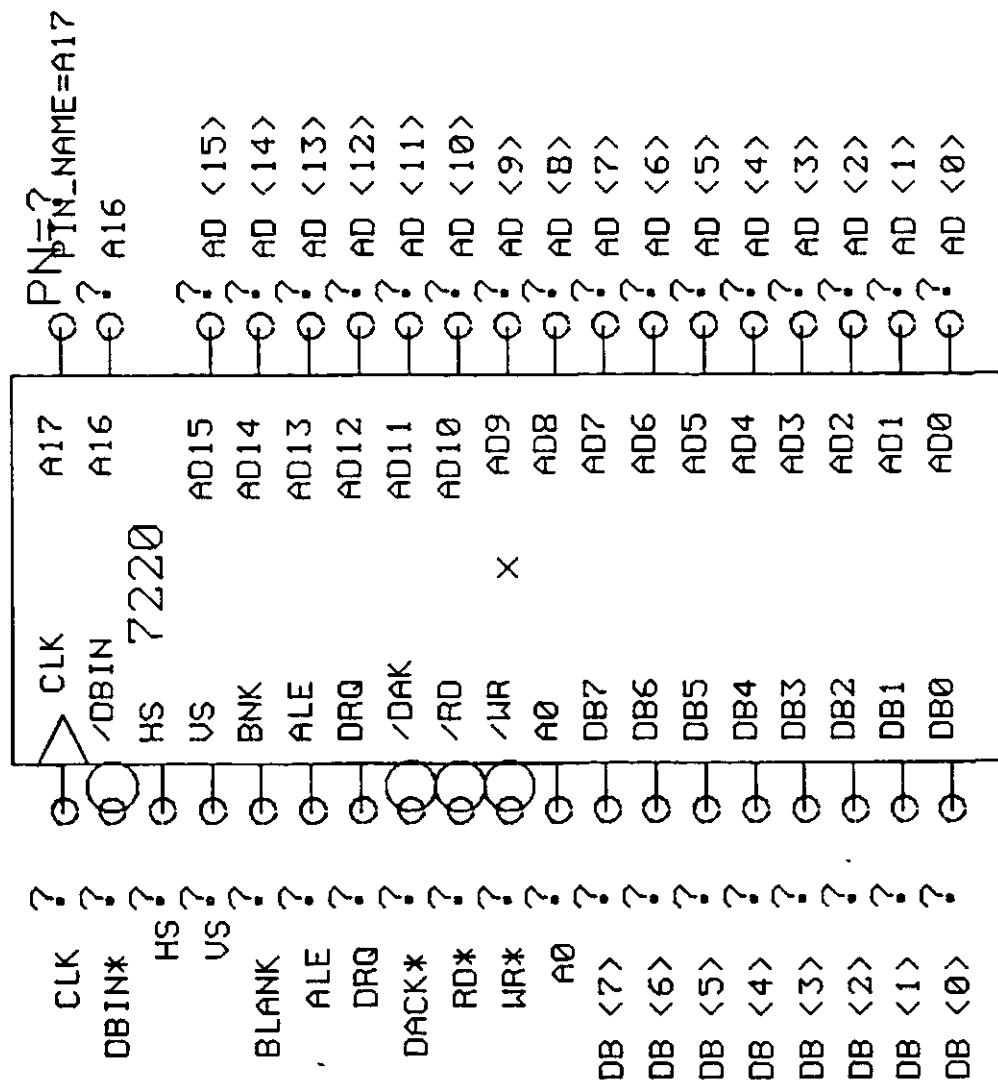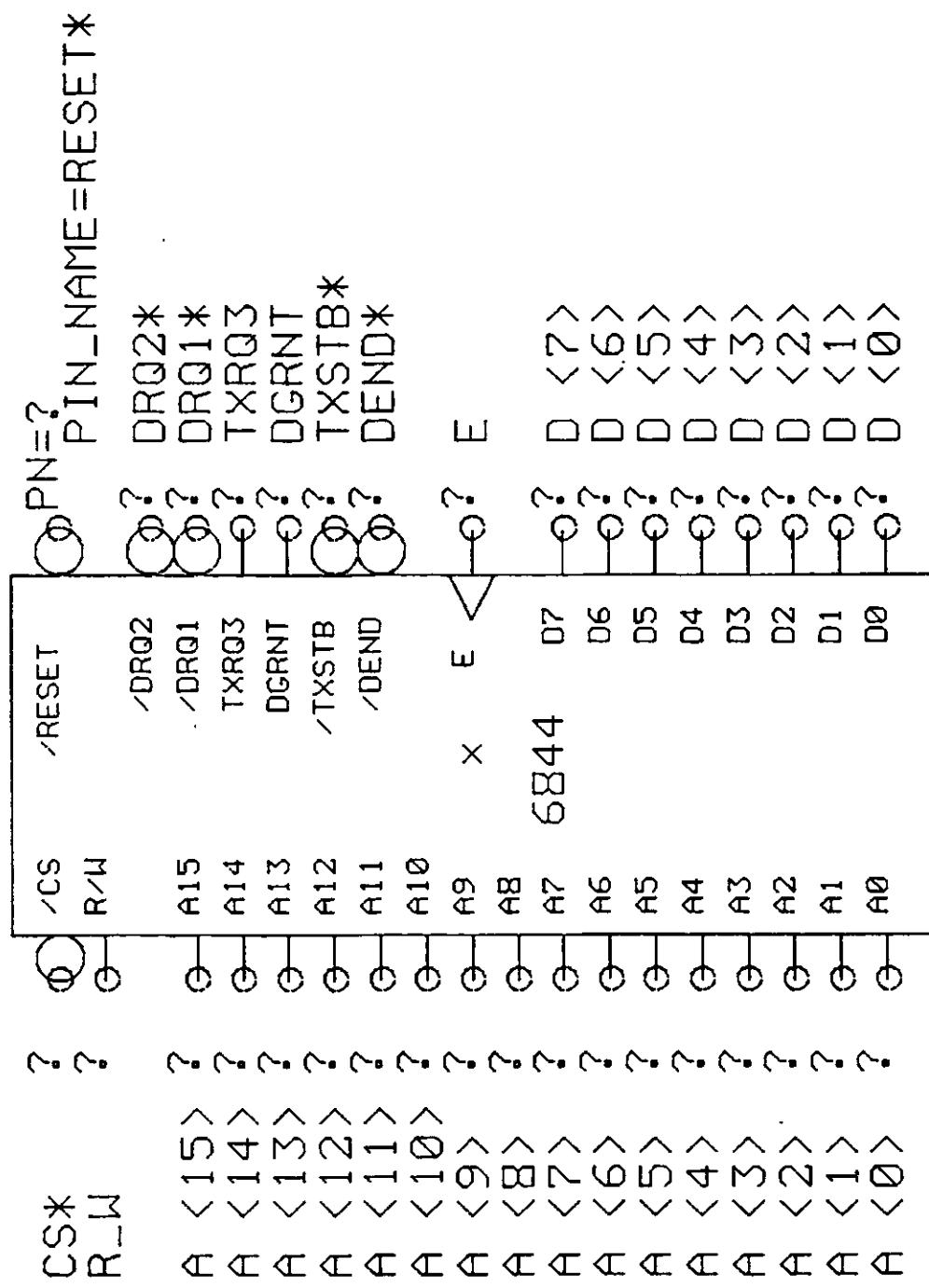
APPENDIX E

CONTENTS OF THE CUSTOM LIBRARY

Only four major components are included. The component 2164 (4164) is a generic 64K dynamic ram.

PN=?
PIN_NAME=A17

7220

Signal	Pin	Pin	Signal
CLK	CLK	A17	A17
DBIN*	/DBIN	A16	A16
HS	HS	AD15	AD <15>
VS	VS	AD14	AD <14>
BLANK	BNK	AD13	AD <13>
ALE	ALE	AD12	AD <12>
DRQ	DRQ	AD11	AD <11>
DACK*	/DAK	AD10	AD <10>
RD*	/RD	AD9	AD <9>
WR*	/WR	AD8	AD <8>
A0	A0	AD7	AD <7>
DB <7>	DB7	AD6	AD <6>
DB <6>	DB6	AD5	AD <5>
DB <5>	DB5	AD4	AD <4>
DB <4>	DB4	AD3	AD <3>
DB <3>	DB3	AD2	AD <2>
DB <2>	DB2	AD1	AD <1>
DB <1>	DB1	AD0	AD <0>
DB <0>	DB0		

PN=?
PIN_NAME=RESET*

DRQ2*
DRQ1*
TXRQ3
DGRNT
TXSTB*
DEND*

E

D<7>
D<6>
D<5>
D<4>
D<3>
D<2>
D<1>
D<0>

/CS   /RESET
R/W

A15   /DRQ2
A14   /DRQ1
A13   TXRQ3
A12   DGRNT
A11   /TXSTB
A10   /DEND

A9   E
A8
A7   D7
A6   D6
A5   D5
A4   D4
A3   D3
A2   D2
A1   D1
A0   D0

X

6844

CS*
R_W

A<15>
A<14>
A<13>
A<12>
A<11>
A<10>
A<9>
A<8>
A<7>
A<6>
A<5>
A<4>
A<3>
A<2>
A<1>
A<0>

BUBBLE_GROUP=(IN11)
BUBBLE_GROUP=(IN10)
BUBBLE_GROUP=(IN9)
BUBBLE_GROUP=(IN8)
BUBBLE_GROUP=(IN7)
BUBBLE_GROUP=(IN6)

BUBBLE_GROUP=(IN3)
BUBBLE_GROUP=(IN0)
BUBBLE_GROUP=(IN5)
BUBBLE_GROUP=(IN4)
BUBBLE_GROUP=(IN1)
BUBBLE_GROUP=(IN2)

PAL12L6

I10 ?IN10
I11 ?IN11

O5 ?OUT5
O4 ?OUT4
O3 ?OUT3

O2 ?OUT2
O1 ?OUT1
O0 ?OUT0

I9   IN9?
I8   IN8?
I7   IN7?
I6   IN6?
I5   IN5?
I4   IN4?
I3   IN3?
I2   IN2?
I1   IN1?
I0   IN0?

×

DIN <SIZE-1..0>

A <7>
A <6>
A <5>
A <4>
A <3>
A <2>
A <1>
A <0>

PN=?
DOUT <SIZE-1..0>
PIN_NAME=A <7>
A <6>
A <5>
A <4>
A <3>
A <2>
A <1>
A <0>

DI   DO
A7
A6
A5   2164
A4   x
A3   /
A2   R
A1   A
A0   S
/WE

WE*   RAS*   CAS*

NEED_NO_SIZE=TRUE

1P

PIN_NUMBER=(39)

INPUT_LOAD=(-0.01,0.01)

(38) OUTPUT_LOAD=(0.01,-0.01)

OUTPUT_LOAD=(0.01,-0.01)

BIDIRECTIONAL=TRUE

(37) INPUT_LOAD=(-0.01,0.01)

7220

IDIRECTIONAL=TRUE   OUTPUT_LOAD=(0.01,-0.01)

OUTPUT_TYPE=(TS,TS)

OUTPUT_TYPE=(TS,TS)

POWER_PINS=(VCC:40;GND:20)
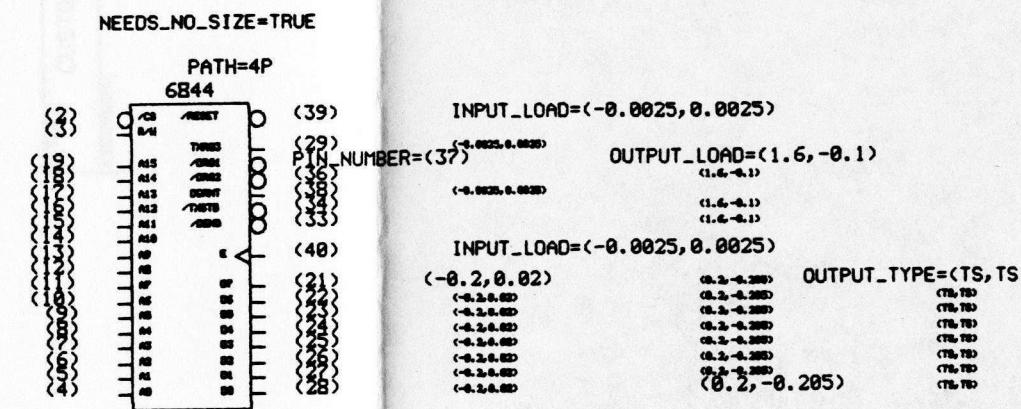
FAMILY=NEC

DRAWING

ABBREV=CUSTOM

TITLE=CUSTOM LIBRARY

LAST_MODIFIED=WED JUL 23 03:29:18 1986

DEFINE

X_FIRST=0

X_STEP=SIZE

TITLE: CUSTOM LIBRARY	DATE: 12/5/84
ENGINEER: PETER SONG	PAGE: 1 OF 5

NEEDS_NO_SIZE=TRUE

PATH=4P
6844

BIDIRECTIONAL=TRUE

INPUT_LOAD=(-0.2,0.02)

OUTPUT_LOAD=(0.2,-0.145)

(-0.2,0.02)

OUTPUT_TYPE=(TS,TS)

INPUT_LOAD=(-0.0025,0.0025)

PIN_NUMBER=(37)

OUTPUT_LOAD=(1.6,-0.1)

INPUT_LOAD=(-0.0025,0.0025)

(-0.2,0.02)

OUTPUT_TYPE=(TS,TS)

(0.2,-0.205)

BIDIRECTIONAL=TRUE

POWER_PINS=(VCC:20;GND:1)
FAMILY=MOTOROLA

DRAWING
ABBREV=CUSTOM
TITLE=CUSTOM LIBRARY
LAST_MODIFIED=WED JUL 23 03:20:44 1986

TITLE:	CUSTOM LIBRARY	DATE: 12/5/84
ENGINEER:	PETER SONG	PAGE: 2 OF 5

NEEDS_NO_SIZE=TRUE

PIN_NUMBER=(14)
OUTPUT_TYPE=(TS,TS)
CHECK_LOADS=(-0.01,0.10)
BOTTOM_LOAD=(50.0,-50.0)

4164

(15)

(4)

INPUT_LOAD=(-0.01,0.10)
INPUT_LOAD=(-0.01,0.10)

PIN_NUMBER=(3)

POWER_PINS=(VCC:8;GND:16)
PART_TYPE=INTEL

PIN_NUMBER=(3)

INPUT_LOAD=(-0.01,0.10)

DRAWING
ABBREV=CUSTOM
TITLE=CUSTOM LIBRARY
LAST_MODIFIED=WED JUL 23 03:05:04 1986

NEEDS_NO_SIZE=TRUE

PAL1216

PIN_NUMBER=(11)

FAMILY=TTL
POWER_PINS=(VCC:20;GND:10)

INPUT_LOAD=(-0.25,0.025)
(-0.25,0.025)
OUTPUT_LOAD=(8.0,-3.2)

DRAWING
ABBREV=CUSTOM
TITLE=CUSTOM LIBRARY
LAST_MODIFIED=WED JUL 23 02:59:24 1986

| TITLE: | CUSTOM LIBRARY | DATE: | 12/5/84 |
| ENGINEER: | PETER SONG | PAGE: | 4 OF 5 |
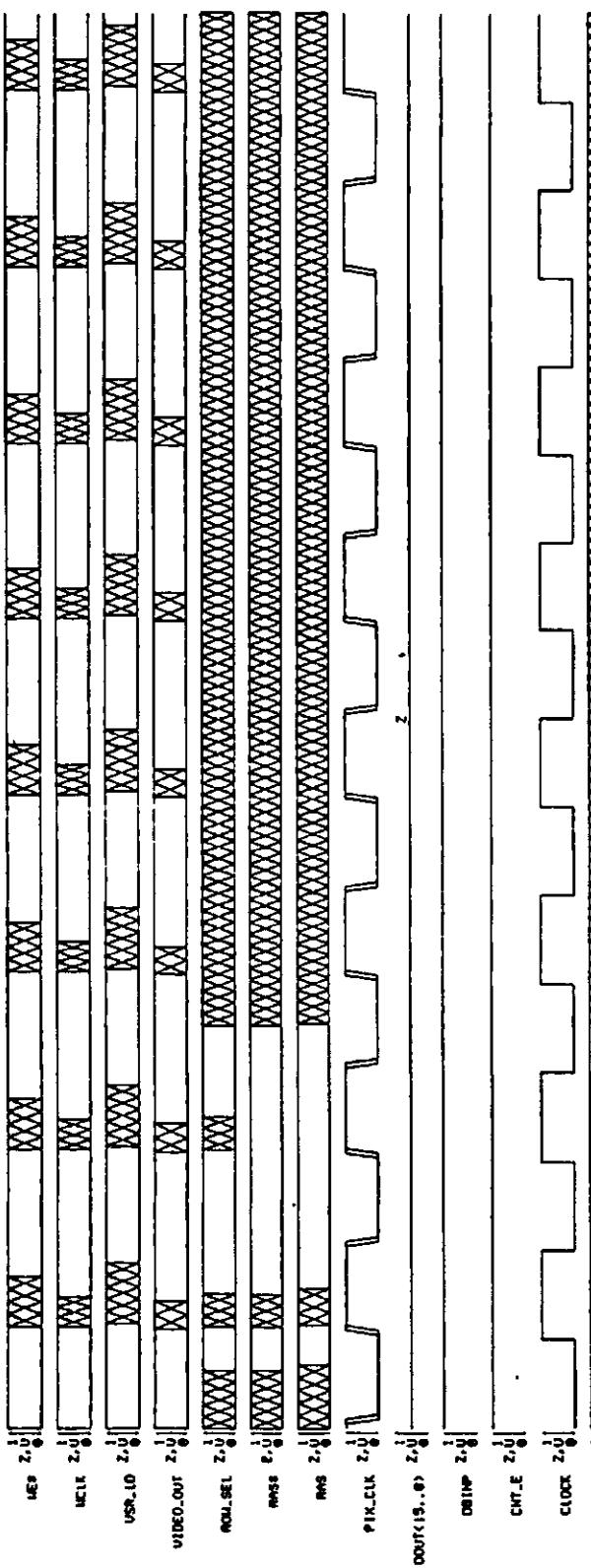
APPENDIX E
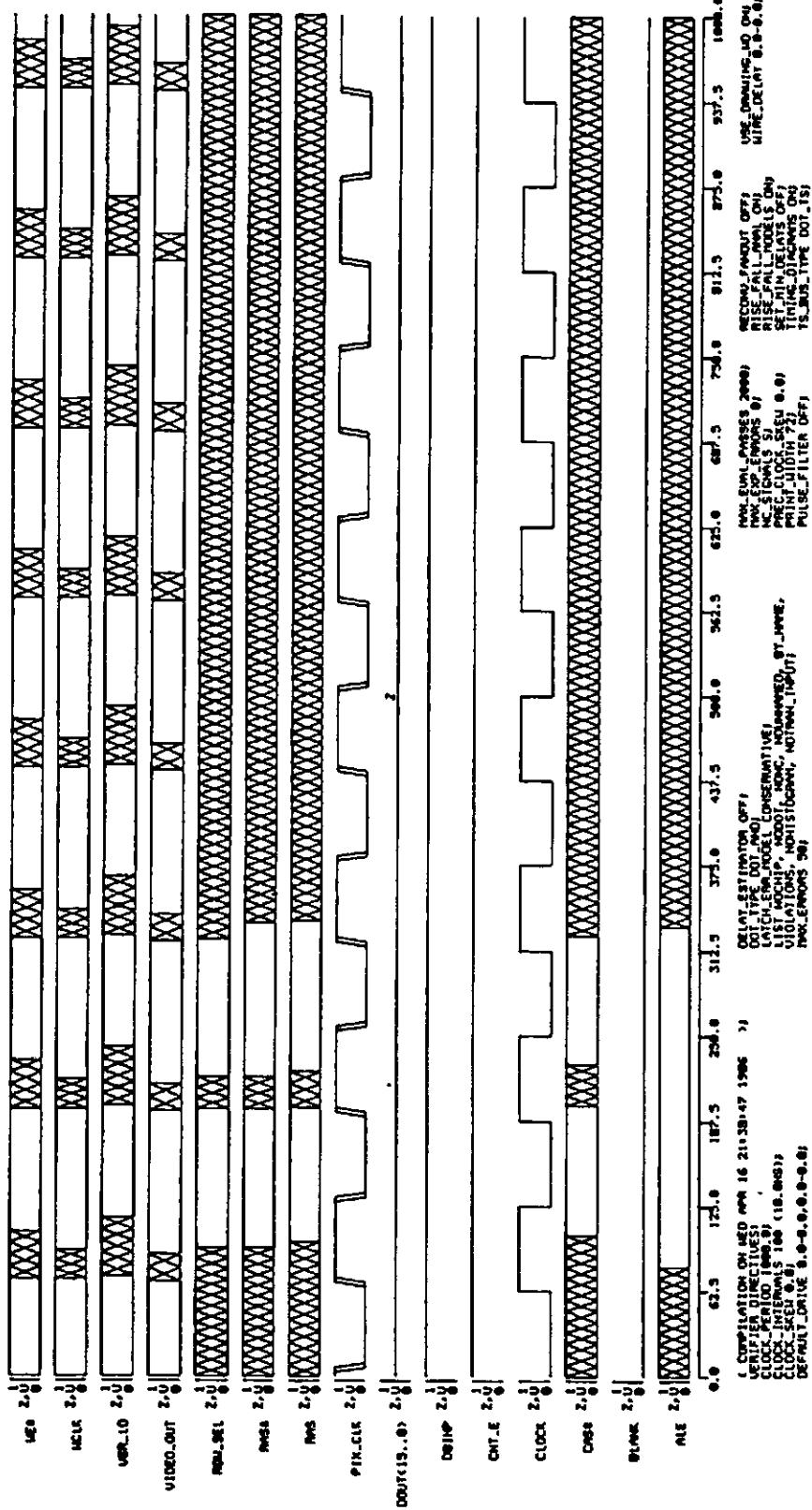
A RESULT OF TIMING VERIFIER RUN

The result of the timing verification on the display memory control circuitry is presented. The resulting waveforms of the timing verification on other portions of the design is not included because they convey very little information.
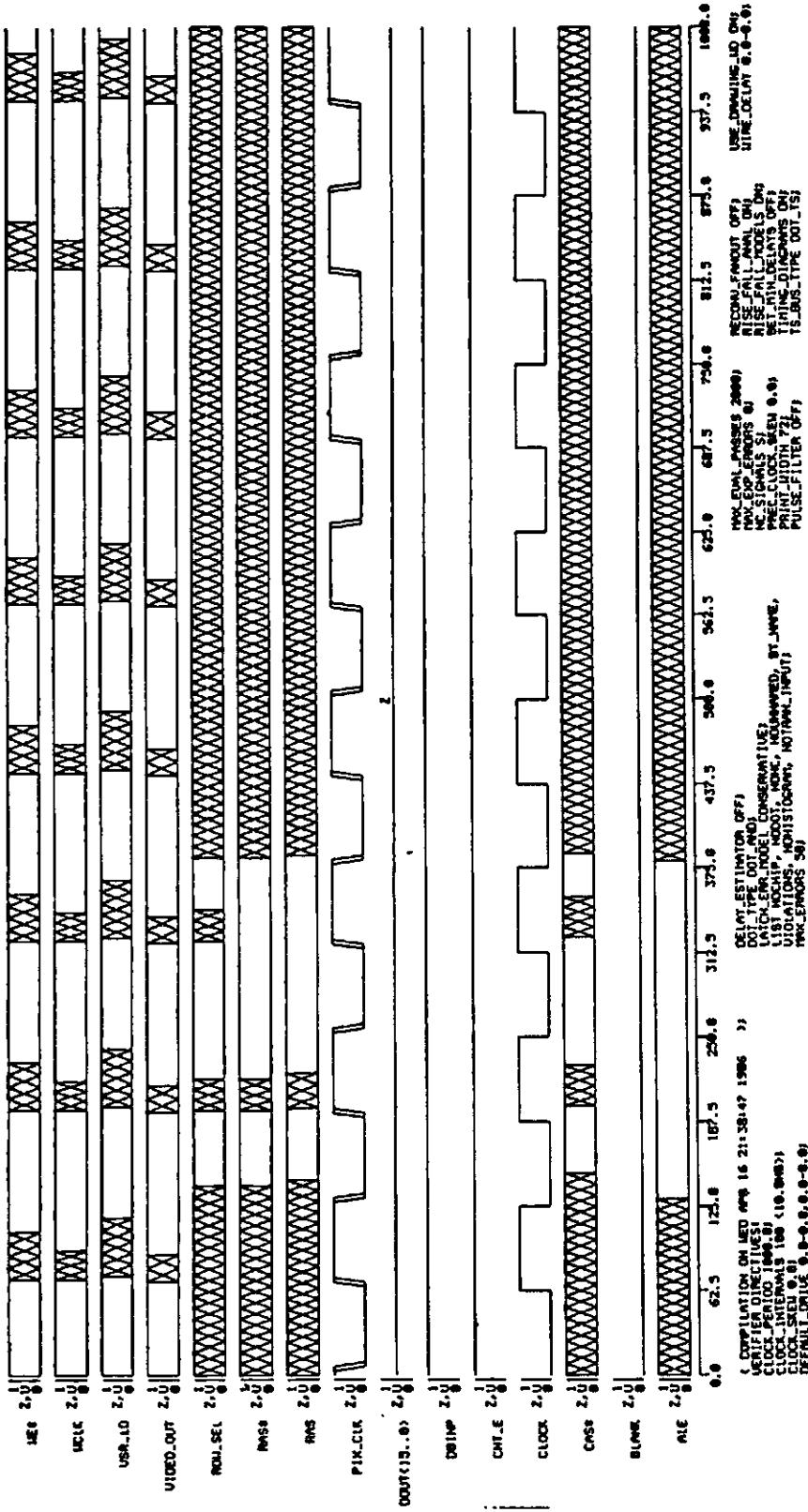
TIMING DIAGRAM OF DISPLAY CYCLE, SIGNAL 'ALE' OCCURING AT EARLIEST TIME.

TIMING DIAGRAM OF DISPLAY CYCLE, SIGNAL 'ALE' OCCURING AT NOMINAL TIME
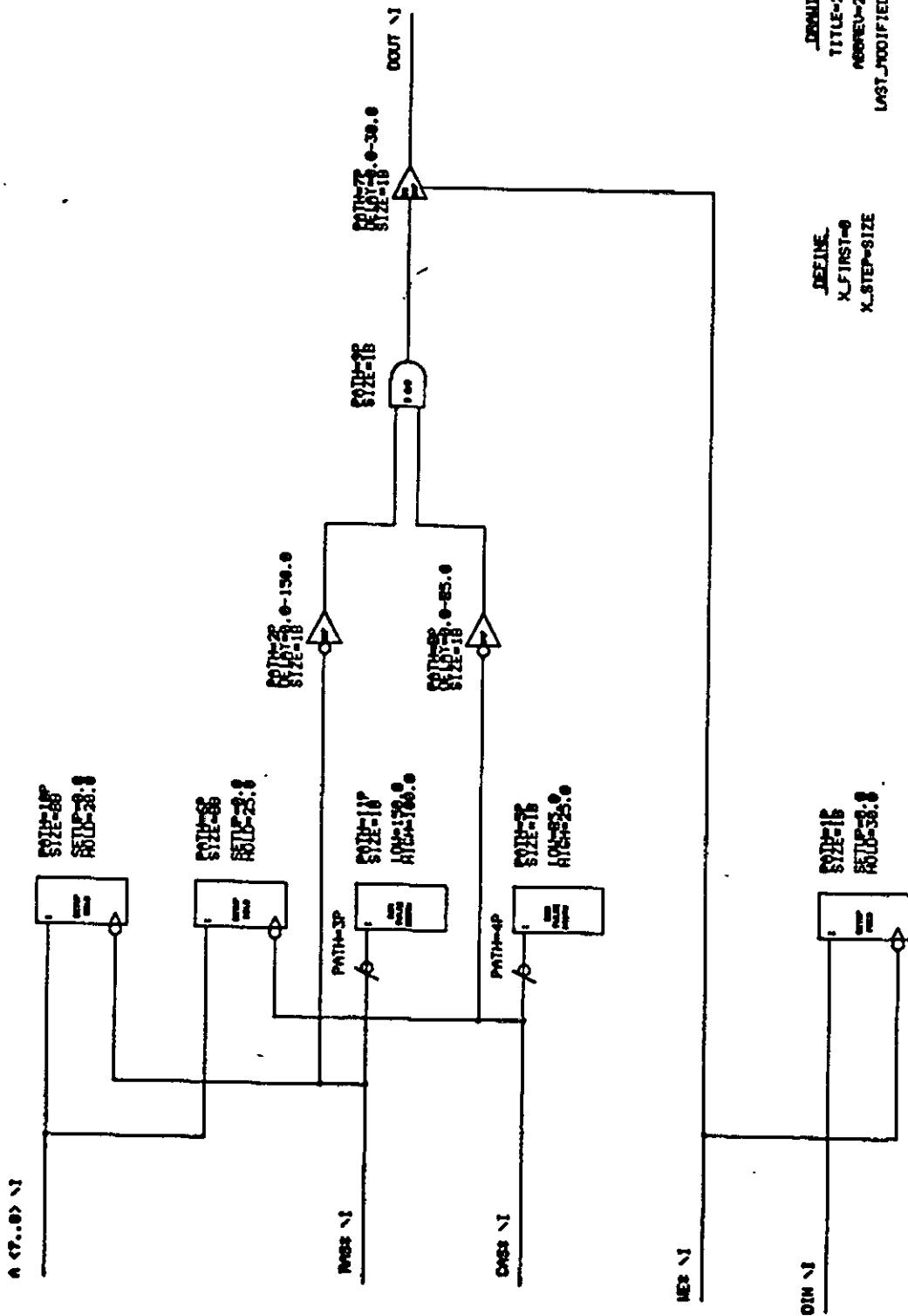
CODE NUMBER 41GM6

TIMING DIAGRAM OF DISPLAY CYCLE. SIGNAL 'ALE' OCCURING AT LATEST TIME

CODE NUMBER 41696

APPENDIX G

TIMING MODEL OF A GENERIC 64K dRAM

APPENDIX H

SCALDSYSTEM AND MERLYN-PCB INTERFACE PROGRAM LISTING

The interface program is composed of many small programs. They are grouped largely into four functions.

1. Generate the master parts file from the Merlyn-PCB parts library,

2. Merge/Split connectors,

3. Convert from the SCALDsystem to the Merlyn-PCB format,

4. Convert from the Merlyn-PCB to the SCALDsystem format.

The listing herein is organized by the functions shown above.

```
# This file is in makemasterpartfile.cmd
# This routine creates the master file containing all the partnames
# used in Merlyn PCB. The created file name is "masterpartfile.dat".

# Check for the proper number of argument.
case $# in
1) ;;
*) echo "Usage: makemasterpartfile (Merlyn-PCB PART LIBRARY FILE)"; exit 2 ;;
esac

if test -f $1
then
    awk ' NF == 2 { print $1} ' $1 !
    sed '/--*/d' )/u0/lib/merlyn/masterpartfile.dat
    echo 'makemasterpartfile: "masterpartfile.dat" is created in /u0/lib/merlyn directory'
else
    echo "makemasterpartfile: file $1 not found" >&2
fi
```

```
# This is in file mgc.cmd.
# This routine merges two connectors into one.

# Check the number of arguments.
if test $# -eq 1
then
    echo "mgc: File $1/merge_edge.dat is empty." >&2
    cat /u0/marval/doc/mergeconnectors.sdoc >&2
    exit 1
fi
# Get the drawing name.
DRAWING=$1
shift

# Repeat for all the edge connectors that need to be merged.
while test $# -gt 0
do

case $# in
    1|2) echo "mgc: Illegally specified format" >&2
         cat /u0/marval/doc/mergeconnectors.sdoc >&2
         exit 1;;
    *) ;;
esac

echo "mgc: Merging connector $1 into $2" >&2
FROMU=`grep $1 $DRAWING/physassign.dat | awk '{print $1}'`
TOU=`grep $2 $DRAWING/physassign.dat | awk '{print $1}'`

sed 's/[.:]/ /g' $DRAWING/$DRAWING.inm |
awk '
NF == 4 {
    if ($1 == "'$FROMU'")
        printf "%s.%s.%1d.%s:\n", '$2',$1d,$3:$3.$4
    else
        printf "%s.%s.%s:\n",$1,$2,$3.$4
}
NF == 3 {
    if ($2 == "'$FROMU'")
        printf "%s.'$TOU'.%1d:\n",$1,'$3'.$3
    else
        printf "%s.%s.%s:\n",$1,$2.$3
}
NF == 1 {print $1}
' >/tmp/$$

mv /tmp/$$ $DRAWING/$DRAWING.inm
shift
shift
shift
done
```

```
# This is in file splitc.cmd.
# This routine splits one connector into two.

# Check the number of arguments.
if test $# -eq 1
then
    echo "splitc: File $1/merge_edge.dat is empty." >&2
    cat /u0/merval/doc/mergeconnectors.sdoc >&2
    exit 1
fi
# Get the drawing name.
DRAWING=$1
shift

# Repeat for all the edge connectors that need to be merged.
while test $# -gt 0
do

    case $# in
    1|2) echo "splitc: Illegally specified format" >&2
         cat /u0/bin/mergeconnectors.sdoc >&2
         exit 1;;

    *) ;;
    esac

    echo "splitc: Splitting connector $2 into $2 and $1" >&2
    FROMU=`grep $1 $DRAWING/physassign.dat | awk '{print $1}'`
    TOU=`grep $2 $DRAWING/physassign.dat | awk '{print $1}'`

    awk '
    NF > 0 {
        if (($1 == "$TOU") && ($3+0 > '$3'+0))
            printf "$FROMU" '$1' %1d %s\n", 0+$3-'$3', $4
        else
            print
    }
    ' pstfnet.dat >/tmp/$$

    mv /tmp/$$ pstfnet.dat
    shift
    shift
    shift
done
```

```
# This is in file rmec.cmd.
# This command removes a connector part from the library file.

# Check for the number of arguments.
case $# in
1) ;;
*) echo "Usage: rmec (part_name)" ; exit 2 ;;
esac

DIR=$HOME/connectors      #### directory in which the library is in ####
LIB_FILE=conn.lib         #### scald to UNIX part-name file          ####
PRT_FILE=conn.prt         #### library file                          ####
PARTNAME=`echo $1 | tr a-z A-Z`

cd $DIR

# Delete the (part_name) from the scald directory file.
if   fgrep "$1" $LIB_FILE >/dev/null
then cp $LIB_FILE /tmp/$$
     sed "/$1/d" /tmp/$$ 1>$LIB_FILE
     echo "rmec: part $1 is deleted from $LIB_FILE"
else echo "rmec: part $1 is not found in $LIB_FILE"
fi

# Delete the informations for (part_name) from the library file.
if   fgrep "$PARTNAME" $PRT_FILE >/dev/null
then mv $PRT_FILE /tmp/$$
     sed "/$PARTNAME/,/END_PRIMITIVE/d" /tmp/$$ 1>$PRT_FILE
     echo "rmec: part $1 is deleted from $PRT_FILE"
else echo "rmec: part $1 is not found in $PRT_FILE"
fi

# Delete all the files in (part_name) directory.
if   ls | fgrep "$1" >/dev/null
then cd $1
     ls
     echo " "
     echo -n "*** the above files are in $1 directory. DELETE ALL? (yes/no) : "
     read answer
     if expr "$answer" : "ye" >/dev/null
     then rm *
     else for i in `ls`
          do
              echo -n "delete $i ? (y/n/q): "
              read answer
              case $answer in
              y*)    rm $i ;;
              q*)    break
              esac
          done
     fi
     cd ..
fi
else echo "rmec: scald directory $1 is not found"
fi
```

```
# Remove the (part_name) directory itself.
if  rmdir $1
then  echo "rmec: $1 scald directory is removed"
else  echo "rmec: $1 scald directory is not removed"
fi

# Clean up.
rm /tmp/$$
```

```
# This is in file mc.cmd.
# This command, given the part-name and size, creates an edge connector.
# for example: "mc j2 33" will create a 33 pin edge-connector whose part-name
# is J2.

DIR=$HOME/connectors        #directory in which the library is in
FILE_LIB=conn.lib           #scald in UNIX part-name file
FILE_PRT=conn.prt           #library file
MAX_SIZE=60                 #maximum number of pins

# Check for the number of arguments to this command.
case $# in
2)  ;;
*)  echo "Usage: mc (part_name) (# of pins)"; exit 1;;
esac

# Check to see if a valid number is given for the (# of pins).
if      echo $2 | grep '[^0-9]' 1>/dev/null
then    echo "mc: $2 is not a number."
        echo "Usage: mc (part_name) (# of pins)"
        exit 2

# Check to see if the size is too big.
elif    test $2 -gt $MAX_SIZE
then    echo "mc: too many pins. Please limit to $MAX_SIZE"
        exit 6
fi

# Set variable PARTNAME to be capitalized word of (part_name).
PARTNAME=`echo $1 | tr "a-z" "A-Z"`
SIZE=$2                     # set variable SIZE to (# of pins).

if test -d $DIR
then cd $DIR
else echo "mc: Connectors library not found, being created in $HOME directory." >&2
     cd
     mkdir connectors
     cp -r /u0/lib/connectors .
     cd $DIR
     echo "mc: Connectors library created." >&2
fi

# Check to see that PARTNAME does not already exist in the library.
# If not, insert     "PARTNAME" 'partname'
# into the custom.lib file.
if      fgrep "$PARTNAME" $FILE_LIB
then    echo "mc: part_name $PARTNAME already exists in library -- see $FILE_LIB file"
        echo "      Use rmec (part_name) to remove the part if it is an edge_connector"
        exit 3

elif    ls | fgrep "$1" 1>/dev/null
```

```
then
        echo "mc: SCALD directory $1 already exists -- use different name"
        exit 5

else
        echo "mc: creating a $SIZE pin edge_connector with part_name of $PARTNAME"
        echo '/ENDC.1/i\'
        "$PARTNAME" +*$1'+:*' )/tmp/$$;
        mv $FILE_LIB $(FILE_LIB).$1
        sed -f /tmp/$$ $(FILE_LIB).$1 :
        tr "+" "*" )$FILE_LIB
        rm $(FILE_LIB).$1

fi

echo "mc: $PARTNAME is added to $FILE_LIB file"


# Create a scald directory. copy all the files from 'edge1' scald directory.
# And replace every occurance of EDGE1 to PARTNAME and EDGE_CONNECTOR to
# PARTNAME.
if      mkdir $1
then    cd $1
        cp ../edge1/* .
        cp chips_prt /tmp/$$
        sed "s/EDGE1/$PARTNAME/
                s/EDGE_CONNECTOR/$PARTNAME/" /tmp/$$ :
        makepins $SIZE )chips_prt

        for i in parts
        do
                sed "s/EDGE1/$PARTNAME/
                        s/EDGE_CONNECTOR/$PARTNAME/" $1 1)/tmp/$$
                mv /tmp/$$ $i

        done
        echo "mc: $1 SCALD directory is created"

else    exit 4
fi

# Insert the chips_prt file of the newly created edge_connector to the custom.prt
# library file.
cd ..
sed /ENDC.1/d $FILE_PRT )/tmp/$$
sed /FILE_TYPE/,/TIME/d $1/chips_prt ))/tmp/$$
mv /tmp/$$ $FILE_PRT
echo "mc: conn.prt file is updated" )&2
```

```
# This file is in vtm.cmd.
# This routine performs compilation, packaging and partname checking
#   of a design and produces a net_list file to be processed by Merlyn.
# This routine can be called with three options:
#   -c : for leaving out the compilation.
#   -p : for leaving out both the compilation and the packaging.
#   +x : for continuing to partname transformation.
#   +k : for transfer to Vax through kermit.
#   +p : for transfer to Vax through fast PIB link.

# Check for the proper calling protocol.
if test $# -eq 0
then echo "Usage: tomerlyn [-c] [-p] [+x] [+k] [+p] (diagram name)" >&2
     echo "  -c : to skip compilation" >&2
     echo "  -p : to skip both compilation and packaging" >&2
     echo "  +x : to perform partname transformation" >&2
     echo "  +k : to perform partname transformation and transfer to Vax through kermit" >&2
     echo "  +p : to perform partname transformation and transfer to Vax through fast PIB link" >&2
     exit 1
fi

# An default, enable both compilation and packaging options.
# Disable partname transformation and transfer options.
compile_option=yes
package_option=yes
transxf_option=no
kermitt_option=no
fastpib_option=no
missing=false

# Decipher the exact command.
while test $# -gt 0

do case $1 in
   -c) compile_option=no; shift;;
   -p) package_option=no; compile_option=no; shift;;
   +x) transxf_option=yes; shift;;
   +k) kermitt_option=yes; transxf_option=yes; shift;;
   +p) fastpib_option=yes; transxf_option=yes; shift;;
   -*|?) echo "tomerlyn: no option $1" >&2; exit 2;;
   *) if test -d $1
      then echo " " >&2
      else echo "tomerlyn: Drawing $1 not found in this directory" >&2
           exit 1
      fi

      if test $compile_option = yes
      then
```

```
echo "tomerlyn: Starting compilation of $1" )&2
compile $1 logic
if tail -8 cmplst.dat | fgrep 'No errors' )/dev/null
then
    echo " Compilation of $1 completed" )&2
    echo " " )&2
else
    echo "tomerlyn: Aborted due to compiling errors" )&2
    echo " See cmplst.dat file for the description of errors" )&2
    exit 3
fi
else
    echo "tomerlyn: Skipping compilation of $1" )&2
    echo " " )&2
fi

if test $package_option = yes
then
    echo "tomerlyn: Starting to package $1" )&2
    if cat packager.cmd | tr "a-z" "A-Z" | fgrep PARTSUMMARY )/dev/null
    then echo " " )&2
    else sed "/[e][n]N][d][D].:/d" packager.cmd )/tmp/$$
        echo "REPORT SPARES.PARTSUMMARY:" ))/tmp/$$
        echo "END." ))/tmp/$$
        mv /tmp/$$ packager.cmd
    fi
    package $1
    echo " " )&2
    echo "tomerlyn: Removing partsfile.dat and physassign.dat files" )&2
    rm $1/partsfile.dat $1/physassign.dat
    echo " " )&2
    if tail -8 pstlst.dat | fgrep 'No errors' )/dev/null
    then
        echo "tomerlyn: Packaging of $1 completed" )&2
        echo " " )&2
    else
        echo "tomerlyn: Aborted due to packaging errors" )&2
        echo " See pstlst.dat file for the description of errors" )&2
        exit 4
    fi
else
    echo "tomerlyn: Skipping packaging of $1" )&2
    echo " " )&2
fi

if test $transxf_option = no
then
    echo "tomerlyn: COMPLETED WITHOUT PARTNAME TRANSFORMATION" )&2
    exit 0
fi

if test -f pstlst.dat
then
```

```
echo " " >&2
else
	cp $1/pstxref.dat pstxref.dat
	cp $1/pstrprt.dat pstprt.dat
	cp $1/cmplst.dat cmplst.dat
fi

echo "tomerlyn: Starting the preparations for partname transformations" >&2
echo " " >&2

if test -f /u0/lib/merlyn/masterpartfile.dat
then echo " " >&2
else echo "tomerlyn: Aborted due to missing /u0/lib/merlyn/masterpartfile.dat." >&2
	echo " Please see the manual for makemasterpartfile" >&2
	exit 4
fi

echo "tomerlyn: Starting to convert from Valid to MERLYN FCB data format" >&2
if test -f pstxref.dat
then
	to_merlyn pstxref.dat
	echo " Conversion done" >&2
	echo " " >&2
	if test -f $1/partsfile.dat
	then
		echo "tomerlyn: Partsfiles exist, thus not created." >&2
		newlycreated=no
	else
		echo "tomerlyn: Starting to create partsfiles" >&2
		makephysassign $1
		echo " Physical part assignment file created in $1/physassign.dat" >&2
		makepartfile $1
		echo " Concise partlist file created in $1/partsfile.dat" >&2
		newlycreated=yes
	fi
	echo " " >&2
else
	echo "tomerlyn: Aborted due to missing pstxref.dat file." >&2
	exit 5
fi

cat temp.out.2 temp.out.1 )$1.inm

if test -f $1/merge_edge.dat
then
	merge_conn=yes
else
	echo -n 'tomerlyn: Do you wish to merge connectors (y/n) ? :' >&2
	read answer
	echo " " >&2
	case $answer in
	y|ye) echo " Connector FROM will be merged into connector TO" >&2
		echo " Please enter the following informations in" >&2
		echo -n " Name of the FROM connector (or quit) ----) " >&2
		read fromname
		while test $fromname != quit
		do
			echo -n " Name of the TO    connector        ----) " >&2
			read toname
```

```
            echo -n " Size of the TO   connector          ---) " )&2
            read tosize
            merge_conn=yes
            echo "$fromname $toname $tosize" : tr "a-z" "A-Z" ))$1/partsfile.dat >/tmp/$$
            echo " " )&2
            sed "/$fromname/d" $1/partsfile.dat >/tmp/$$
            mv /tmp/$$ $1/partsfile.dat
            echo -n " Name of the FROM connector (or quit)  ---) " )&2
            read fromname
        done
        echo " " )&2 ;;
    e)
        merge_conn=no
        echo " " )&2 ;;
    esac
fi

if test $merge_conn = yes
then
    echo "tomerlyn: Found "merge_edge.dat" file, and starting to merge two connectors" )&2
fi

if test $newlycreated = yes
then
    echo "tomerlyn: Checking for the missing partnames from the MERLYN-PCB masterpartfile" )&2
    if searchmissingpartname 'awk "(print $1)' $1/partsfile.dat' )$1/missingpartsfile.dat
    then
        echo " Following partnames are missing from the MERLYN-PCB masterpartfile" )&2
        echo " " )&2
        cat $1/missingpartsfile.dat )&2
    else
        echo " All the partnames are in MERLYN-PCB masterpartfile" )&2
        rm $1/missingpartsfile.dat
    fi
    echo " " )&2
else "tomerlyn: Partsfiles were not created, thus not verified at this time" )&2
fi

if test -f $1/missingpartsfile.dat
then
    echo "tomerlyn: Performing Valid to MERLYN-PCB name transformations" )&2
    if test -f $1/namexref.dat
    then
        echo " " )&2
        localtrans=yes
    else
        echo " " )&2
        echo " Did not find $1/namexref.dat." )&2
        echo -n ' ??? Do you wish to perform Local Name Transformation? (y/n):  ' )&2
        read answer
        case $answer in
        y:Y*) echo '  ??? Please enter the MERLYN-PCB partnames for the following:' )&2
              echo ' *** For an edge connector, YOU MUST NOT GIVE A NAME, ONLY (CR)' )&2
              for i in `cat $1/missingpartsfile.dat`
              do
                  fgrep $i $1/partsfile.dat !
                  awk '
                  NF == 2 {
                      printf "\t%s (%s)\t\t ---) ", $1, $2
```

```
            }
            NF == 1 {
                    printf "\t%s    \t\t --> ", $1
            }' >&2
            read newname
            echo "$i $newname" : tr "a-z" "A-Z" >>$1/namexref.dat
    done
    localtrans=yes;;
*) echo "tomerlyn: Local Name Transformation not applied" >&2
    localtrans=no;;
esac
echo " " >&2
fi

if test $localtrans = yes
then
echo "tomerlyn: Verifying user-supplied Merlyn partnames in $1/namexref.dat file" >&2
cat $1/namexref.dat : tr "a-z" "A-Z" >/tmp/$$
mv /tmp/$$ $1/namexref.dat
if searchmissingpartname 'awk '{print $2}' $1/namexref.dat' >$1/invalidnamesfile
then
    echo "tomerlyn: Following names are invalid Merlyn partnames" >&2
    echo " " >&2
    cat $1/invalidnamesfile
    echo " " >&2
    echo -n " Did you wish to stop ? :" >&2
    read answer
    case $answer in
    y*:Y*) cp $1/namexref.dat $1/namexref.back
           rm $1/namexref.dat
           exit 12;;
    *) echo "       Continuing with the local transformation" >&2;;
    esac
else
echo "tomerlyn: All user-supplied MERLYN-PCB partnames match" >&2
rm $1/invalidnamesfile
fi

echo "tomerlyn: Performing local transformation" >&2
mapvtm $1/namexref.dat $1.inm
echo " " >&2
fi

if test -f /u0/lib/merlyn/namexref.dat
then
    echo "tomerlyn: Performing global transformation" >&2
    mapvtm /u0/lib/merlyn/namexref.dat $1.inm
    echo " " >&2
fi

echo "tomerlyn: Moving the output files from compiler and package to $1 directory" >&2
mv cmp.dat $1
mv pst*.dat $1
echo " " >&2

rm temp.out.1 temp.out.2 temp.out.1
if test $faststrip_option = yes
then
```

```
echo "tomerlyn: Starting to transfer $1.inm file to Vax through fast FIB link" >&2
tovax $1.inm
echo "tomerlyn: Transfer completed" >&2

elif test $kermitt_option = yes
then
echo "tomerlyn: Running kermit.  Please login, and set kermit to receive." >&2
kermit c
echo "tomerlyn: Starting to transfer $1.inm file to Vax through kermit" >&2
kermit s $1.inm
echo "tomerlyn: Completed transfer." >&2
echo " Connecting back to kermit. You may exit from kermit." >&2
kermit c

fi

mv $1.inm $1/$1.inm
echo " " >&2
echo "tomerlyn: COMPLETED" >&2
cat /u0/merval/doc/tovalid.sdoc >&2
exit 0 ;;

esac

done
```

```
# This file is in mtv.cmd.
# This routine performs the backannotation

# Check for the correct number of arguments.
if test $# -eq 0
then echo "Usage: tovalid [+k] <diagram name>" >&2
     echo " +k to transfer from Vax through kermit link" >&2
     echo " The fast PIB link is selected as default" >&2
     exit 1
fi

# As default, select the fast PIB link.
fastpib_option=yes

# Decipher the command.
while test $# -gt 0
do case $1 in

+k) fastpib_option=no; shift;;

+?|-?) echo "tovalid: no option $1" >&2 ; exit 2;;

*) if test -d $1
   then echo " ." >&2
   else echo "tovalid: Aborted due to missing drawing name $1" >&2
        exit 3
   fi

if test $fastpib_option = yes
then
     echo "tovalid: Starting to transfer $1.out file from Vax through PIB link" >&2
     fromvax $1.out
else
     echo "tovalid: Running kermit. Please login. start kermit and send $1.out to valid." >&2
     kermit c
     echo "tovalid: Receiving file $1.out" >&2
     kermit r
     echo "tovalid: Connecting to kermit. You may exit from kermit and log out if you wish." >&2
     kermit c
fi

if fgrep "Parts Net Document" $1.out >/dev/null
then echo " ." >&2
else echo "tovalid: File $1.out is not produced by MERLYN-FCB (Parts Net link) command" >&2
     exit 4
fi

if test -f $1/physassign.dat
then
     echo "tovalid: Producing pstfnet.dat file for the package." >&2
     fmerlyn $1
else
     echo "tovalid: Aborted due to missing $1/physassign.dat file" >&2
     exit 5
fi

echo "tovalid: Retrieving compiler and packager state files" >&2
cp $1/cmps.dat .
cp $1/pste.dat .
```

159

```
echo "tovalid: Starting the Packager to produce backannotation file for GED" >&2
sed '/[Ee]CN:nJ[Dd]:./d' package.cmd )/tmp/$$
mv /tmp/$$ packager.cmd
echo "FEEDBACK_ORDER FEEDBACK_NETLIST:" >>packager.cmd
echo "END." >>packager.cmd
package $1
sed '/FEEDBACK_ORDER FEEDBACK_NETLIST:/d' packager.cmd )/tmp/$$
mv /tmp/$$ packager.cmd

if tail -8 pstlst.dat : fgrep 'No errors' >/dev/null
then
    echo "tovalid: Backannotation file for GED produced successfully." >&2
    echo " To backannotate the changes, start ged and give "backannotate" command" >&2
    mv pstbark.dat backann.cmd
else
    echo "tovalid: Aborted due to packager errors. See pstlst.dat file" >&2
    exit 3
fi

echo "tovalid: Moving the new packager state files to $1 directory" >&2
mv pst*.dat $1

echo "tovalid: COMPLETED" >&2
shift;;
esac

done
```

```
# This file is in #merlyn.cmd.
# This routine takes the "Parts Net Document" file from MERLYN-PC
# produces the netlist for feedback processing by Valid packager.
# The partname should be a single word without any embedded blank

# Check for the correct number of arguments.
case $# in
1)!!
2) echo "Usage: $0 (Drawing Name)" >&2; exit ! ;;
esac

awk '
NF > 7 {
    if ($4 == "X")
        print $1, $2
    if ($6 == "X")
        print $1, $2, $3
    if ($7 != "X")
        print $1, $2, $3
}
' $1.out !

awk '
NF == 3 {
    u_number = $1
}
NF > 0 {
    if ( NF == 3)
        print u_number, u_number, $2, $3
    else
        print u_number, u_number, $1, $2
}
' >pstfnet.dat

# Transforming MERLYN-PCB partnames into Valid partnames.
mapuntv $1/physassign.dat pstfnet.dat

# Split merged connectors, if exist.
if test -f $1/merge_edge.dat
then
    echo "#merlyn: Found $1/merge_edge.dat file.  Merging connecto
    split< $1 `cat $1/merge_edge.dat ; tr "a-z" "A-Z"`'
fi

# Create pstfnet file.
echo "FILE_TYPE = FEEDBACK_NETLIST;" >/tmp/$$
echo "ROOT_DRAWING = '$1';" >>/tmp/$$
awk '(printf \"'%s' '%s' %s '%s';\n\", "$1, $2, $3, $4)' pstfnet.
echo "END." >>/tmp/$$
mv /tmp/$$ pstfnet.dat
```

```c
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{ int c,maxpin;

    maxpin = atoi(argv[1]);
    while ((c=echo()) != EOF)
        if (c == 'B')
            if ((c=echo()) == 'E')
                if ((c=echo()) == 'R')
                    if ((c=echo()) == '=')
                    { echo();        /* echo ' */
                      echo();        /* echo ( */
                      echo();        /* echo 1 */
                      for (c=2; c<=maxpin; c++)
                          printf(",%1d",c);
                      getchar();     /* delete , */
                      getchar();     /* delete A */
                    }

}

int echo()
{ int c;

    return(putchar(c=getchar()));
}
```

```
# This is in file makepartfile.cmd
# This routine echoes the the partnames used in the diagrams to standard output.
# The packager must be run with "report partsummary" directive.
# This routine uses "pstrprt.dat" file output by the packager.

# check for the existence of partnames, indicated by the presence of "PART SUMMARY".
if (grep "PART SUMMARY" pstrprt.dat >/dev/null
then
    sed '1,/PART SUMMARY/d
         /Total/d' pstrprt.dat |
    awk '
    NF == 2 {
        if ($2 > 0)
            print $1
            version = 0
    }
    NF == 3 {
        printf "%s-%ld %s\n", $1, ++version, $2
    }' >$1/partsfile.dat
else
    echo "makepartfile: ERROR in pstrprt.dat file: partnames not found." >&2
    echo " Packager.cmd file must contain "report partsummary" directive" >&2
    echo " " >&2
    exit 2
fi
```

```
# This file is in makephysassign.cmd.
# This routine creates a file containing the physical part
# assignments to the partnames.

sed '1,/GLOBAL PART/d' pstxref.dat |
awk 'NF == 2 { print $1, $2}' |
grep 'U[@-9][@-9].' >$1/physassign.dat

# awk '{print $2}' $1/physassign.dat |
# sort -u >$1/partsfile.dat
```

164

```
# This routine searches the masterpartfile for the given partname.
# An appropriate message is issued if found or not found.
# All the partnames that were not found in masterpartfile are
# put into "missingpartnames" file.

for i
do
    if fgrep $i masterpartfile >/dev/null
    then echo "$i found"
    else echo "$i not found"
        echo "$i" >>missingpartnames
    fi
    shift
done
echo "missing partnames, if exist, are listed in "missingpartnames" file"
```

```
# This file is in searchmissingpartname.cmd.
# This routine searches the masterpartfile.dat for the given partname.
# All the partnames that are not found in masterpartfile are
# written to standard output.

# Initially, indicate that there is no missing partname.
missing=false

# Ln for all the arguments to this command.
for i
do
    if fgrep $i /u0/lib/merlyn/masterpartfile.dat  >/dev/null
       then shift        #If found, check with the next partname.
       'else echo "$i"    #If not, echo the partname.
             missing=true #Indicate the search failure.

    shift
    fi
done

# Indicate whether there was a partname missing or not.
if test $missing = true
   then exit 0
   else exit 1

fi
```

```
# This file is in searchmerlynparts.cmd.

# Initially, indicate that there is no missing partname.
missing=false

# Do for all the arguments to this command.
for i
do
    if fgrep $i /u0/bin/merlyn/merlynpartsfile.dat
    then shift
    else missing=true
         shift
    fi
done

# Indicate whether there was a partname missing or not.
if test $missing = true
then exit 0
else exit 1
fi
```

```
# This file is in map.cmd
# This command performs word(s) substitution.
# The mapping table file should contain two words per line, the first
# word being the word to be replaced and the second word being the
# replacement. The word separator is blank(s).


# check for the proper number of arguments to this command.
case $# in
2);;
*)echo "Usage: $0 (mapping table) (file to be transformed)"; exit 2;;
esac


# create a shell script to perform the substitution using sed.
echo "sed 's/+/+/" >/tmp/$0$$            #A dummy substitution.
awk ' NF == 2 { printf "s/.%s./.%s./\n", $1, $2 }' $1 >>/tmp/$0$$
echo "s/=/=/' $2$$" >>/tmp/$0$$


# run the script and clean up.
sh /tmp/$0$$
rm /tmp/$0$$
mv $2$$ $2          #write the changes into the original file.
```

```
# This file is in mapvtn.rmd
# This command performs word(s) substitution.
# The mapping table file should contain two words per line, the first
# word being the word to be replaced and the second word being the
# replacement. The word separator is blank(s).
# If the second word is missing, the first word is deleted from the file.

# check for the proper number of arguments to this command.
case $# in
2);;
*)echo "Usage: mapvtn (mapping table) (file to be transformed)"; exit 2;;
esac:

# create a shell script to perform the substitution using sed.
echo "sed 's/+/+/" )/tmp/$$        #A dummy substitution.
awk ' NF )= 1 ( printf "s/%s,/%s,/\n", $1, $2 }' $1 )>/tmp/$$
echo "s/=/=/' $2 )$;$$" >>/tmp/$$

# run the script and clean up
sh /tmp/$$
mv $2$$ $2
rm /tmp/$$
```

```
# This file is in mapmtv.cmd
# This command performs word(s) substitution.
# The mapping table file should contain two words per line, the second
# word being the word to be replaced and the first word being the
# replacement. The word separator is blank(s).

# check for the proper number of arguments to this command.
case $# in
2);;
*)echo "Usage: mapmtv (mapping table) (file to be transformed)"; exit 2;;
esac

# create a (shell) script to perform the substitution using sed.
echo "sed 's/+/+/' >/tmp/$$               #A dummy substitution.
awk ' NF == 2 { printf "s/ %s / %s /\n", $1, $2 }' $1 >>/tmp/$$
echo "s/=/=/' $2 )$2$$ ">)/tmp/$$

# run the script and clean up
sh /tmp/$$
mv $2$$ $2
rm /tmp/$$
```

APPENDIX I
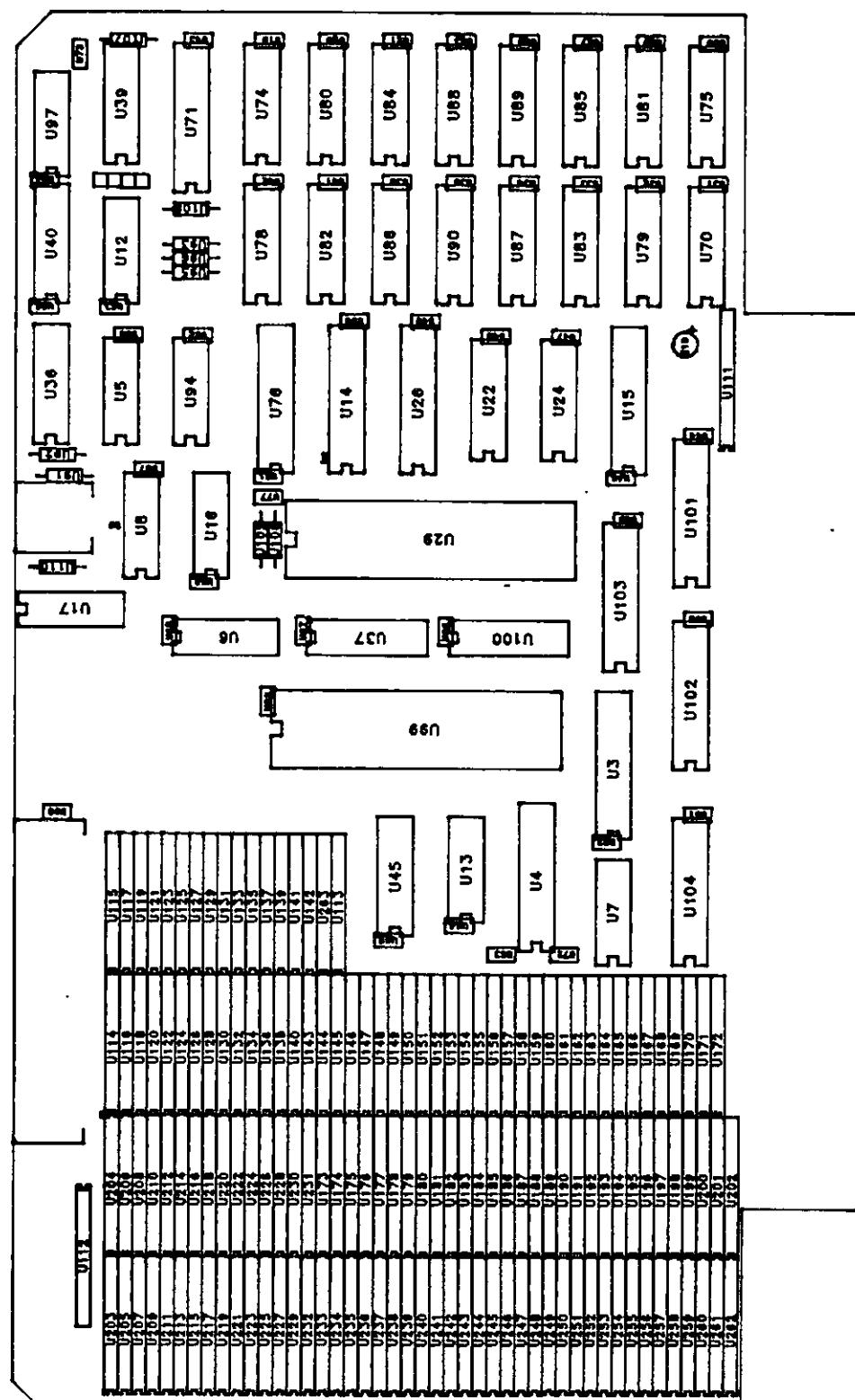
PHYSICAL DESCRIPTION OF THE GDC BOARD
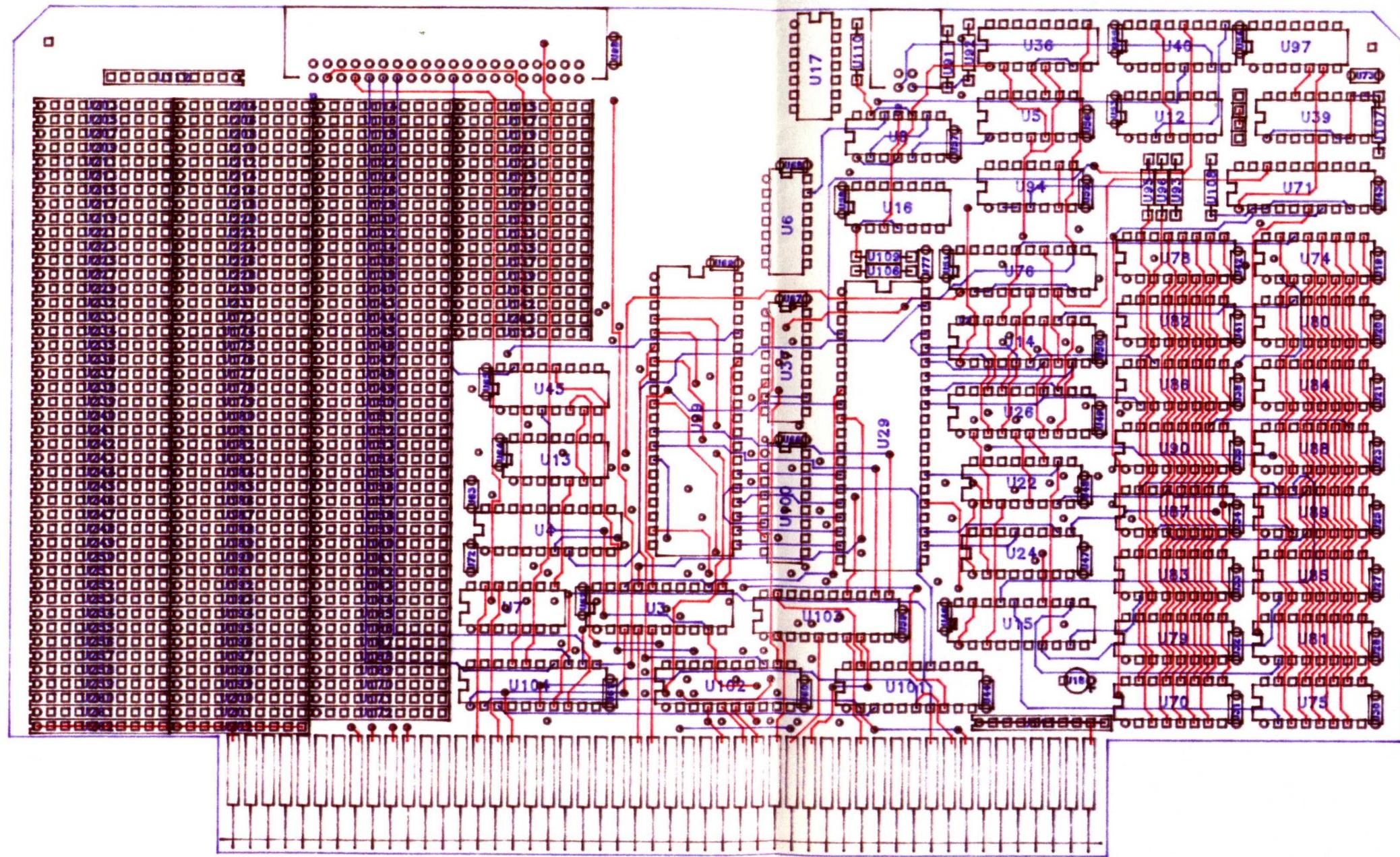
The layout of the original board is shown in page 171.
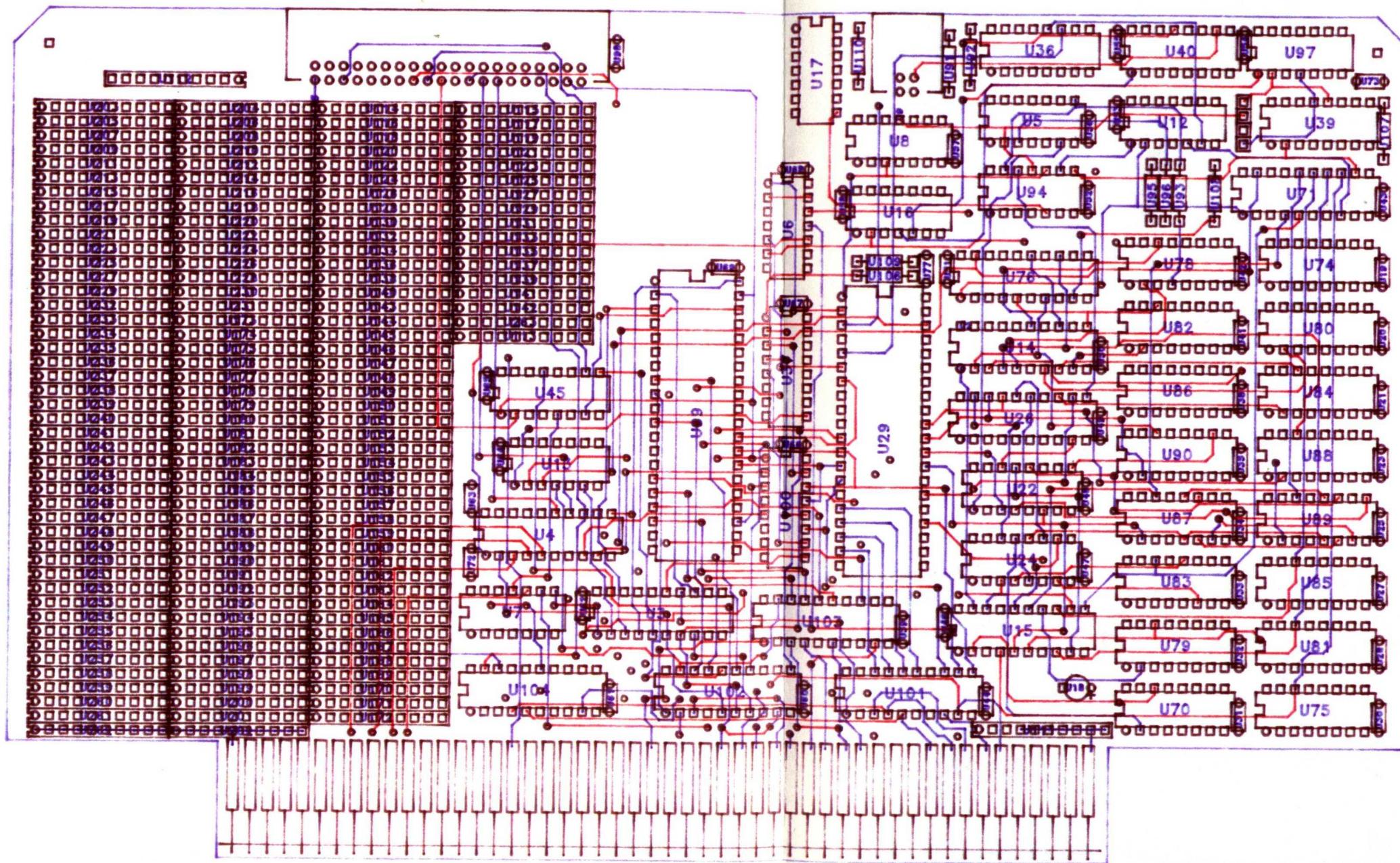
The routing of top and inner 1 layers is shown in page 172.
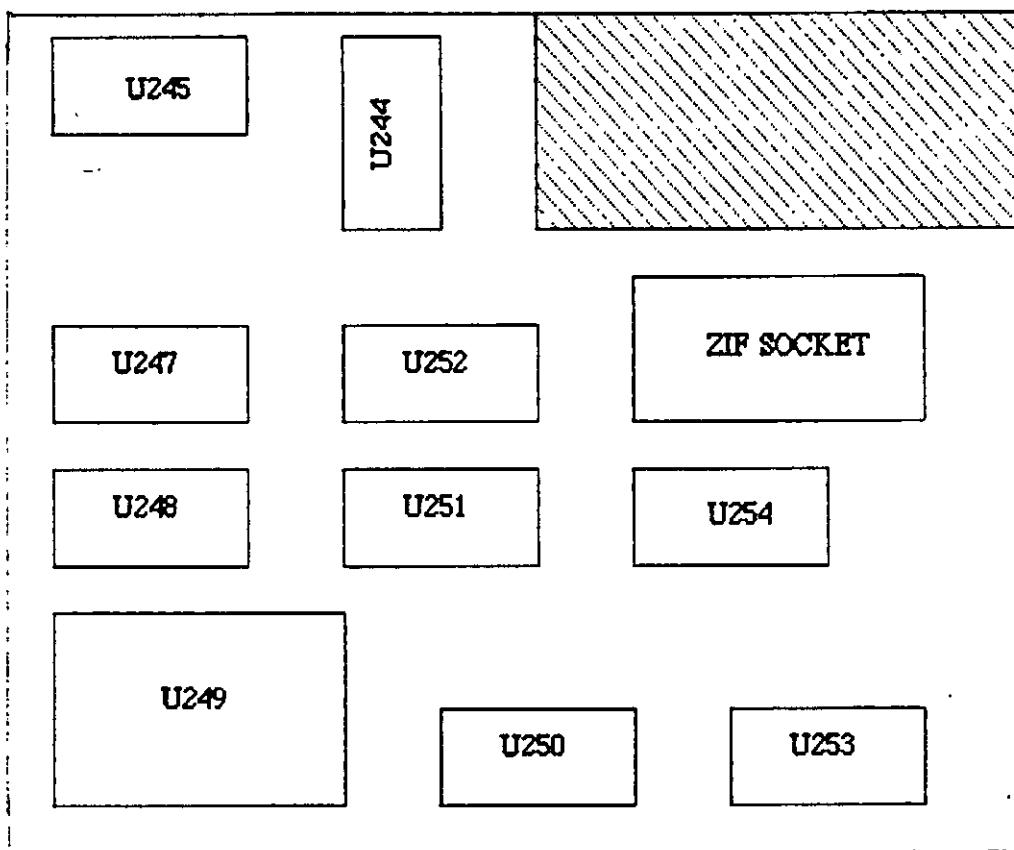
The routing of inner 2 and bottom layers is in page 173.

The layout of the added chips is shown in page 174.

U245

U244

U247

U252

ZIF SOCKET

U248

U251

U254

U249

U250

U253

APPENDIX J

DEVICE DESCRIPTOR LISTING

```
 NAM GDC
 TTL Device Descriptor for NEC7220 GDC
 MOD GDCEND,GDCNAM,$F1,$81,FMNS,DDNS
 FCB 3             update mode
 FCB $FF           extended port address
 FDB $FF60         base address of GKS graphics board
 FCB GDCNAM-*-1    optional parameter byte count
 FCB 0             scf device type

* Default parameters
RESET EQU 0       reset opcode
PITCH EQU $47     pitch opcode
PRAM  EQU $70     ram parameter load opcode
CCHAR EQU $4B     cursor characteristic opcode
NP    EQU $7F     special opcode to indicate that the next byte
*                 is an opcode with 1 to 16 parameters. Example:
*                 NP+4 means that this opcode has 4 parameters
EOFOP EQU $FF     special opcode to indicate end of file

WWDTH FCB 50      window width is 50 words
BWDTH FCB 50      frame buffer width is 100 words
WLINE FDB 521     window length is 521 lines
BLINE FDB 655     frame length is 65535 div pitch

* command sequences to configure 7220
 FCB NP+8          an opcode with 8 parameters
 FCB RESET
 FCB %00011111     graphics mode, interlaced, dRAM refresh
 FCB 50-2 50       active words per line
 FCB %01100100     HS = 5
 FCB %00001000     HFP = 3, VS = 3
 FCB 5-1           HBP = 5
 FCB 1             VFP = 1
 FCB $04           260 ($104) active lines/field
 FCB %00010001     VBP = 4
 FCB NP+1
 FCB PITCH
 FCB 50            50 words in horizontal direction
 FCB NP+4
 FCB PRAM+0        parameter load from 0
 FCB 0             starting address is 0
 FCB 0
 FCB $90           line length is 521 ($209) lines
 FCB $20
 FCB NP+3
 FCB CCHAR
 FCB 0
 FCB %11000101
 FCB %00011000
 FCB EOFOP
GDCNAM FCS "GDC"      device name
FMNS   FCS "SCF"      file manager
DDNS   FCS "GDC.DD"   device driver name
 EMOD MODULE CRC
GDCEND EQU *
```

APPENDIX K

DEVICE DRIVER LISTING

```
/* This is sks.h file */
/* This file contains data structures used in the device driver */
/* All routines written in C must use structure definitions    */
/*      declared herein */



/* 6809 register packet declaration */
struct registers {
    char rs_cc,rs_a,rs_b,rs_dp;
    unsigned rs_x,rs_y,rs_u;
    } ;



/* Type declaration */
typedef struct {        /* device descriptor type for sdc */
  char sys[0x12];       /* module header */
  char dev_type,        /* device type */
       wwdth,           /* window width in words */
       bwdth;           /* frame buffer width in words */
  unsigned wline,       /* window length in lines */
           bline;       /* frame buffer length in lines */
  char commands[25];    /* upto 25 bytes of commands */
} dev_sdc;


typedef struct {        /* device static storage for sdc */
  char port_ext;        /* extended port address */
  char *port;           /* port address */
  char junk[32];        /* area used by os9 system */
  char bcount;          /* counts number of parameters a command */
  unsigned wxmin,       /* left edge of window in pixels */
           wxmax,       /* right edge ofindow in pixels */
           wymin,       /* top edge of window in pixels */
           wymax,       /* bottom edge of window in pixels */
           fxmax,       /* right edge of frame buffer in pixels */
           fymax,       /* bottom edge of frame buffer */
           param[5];    /* figure drawing parameters */
  char zoomf;           /* display and schar zoom factor */
  char table[16];       /* direction table for line drawing */
  char mode;            /* drawing mode */
  unsigned pattern;     /* drawing pattern */
  char atable[8];       /* direction table for arc drawing */
} dss_sdc;


typedef struct {                /* path descriptor */
  char xx[6];                   /* not interested */
  struct registers *pd_rgs;     /* pointer to register packet */
  } pd_sdc;
```

```
/* functions to access the 6809 MPU registers */
char set_a(),          /* returns the value of accumulator A */
     *set_y(),         /* returns the value of Y register */
     *set_u();         /* returns the value of U register */
unsigned swap();       /* swaps high and low bytes */


/* new names for the functions */
#define set_dss()    ((dss_sdc *)set_u())
#define set_dd()     ((dev_sdc *)set_y())
#define set_pd()     set_y()



/* offsets to device addresses from sks_board base address */
#define sdcbase 0x18       /* base address of 7220 GDC chip */
#define zoomadrs 0x1a      /* address of zoom pre-scaler */
#define dma3port 0x0c      /* port 3 of dma controller */
#define channel_cont 0x13  /* channel 3 control reg */

#include <sks.h>
term()
{ noerr(); }

read()
{ noerr(); }

setstt()
{ noerr(); }
```

```
/* This is gksop.h file */
/* This file contains the GDC command byte definitions */
/* All routines should use the constants defined here  */
/*      to prosram the GDC */


/* 7220 GDC opcodes */

#define eopcode 0xffff /* end of init command sequence opcode */
#define vsync   0x6f    /* vertical sync */
#define pram    0x70    /* parameter load */
#define zoom    0x46    /* set zoom */
#define start   0x6b    /* exit from idle mode */
#define bctrl   0x0c    /* blanking control */
#define wdat    0x20    /* write data */
#define curs    0x49    /* cursor set */
#define figs    0x4c    /* fisure setup */
#define mask    0x4a    /* mask setup */
#define reset   0      /* reset */
#define sync    0x0e    /* sync */
#define cchar   0x4b    /* cursor characteristic */
#define pitch   0x47    /* pitch */
#define figd    0x6c    /* fisure draw */
#define achrd   0x68    /* sraphics character */
#define rdat    0xa0    /* read data */
#define curd    0xe0    /* cursor location read */
#define dmar    0xa4    /* dma read */
#define dmaw    0x24    /* dma write */


/* RMW cycle losic operation */
#define mod_rpl    0    /* replace mode */
#define mod_com    1    /* complement mode */
#define mod_reset 2     /* reset mode */
#define mod_set    3    /* set mode */


/* data transfer type */
#define tfr_wd     0            /* word transfer, low first */
#define tfr_low    0x10         /* byte transfer, low byte */
#define tfr_hi     0x18         /* byte transfer, hish byte */


/* mixed commands */
#define bksndop wdat+tfr_wd+2  /* backsround set */


/* Error codes */
#define E_UNKSVC  0xd0          /* unknown service request */
```

```
/* This is skssvc.h file */
/* This file contains the SETSTAT function code definitions */
/*       for the hish level sraphics prosrammins packase */


#define _off          0
#define _on           1


/* SETSTAT function code definitions */
#define _blank       -1    /* blank screen */
#define _display     -2    /* unblank screen */
#define _bksndon     -3    /* set backsround to on */
#define _bksndoff    -4    /* set backsround to off */
#define _dzoom       -5    /* set display zoom factor */
#define _czoom       -6    /* set sraphics character zoom factor */
#define _dot         -7    /* turn dot on at cursor */
#define _move        -8    /* move cursor to new coordinate */
#define _rect        -9    /* rectansle draw */
#define _dia        -10    /* diamond draw */
#define _line       -11    /* line draw */
#define _mode       -12    /* drawins mode set */
#define _pattern    -13    /* drawins pattern set */
#define _arc        -14    /* arc drawins */
#define _sfill      -15    /* set fill pattern */
#define _dfill      -16    /* draw area fill */
#define _ccc        -17    /* set character heisht */
#define _part1      -18    /* set display area 1 */
#define _part2      -19    /* set display area 2 */
#define _dmaw       -20    /* dma write to sdc */
#define _dmar       -21    /* dma read from sdc */


/* structure for passins parameters to the SETSTAT functions */
typedef struct {    /* parameter packet */
    unsisned _p1,_p2,_p3,_p4,_p5;
} ppacket;
```

```
/* This is header.a file */
/* This file contains the module header declaration needed */
/*      for linking with the device driver written in C */


 NAM HEADER.A header file for device drivers
 PSECT GDC.DD,$e1,$81,1,200,entry
entry lbra init
      lbra read
      lbra write
      lbra setstt
      lbra setstt
      lbra term



/* Embedded assembly language routines */

set_y:
 tfr y,d     returns the content of Y register to caller
 rts         use set_y() to call this function


set_u:
 tfr u,d     returns the content of U register to caller
 rts         use set_u() to call this function


set_a:
 tfr a,b     returns the content of A accumulator to caller
 rts         use set_a() to call this function


noerr:
 clrb        clear carry
 rts         exit from routine with no error condition

error:
 ldb 3,s     set error number
 coma        set carry
 rts         exit from routine with error(code) call


swap:
 exg a,b  swap low and high byte
 rts
```

```
/* This function returns the value of SINE function for */
/*      an angle.  The function is an implementation of the */
/*      SINE(angle) = ( angle * 16 + correction ) / 1000 */
/*      where the correction is defined in the look-ip table */
/* This function works only between 0 to 45 degrees */

sin1k:
 ldd 2,s        set angle
 leax stbl,pcr  point to sine table
 ldb b,x        set correction value for the angle
 sex            sign extension for 16 bit integer
 ldx 2,s        set angle
 exg d,x        exchange so angle will be pushed later
 pshs d,x       save x, pass angle in d to mul16
 lbsr mul16     calculate 16*angle = rough value
 addd 2,s       add the correction factor
 leas 4,s       balance stack
 rts            return the value thru D accumulator


stbl fcb 0,1,3,4,6,7     for angles 0  to 5  degrees
     fcb 9,10,11,12,14             6  to 10 degrees
     fcb 15,16,17,18,19            11 to 15 degrees
     fcb 20,20,21,22,22            16 to 20 degrees
     fcb 22,23,23,23,23            21 to 25 degrees
     fcb 22,22,21,21,20            26 to 30 degrees
     fcb 19,18,17,15,14            30 to 35 degrees
     fcb 12,10,8,5,3              36 to 40 degrees
     fcb 0,-3,-6,-9,-13           41 to 45 degrees


endsect
```

```
/* This is init.c file */
/* This file contains the INIT routine of the device driver */


#include <sks.h>
#include <rksop.h>


unsigned mul16();

init()
{ char *cmd,                /* pointer to commands */
       *table,              /* pointer to direction tables */
       c;
  dev_sdc *dd;              /* device descriptor pointer */
  dss_sdc *dss;            /* device static storage pointer */


  dd = set_dd();            /* set device descriptor */
  dss = set_dss();          /* set device static storage */

  table = dss->table;       /* initialize line table */
  table[0]  = 0x11;         /* octant 0, swap axis */
  table[1]  = 0x12;         /* 1 */
  table[2]  = 0x12;         /* 1, -45 degree line down */
  table[3]  = 0x14;         /* 2, horizontal right */
  table[4]  = 0x17;         /* 3, swap axis */
  table[5]  = 0x14;         /* 2 */
  table[6]  = 0x16;         /* 3, 45 degree line up */
  table[7]  = 0x18;         /* 4, vertical line up */
  table[8]  = 0x1f;         /* 7, swap axis */
  table[9]  = 0x1c;         /* 6 */
  table[10] = 0x1e;         /* 7, 225 degree line down */
  table[11] = 0x1e;         /* 4, not possible combo */
  table[12] = 0x19;         /* 4 */
  table[13] = 0x1a;         /* 5 */
  table[14] = 0x1a;         /* 5, 135 degree line up */
  table[15] = 0x1a;         /* 5, not possible combo */


  table = dss->atable;      /* initialize atable */
  table[0] = 4;             /* octant 2 */
  table[1] = 1;             /* octant 3 */
  table[2] = 6;             /* octant 4 */
  table[3] = 3;             /* octant 5 */
  table[4] = 0;             /* octant 6 */
  table[5] = 5;             /* octant 7 */
  table[6] = 2;             /* octant 0 */
  table[7] = 7;             /* octant 1 */
```

```
dss->bcount = dss->zoomf = 0;      /* init for WRITE routine */
dss->wxmin = dss->wymin = 0;       /* initialize window */
dss->wxmax = dd->wwdth;
dss->wymax = dd->wline;
dss->fxmax = dd->bwdth;            /* initialize frame buffer */
dss->fymax = dd->bline;


cmd = dd->commands;                     /* point to list of commands */
while ( *cmd != eopcode )                /* repeat until EOPCODE */
   if ( *cmd & 0x80 )                    /* if bit 7 is set */
     { c = (*cmd++ + 1) & 0x7f;          /* add 1, clear bit 7 */
       writecmd(*cmd++);                 /* write opcode */
       for ( ; c ; c--)
         writepar(*cmd++);               /* write c parameters */
     }
   else writecmd(*cmd++);                /* write command by itself */


writecmd(vsync);
set_pattern(dss,0xffff);
set_mode(dss,mod_set);
set_zoom(dss,0);
writecmd(start);


noerr();                                 /* no error */
}
```

```
/* This is write.c file */
/* This file contains the WRITE routine of device driver */


#include <sks.h>


write()
{ char c,
     *cnt;          /* pointer to the number of parameter counts */


  c = set_a();                     /* set content of A accumulator */
  cnt = &((set_dss())->bcount);              /* point to bcount */

  if ( *cnt )                  /* if bcount is not zero */
    if ( *cnt & 0x80 )         /* if command bit is set */
      { writecmd(c);           /* output opcode */
        *cnt &= 0x7f;          /* clear cmdfls flag */
      }
    else                       /* command bit is not set */
      { writepar(c);           /* output parameter */
        *cnt -= 1;             /* decrement bcount */
      }
  else                         /* bcount is zero */
    if ( c & 0x80 )            /* if bit 7 is set */
      *cnt = ++c;              /* initialize bcount */
    else                       /* bit 7 is cleared */
      writecmd(c);             /* write a command by itself */

  noerr();
}
```

```c
/* This is setstt.c file */
/* This file contains the SETSTT routine of device driver */
/* The utility functions in sksutil.c and util.c files are used */


#include <sks.h>
#include <sksop.h>
#include <skssvc.h>


#define   shiftL4(x) mul16(x)

setstt()
{ struct registers *rpack;        /* pointer to register pack */
  char code;                      /* service request code */
  ppacket *xpar;                  /* pointer to parameter packet */
  dss_sdc *dss;                   /* static storage pointer */


  rpack = (set_rd())->pd_rss;     /* point to register packet */
  code = rpack->rs_b;             /* set service code */
  xpar = rpack->rs_x;             /* point to parameterr packet */
  dss = set_dss();                /* point to static storage */


  switch (code) {                 /* jump to requested function */
    case _blank    : blanking(_off);
                     break;       /* blank screen */
    case _display  : blanking(_on);
                     break;       /* unblank screen */
    case _bksndon  : set_bksnd(dss,_on);
                     break;       /* turn backsround on */
    case _bksndoff : set_bksnd(dss,_off);
                     break;       /* turn backsround off */
    case _dzoom    : set_zoom(dss,dss->zoomf & 0x0f | shiftL4(xpar->_p1));
                     break;       /* chanse display zoom factor */
    case _czoom    : set_zoom(dss,dss->zoomf & 0xf0 | xpar->_p1);
                     break;       /* chanse sraphics character zoom */
    case _dot      : dot(dss,xpar);
                     break;       /* turn on dot */
    case _move     : set_cursor(dss,xpar->_p1,xpar->_p2);
                     break;       /* move cursor to (_p1,_p2) */
    case _rect     : rectansle(dss,xpar,0x40);
                     break;       /* draw rectansle */
    case _dia      : rectansle(dss,xpar,0x47);
                     break;       /* draw slanted rectansle */
    case _line     : line(dss,xpar);
                     break;       /* draw a line */
    case _mode     : set_mode(dss,(char)xpar->_p1);
                     break;       /* set drawins mode */
    case _pattern  : set_pattern(dss,xpar->_p1);
                     break;       /* set drawins pattern */
    case _arc      : arc(dss,xpar);
                     break;       /* draw an arc */
```

```
    case _sfill     : set_fill(dss,xpar);
                      break;       /* set area_fill pattern */
    case _dfill     : draw_fill(dss,xpar);
                      break;       /* draw area_fill */
    case _ccc       : set_ccc(dss,xpar);
                      break;       /* set character height */
    case _part1     : partition(dss,xpar,0);
                      break;       /* set display area partition 1 */
    case _part2     : partition(dss,xpar,4);
                      break;       /* set display area partition 2 */
    case _dmaw      : dmaw_write(dss,xpar);
                      break;       /* DMA write to the GDC */
    case _dmar      : dma_read(dss,xpar);
                      break;       /* DMA read from the GDC */
    default         : error(E_UNKSVC) ; break;
  }             /* end of case */
}             /* end of setstat */




blanking(mode)
unsigned mode;
{ writecmd(bctrl + mode);
  noerr();
}




set_bkgnd(dss,mode)
dss_sdc *dss;
char mode;
{ char tmode,i;
  unsigned tpattern;

  tmode = dss->mode;
  tpattern = dss->pattern;

  set_cursor(dss,0,0);                 /* define background area */
  set_mask(0xffff);                    /* effects all bits */
  dss->param[0] = 0x3fff;              /* of 16k words */

  for (i = 4 ; i ; i--)
  { set_figure(dss,2,1);               /* 1 parameter. */
    write_data(wdat+tfr_wd+2+mode,0xffff);
  }

  set_mode(dss,tmode);                 /* restore drawing mode */
  set_pattern(dss,tpattern);           /* restore drawing pattern */

  noerr();
}             /* end of set_bkgnd */
```

```
dot(dss,xpar)
dss_sdc *dss;
ppacket *xpar;
{
   set_cursor(dss,xpar->_p1,xpar->_p2);
   set_figure(dss,2,0);
   writecmd(fisd);
}



rectangle(dss,xpar,op)
dss_sdc *dss;
ppacket *xpar;
unsigned op;
{
   unsigned *param;

   set_cursor(dss,xpar->_p1,xpar->_p2);
   param = dss->param;

   param[0] = 3;
   param[1] = param[4] = xpar->_p3 - 1;
   param[2] = xpar->_p4 - 1;
   param[3] = 0x3fff;                  /* -1 in 14-bit 2's complement */

   set_figure(dss,op,5);
   writecmd(fisd);
}               /* end of rectangle */



line(dss,xpar)
dss_sdc *dss;
ppacket *xpar;
{
   int delx,dely;
   char op,index;
   unsigned *param;

   index = 0;
   delx = xpar->_p3 - xpar->_p1;
   dely = xpar->_p4 - xpar->_p2;

/* formulate index into the direction look-up table */
   if (delx < 0)
     { index += 8;
       if (delx == 0) index += 3;
     }
   else if (dely == 0) index += 2;
```

```
    if (dely < 0) index += 4;
    if (delx == dely) index += 2;
    delx = abs(delx);
    dely = abs(dely);
    if (delx > dely) index+= 1;

 /* set the coded value of direction */
    op = dss->table[index];

 /* X axis is always independent axis */
    if (op & 0x01) xswap(&delx,&dely);

    param = dss->param;
    param[0] = delx;
    param[3] = dely << 1;
    param[1] = (param[3] - param[0]) & 0x3fff;
    param[2] = (dely - param[0]) << 1 & 0x3fff;

    set_cursor(dss,xpar->_p1,xpar->_p2);
    set_figure(dss,op >> 1,4);
    writecmd(fisd);
}




set_pattern(dss,pattern)
dss_sdc *dss;
unsigned pattern;
{ dss->pattern = pattern;
  writecmd(pram+8);
  write2par(pattern);
}




set_mode(dss,mode)
dss_sdc *dss;
char mode;
{ dss->mode = mode;
  writecmd(wdat+tfr_wd+mode);
}




arc(dss,xpar)
dss_sdc *dss;
ppacket *xpar;
```

```
{
  char *dir,            /* pointer to directon table */
        si,fi,          /* octant = index into direction table */
        sa,fa;          /* starting and ending angle of an arc */

  dir = dss->atable;
  si = xpar->_p4 / 45;            /* set octant of starting angle */
  fi = (xpar->_p5 - 1) / 45;     /* set octant of ending angle */
  sa = xpar->_p4 % 45;        /* set starting angle in an octant */
  fa = (xpar->_p5 - 1) % 45 + 1;      /* same for ending angle */

  if (si == fi)                   /* if an arc is in one octant */
    A_arc(dss,xpar,sa,fa,dir[si]);
  else                            /* else more than in one octant */
    { A_arc(dss,xpar,sa,45,dir[si]);
      for (si++ ; si < fi ; si++)
        A_arc(dss,xpar,0,45,dir[si]);
      A_arc(dss,xpar,0,fa,dir[si]);
    }

  noerr();
}




set_ccc(dss,xpar)
dss_sdc *dss;
ppacket *xpar;
{ writecmd(cchar);
  writepar(xpar->_p1 - 1 | 0x80);
  write2par(0x3bc1);
  noerr();
}




partition(dss,xpar,area)
dss_sdc *dss;
ppacket *xpar;
unsigned area;
{
  unsigned temp;

  writecmd(pram + area);
  write2par(xpar->_p1);
  temp = xpar->_p2 << 4;
  if (xpar->_p4)  temp |= 0x4000;
  write2par(temp);
  noerr();
}
```

```
dma_write(dss,xpar)
dss_sdc *dss;
ppacket *xpar;
{
   unsigned *param;

   set_cursor(dss,xpar->_p1,xpar->_p2);

   param = dss->param;
   param[0] = xpar->_p4 - 1;
   param[1] = (xpar->_p3 << 1) - 1;

   set_figure(dss,4,2);

   writeDMAC(dss,dss->_p5,dss->_p4 * (dss->_p3 << 2),1);
   writecmd(dmaw + mod_set + tfr_wd);

   noerr();
}




dma_read(dss,xpar)
dss_sdc *dss;
ppacket *xpar;
{
   unsigned *param;

   set_cursor(dss,xpar->_p1,xpar->_p2);

   param = dss->param;
   param[0] = xpar->_p4 - 1;
   param[1] = (xpar->_p3 << 1) - 1;
   param[2] = param[1] >> 2;

   set_figure(dss,4,3);

   writeDMAC(dss,dss->_p5,dss->_p4 * (dss->_p3 << 2),0);
   writecmd(dmar + mod_set + tfr_wd);
   noerr();
}
```

```
/* This is sksutil.c file */
/* This file contains the utility routines that SETSTAT */
/*        functions call directly to program the GDC */



#include <sks.h>
#include <sksop.h>
#include <skssvc.h>

#define odd(x) (x & 0x01)

unsigned div16(),mul16();



set_cursor(dss,x,y)                          /* move cursor to (x,y) */
dss_sdc *dss;
unsigned x,y;
{
  unsigned ead,                              /* word address */
           xp;

  writecmd(curs);
  xp = dss->wxmin + x;                       /* map the point to frame */

  ead = (dss->wymin + y) * dss->fxmax + div16(xp);
  write2par(ead);                            /* word address */
  writepar(mul16(xp & 0x000f));              /* dot address */
}



set_figure(dss,op,num)
dss_sdc *dss;
unsigned op;
char num;
{
  unsigned *param;

  writecmd(figs);
  writepar(op);

  param = dss->param;
  for ( ; num ; num--)
    write2par(*param++);
}
```

```
set_mask(pattern)
unsigned pattern;
( writecmd(mask);
  write2par(pattern);
)


write_data(op,pattern)
unsigned op,pattern;
( writecmd(op);
  write2par(pattern);
)


set_zoom(dss,factor)
dss_sdc *dss;
char factor;
( char *port;

  port = dss->port + zoomadrs;
  dss->zoomf = factor;              /* update zoom factors */
  writecmd(zoom);
  writepar(factor);
  *port = ~factor;                  /* inverted data to pre-scaler */
)


A_arc(dss,xpar,sa,fa,dir)
dss_sdc *dss;
ppacket *xpar;
char sa,          /* starting angle of the arc */
     fa,          /* ending angle of the arc */
     dir;         /* direction */
(
  unsigned *param,radius,x,y;
  char t;
  long int temp_long;

  param = dss->param;

  if odd(dir)    /* if direction is odd */
   ( t = sa;
     sa = 45 - fa;
     fa = 45 - t;
   )
```

```
    radius = xpar->_p3;

    temp_long = (radius * sin1k(fa)) / 1000;
    param[0] = temp_long;
    param[1] = radius;
    param[2] = (radius - 1) << 1;
    param[3] = 0x3fff;                          /* -1 in 14-bit 2's comp. */

    if (sa)                                     /* if sa is not 0 */
      { temp_long = (radius * sin1k(sa)) / 1000;
        param[4] = long_temp;
      }
    else param[4] = radius;

    x = xpar->_p1;                              /* x coordinate of center */
    y = xpar->_p2;                              /* y coordinate of center */

    if        (dir == 0 || dir == 3) x -= radius;
    else if (dir == 1 || dir == 6) y -= radius;
    else if (dir == 2 || dir == 5) y += radius;
    else if (dir == 4 || dir == 7) x += radius;

    set_cursor(dss,x,y);
    set_figure(dss,0x20 + dir,5);
    writecmd(fisd);
}




set_fill(dss,xpar)
dss_sdc *dss;
*packet *xpar;
{ unsigned *param;
  char i;

  writecmd(pram+8);
  param = &(xpar->_p4);
  for (i = 4 ; i ; i--)
    write2par(*param--);
}




draw_fill(dss,xpar)
dss_sdc *dss;
ppacket *xpar;
{ unsigned *param;

  set_cursor(dss,xpar->_p3,xpar->_p2);   /* cursor at (x2,y1) */
```

```
    param = dss->param;
    param[0] = xpar->_p4 - xpar->_p2 - 1;
    param[1] = param[2] = xpar->_p3 - xpar->_p1;

    set_fisure(dss,0x16 + xpar->_p5,3);
    writecmd(schrd);
}




writeDMAC(dss,sa,nb,op)
dss_sdc *dss;
unsigned sa,           /* startins address */
         nb;           /* number of bytes to be moved */
char op;               /* 0 for DMA READ, 1 for DMA WRITE */
{ unsigned *port;      /* pointer to a word in memory */
  char *cport;         /* pointer to a byte in memory */

  port = dss->port + dma3port;  /* point to port 3 of DMAC */
  *port++ = sa;                      /* write the startins address */
  *port   = nb;                 /* write number of bytes to move */

  cport = dss->port + channel_cont; /* point to channel control */
  *cport++ = 0x0a + op;        /* write to control resister */
  *cport++ = 8;                /* write to priority control res */
  *cport++ = 0;                /* write to interrupt control res */
  *cport   = 0;                /* write to data chain res */
}
```

```c
/* This is util.c file */
/* This file contains the most basic utility routines */


#include <sks.h>

writecmd(c)
char c;
{ char *port;
  port = (set_dss())->port + sdcbase;     /* point to GDC */
  if ( c )                                /* if not reset command */
    while ( *port & 0x02 );               /* wait until fifo not full */
  *++port = c;                            /* output opcode */
}


writepar(c)
char c;
{ char *port;
  port = (set_dss())->port + sdcbase;
  while ( *port & 0x02 );                 /* wait until fifo not full */
  *port = c;                              /* output parameter */
}


write2par(param)
unsigned param;
{ writepar(param);                        /* output low byte only */
  writepar(swap(param));                  /* output high byte */
}


unsigned div16(num)
unsigned num;
{ return(num >> 4); }


unsigned mul16(num)
unsigned num;
{ return(num << 4); }


abs(num)
int num;
{ return((num < 0) ? -num : num); }


xswap(x,y)
unsigned *x,*y;
{ unsigned temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

APPENDIX L

PARTS SUMMARY AND LIST

PARTS LIST

U1	GEN_JUMPER
U2	GEN_JUMPER
U3	PAL12L6
U4	PAL12L6
U5	74S00
U6	74S04
U7	74S08
U8	74S74
U9	J3
U10	J1 (TOP OF EDGE CONNECTOR)
U11	J2 (BOTTOM OF EDGE CONNECTOR)
U12	74S37
U13	74S02
U14	74S244
U15	74S244
U16	74S74
U17	CLOCK_GENERATOR
U18	GEN_CAPACITOR-1 (10 MICRO FARAD)
U19 - U21	GEN_CAPACITOR-2 (0.1 MICRO FARAD)
U22	74S257
U23	GEN_CAPACITOR-2
U24	74S257
U25	GEN_CAPACITOR-2
U26	74S374
U27 - U28	GEN_CAPACITOR-2
U29	NEC7220
U30 - U35	GEN_CAPACITOR-2
U36	74S175
U37	74S163
U38	GEN_CAPACITOR-2
U39	74S163
U40	74S175
U41 - U44	GEN_CAPACITOR-2
U45	74S175
U46 - U69	GEN_CAPACITOR-2
U70	4164
U71	74S299
U72 - U73	GEN_CAPACITOR-2
U74 - U75	4164
U76	74S299
U77	GEN_CAPACITOR-2
U78 - U90	4164
U91	GEN_RESISTOR-5
U92	GEN_RESISTOR-4
U93	GEN_RESISTOR-3

```
U94          74S04
U95          GEN_RESISTOR-3
U96          GEN_RESISTOR-3
U97          74LS21
U98          GEN_CAPACITOR-2
U99          MC6844
U100         74LS174
U101         74LS245
U102         74LS245
U103         74LS245
U104         74LS245
U105         BERG40
U106 - U110  GEN_RESISTOR-2 for pull-up resistors
U111- U240   GEN_RESISTOR-1 used for plated holes
U244         74S21       97P, 98P in page 4
U245         74S175      59P in page 3
U247         74S32       61P, 62P in page 3
U248         74S299      8P in page 8
U249         27256       20P in page 8
U250         74S163      14P in page 8
U251         74LS273     15P, 16P in page 8
U252         74S74       2P, 6P in page 8
U253         74S163      3P in page 3
U254         14 PIN DIP FOR COLOR COMPUTER INTERNAL BUS
```

PART SUMMARY

4164		16
74LS174		1
74LS21		1
74LS245		4
74LS273		1
74S00		1
74S02		1
74S04		2
74S08		1
74S163		4
74S175		4
74S21		1
74S244		2
74S257		2
74S299		3
74S32		1
74S37		1
74S374		1
74S74		3
BERG40		1
CLOCK_GENERATOR		1
GEN_CAPACITOR		0
GEN_CAPACITOR	DIPTANT	1
GEN_CAPACITOR	CK05BX-104M	46
GEN_JUMPER		2
GEN_RESISTOR		0
GEN_RESISTOR	RPACK	153
GEN_RESISTOR	RCR07G102JM	5
GEN_RESISTOR	RCR07G220JM	3
GEN_RESISTOR	RCR07G271JM	1
GEN_RESISTOR	RCR07G331JM	1
J1		1
J2		1
J3		1
MC6844		1
NEC7220		1
PAL12L6		2
27256		1
Total		273

REFERENCES

[AHRE 81]   Ahrens, T., et al.
             What's Inside Radio Shack's Color Computer.
             BYTE Magazine : 90-129, March, 1981.

[BALA 83]   Balachandran, S.
             Workstation Based Lookahead Network - The Network Interface.
             Master's thesis, Dept. of Electrical Engineering, University of Texas at
             Austin, Aug., 1983.

[CHAM 85]   Champine, G.
             "Course Notes for EE397K Advanced Computer Graphics."
             Dept. of Electrical Eng., University of Texas at Austin, Jan., 1985.

[CONR 85]   Conrac Corporation.
             Raster Graphics Handbook, 2nd Ed.
             Van Nostrand Reinhold Company Inc, 1985.

[ENDE 84]   Enderle, G., et al.
             Computer Graphics Programming
             Springer-Verlag, 1984.

[FOLE 82]   Foley, J.D. and Van Dam, A.
             Fundamentals of Interactive Computer Graphics
             Addison Wesley Publishing Company, Inc., 1982.

[LIPO 82]   Lipovski, G.J.
             "Instructions for Using the Trace Program"
             1982.

[NEC 82a]   "NEC uPD7220 Graphics Display Controller Data Sheet."
             NEC Electronics U.S.A. Inc., 1982.

[NEC 85]   "NEC uPD7220 GDC User's Manual"
             NEC Electronics U.S.A. Inc., 1985.

[OS-9C 83]   "C Compiler User's Guide"
             Microware Systems Corporations, 1983.

[OS-9S 84]   "OS-9/6809 Operating System System Programmer's Manual"
             Microware Systems Corporations, 1984.

VITA

Seungyoon Peter Song was born in Seoul, Korea on September 19, 1961, the son of Bok-Joo and Chi-Soon Song. He attended the Hillsboro High School in Nashville, Tennessee from 1977 to 1979. He then entered the University of Texas at Austin to major in electrical engineering in 1979. From August of 1981 to August of 1982, he worked as an undergraduate research assistant in Chemical Engineering Department. After receiving the Bachelor of Science degree in Electrical Engineering in August of 1982, he entered the Graduate School of The University of Texas at Austin to continue the study in electrical and computer engineering. During the first two years of graduate school, he worked as a technical assistant in the Safety Office, Department of Planning. From September of 1984 to May of 1985, he worked as a teaching assistant in the Electrical and Computer Engineering Department. The following year, he worked as a research assistant.

Permanent address: POBOX 8600
                        Austin, Tx. 78713

This thesis was typed by the author himself.