THE STELLA SYSTEM TRAINING MANUAL

Version 1.1

Chris Powell

February 1983

INTRODUCTION

This course takes approximately 4 weeks and assumes that you know 6502 assembly language and Stella development system operations. This training manual is presented as a complement to the Stella Programmer's Guide by Steve Wright and the TIA Manual which should be read concurrently. The Stella system is presented in logical blocks each consisting of a reading unit followed by a programming exercise. The exercises take typically three to four working days.

When each exercise is finished it should be discussed with the instructor who will run it on a development station to check frame timing and all numerical parameters. The code will be examined to see that the problem was approached in a logical and intelligent way. For the first exercises the code will not be checked for optimal use of RAM, ROM or CPU time.

All the exercises together are designed to fully exercise the Stella system, using most TIA registers. The course covers Stella programming, including the philosophy of kernel design and techniques for saving RAM, ROM and CPU time. It is intended to be a general strategic guidance of software approaches and overall organizational strategies to enhance quick, neat and efficient cartridge development.

Code segments and tables of values are given as examples and are not necessarily only correct solutions to the problem at hand.

This manual is based on the Stella Training Lecture Series conducted by Steve Wright for his department and the contents of this manual were checked for correctness by him.

UNIT1

The material covered in the first unit will include:

- Stella System Architecture
- Stella System Memory Map
- Frame Timing
    - Vertical Timing
    - Horizontal Timing
- Code Structure
- Playfield
- Exercise
    - Registers Needed
    - Programming Hints
- Related Reading

STELLA SYSTEM ARCHITECTURE

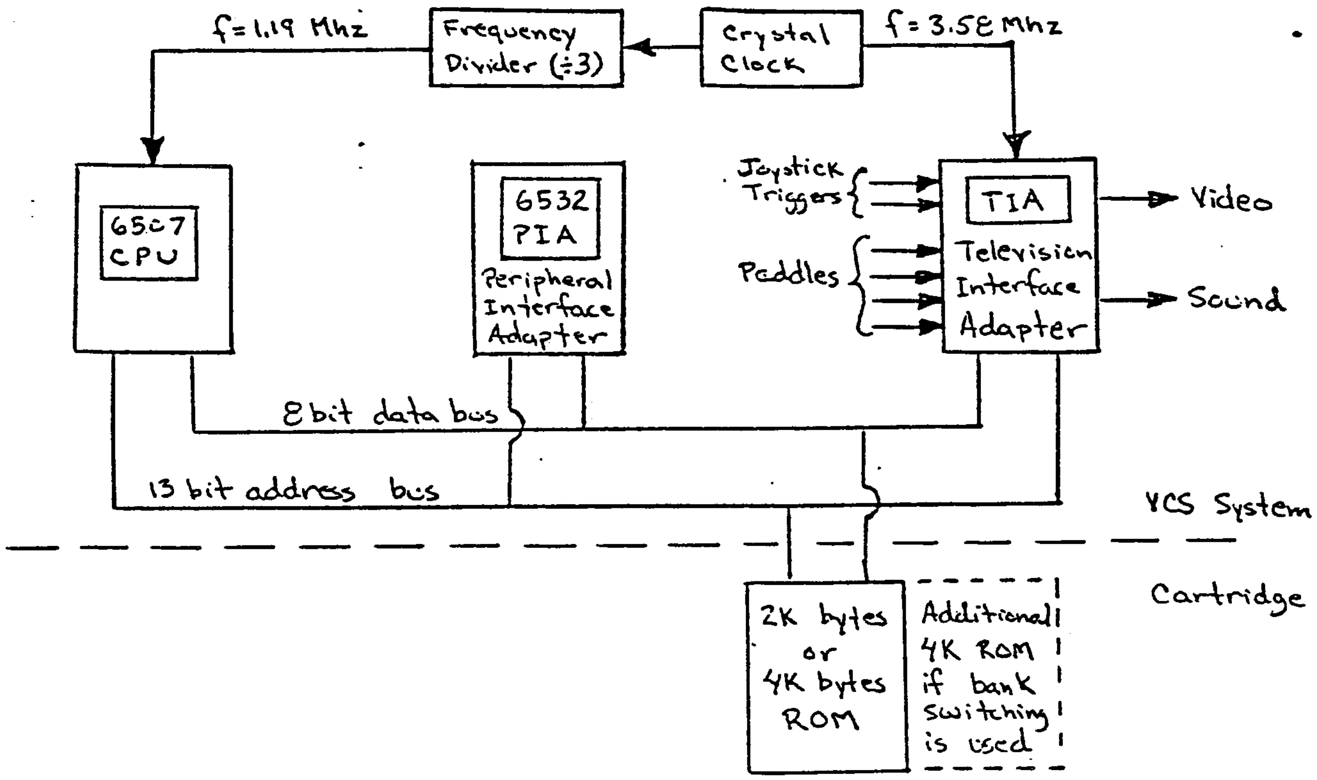


Figure 1.1. Stella System High Level Architecture

There are only enough address lines out to the cartridge socket to accept 4K of addresses, there are no clock, or data read/write lines (see Figure 1.1). Multiples of 4K may be used by using the bank switching technology. By addressing a reserved address in a custom chip the second 4K bank is accessed. Although in theory this process could be repeated indefinately (i.e. 12K, 16K, etc.) only 8K has been implemented to date.

One CPU machine cycle takes the same amount of time as three color clocks on the TIA. This means that the CPU works relatively slowly compared to the TIA. For example a load immediate instruction of 2 cycles and store to a TIA register of 3 cycles uses a total of 5 machine cycles which is 15 color clocks, about 6.5% of a complete scan line. Because both the CPU and the TIA are driven by the same clock crystal they are synchronized.

The PIA contains the following:

- 128 bytes of RAM
- Programmable Timer
- Two 8-bit I/O ports comprised of:
  - a) Controller ports
    - 4 right port bits
    - 4 left port bits
  - b) Console switches
    - Left difficulty
    - Right difficulty
    - Game select
    - Game reset
    - color/ black and white switch
    - 3 unused pins

The programmable timer is set with two pieces of information. One is the address into which the data is loaded which indicates the timer speed. The second is the actual data loaded which is used as the initial value for the countdown timer. For example if you load 100 into Count3 then the timer counts down from 100 at Count3 rate. The timer is most often used in timing the vertical interval of the TV frame.

The controller ports will be discussed in Unit 4.

STELLA SYSTEM MEMORY MAP

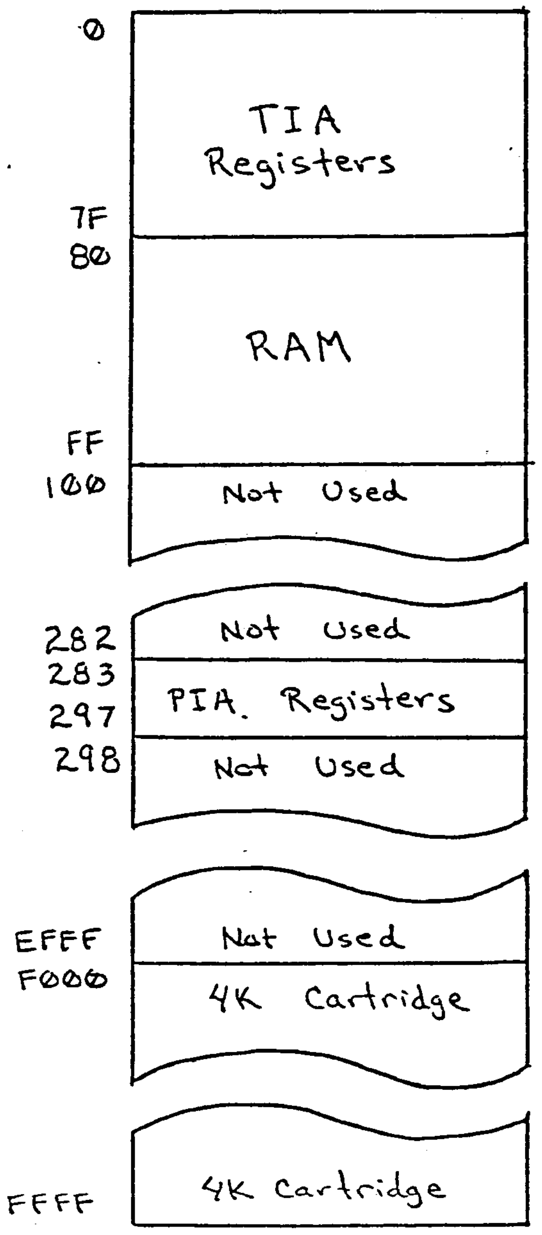
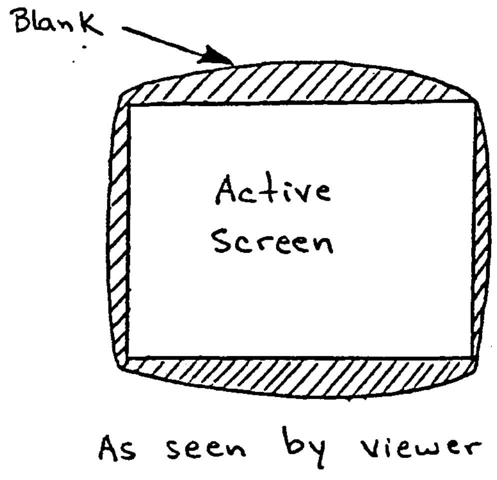


Figure 1.2. Stella System Memory Map



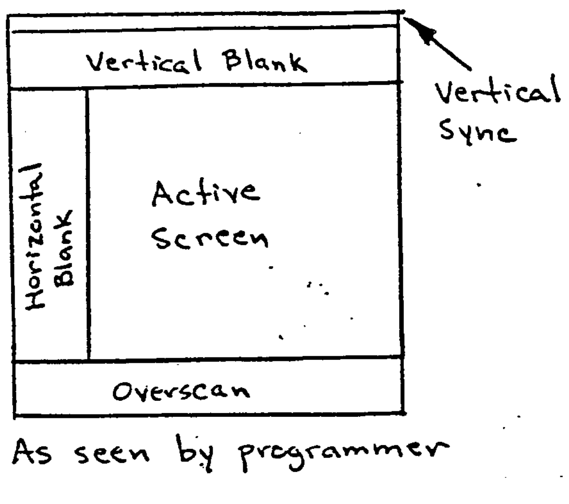


Figure 1.3. TV Screen - Two views

Figure 1.3 shows the TV screen as seen by the viewer and as seen by the programmer showing the programmatic names for the various portions of the screen and their corresponding portions in the TV image. During the scanning of the television screen there are two distinct processes, the vertical deflection and the horizontal deflection. The electron beam starts at the upper left corner of the screen and sweeps to the right at a slight downward slope caused by the on-going vertical deflection. At the end of the scan line the horizontal blank signal turns off the electron beam and deflects the scan back to the left side of the screen. This is called the HSYNC signal and is used for horizontal synchronization. This process is repeated continually by the TV set until it receives a vertical synchronization signal (VSYNC) and the beam is returned to the top of the screen. This entire process is repeated 60 times per second.

Vertical Timing

At the vertical synchronization portion of the frame the VSYNC register of the TIA is set and the TIA outputs a continuous vertical synchronization signal to the TV (See Example 1.1). The software counts a minimum of 3 scan lines of vertical synchronziation then resets the VSYNC register. The TV will accept from 3 to 5 scan lines of vertical synchronization but 3 or fractionally more is the games standard.

Variations in the amount of overscan by different TV manufacturers cause the amount of picture cropping at the top and bottom of the screen to differ. For this reason the standard 192 lines per active screen leaves blank (black) at the top and bottom of the screen.

In the overscan area the PIA timer is used to determine the end of the 30 scan lines (about 1910 microseconds or 2200 cycles)(See Example 1.2). This allows intervening program logic before the timer is read. All possible logic threads must complete in time to check the timer to determine if the proper time length has passed. The vertical blank area is similarly timed with the PIA timer (about 2350 microseconds or 2700 cycles). The software usually counts the 192 scan lines (about 12470 microseconds or 14592 cycles) of the active screen instead of using a timer because the displayed information is line dependent but the timer may be used equally well.

```
LDA      #02      ;TURN VSYNC ON
STA      VSYNC

STA      WSYNC    ;WAIT FOR AT LEAST 3 LINES OF VSYNC
STA      WSYNC
STA      WSYNC
STA      WSYNC

LDA      #0       ;TURN OFF VSYNC
STA      VSYNC
```

Example 1.1. Vertical Synchronization Code

```
          LDA      #N       ;SET VBLANK/OVERSCAN TIMER
          STA      TIM64T

;         THE PROGRAM MAY DO ANYTHING HERE BUT MUST
;         FINISH BEFORE THE TIMER EXPIRES

LOOP:     LDA      INTIM    ;WAIT FOR TIMER
          BNE      LOOP
```

Example 1.2. Overscan/Vertical blank timing code
N is some number to be determined

Horizontal Timing

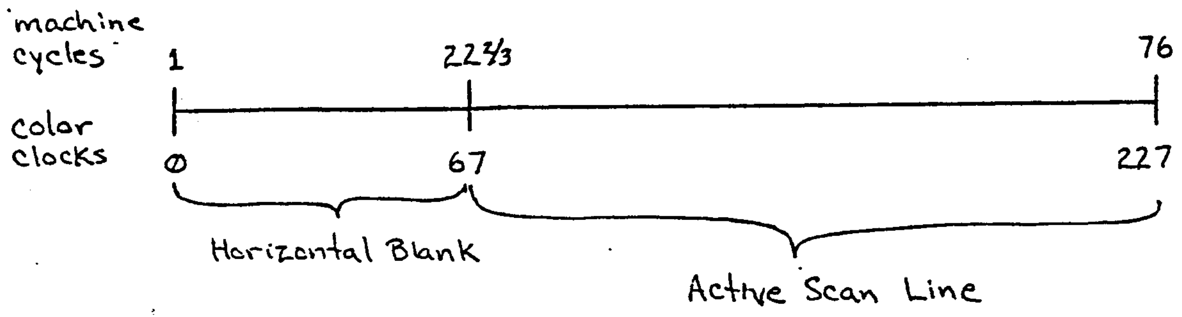


Figure 1.4. The Scan Line

The horizontal synchronization signal (HSYNC) is generated automatically by the TIA chip, beginning the horizontal blank time. The horizontal blank time (22 2/3 machine cycles) is the time from the completion of one active screen line to the beginning of the next active screen line (see Figure 1.4). This time may be used programatically to set up for the upcoming scan line. There are 160 visible color clocks per active scan line, hence 160 pixels. Each moveable object may be on each of the 160 positions regardless of the objects own resolution.

Strobing the WSYNC register causes the CPU processing to be suspended until the beginning of the horizontal blank of the next scan line. At that time the CPU processing is restarted. This is used to synchronize processing with the beginning of the scan line.

CODE STRUCTURE

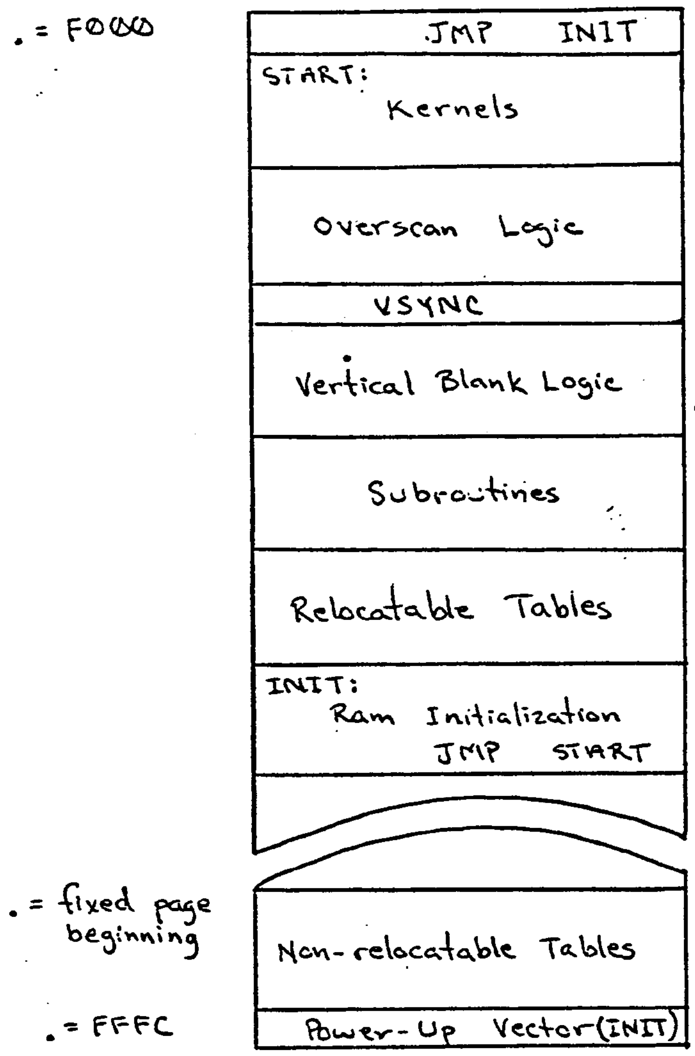


Figure 1.5. VCS Code Structure

The development station does not enter programs at FFFC like the VCS does at power-up, it enters at the first executable address, for example F000. Therefore the first instruction at F000 should be a jump to the initializing routine (See Figure 1.5). At FFFC is the power-up vector which should contain the address of the initializing routine. The VCS automatically jumps to the address at power-up. In this way the code will jump to the initializing routine first for both the development station and the VCS.

The kernel is the part of the game program that executes during the visible portion of the screen. It is responsible for setting the graphics registers to obtain the desired graphic effects. The kernels should follow the jump instruction, beginning at F003. It is important that the kernels are aligned to page boundaries so that branches do not cross page boundaries needlessly. Recall that branching across page boundaries requires an extra machine cycle to execute (4 cycles instead of 3). The additional cycles will prove to be very detrimental to effective kernel design. To insure the alignment during the addition of program code the kernels are the first code in the program.

The Vertical Blank logic and Overscan logic are the game logic sections of the code. They execute during the vertical blank and overscan because nothing is being displayed on the screen allowing free use of the CPU without worry over synchronization with the TIA and television until the end of each section.

Graphics tables, used during the kernels, are non-relocatable tables. They should be aligned with page boundaries so that references to them do not cross page boundaries needlessly. Recall, again, that references that cross page boundaries require an extra machine cycle. Moving these tables during development is possible in 1-page increments to insure page alignment.

PLAYFIELD

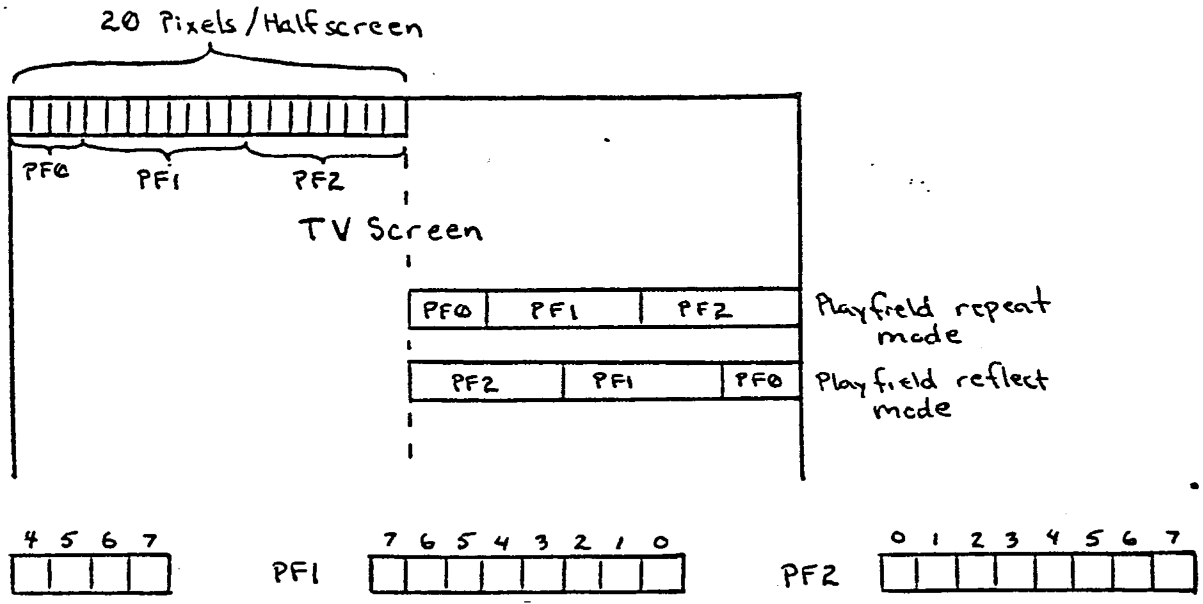


Figure 1.6. Playfield Registers and Modes (not to scale)

The playfield is a bit map for the screen at low resolution. Each half of the screen is 20 pixels wide so there are four color clocks per pixel. Where the register value for the pixel is zero the background color shows through. Where the register value for the pixel is one the playfield color is displayed. Playfield repeat mode causes the right half of the screen to be an exact duplicate of the left. Playfield reflect mode causes the right half of the screen to be a mirror image of the left.

Figure 1.6 shows the alignment of the playfield registers on the left side of the screen, the alignment of the playfield registers on the right side of the screen for both repeat and reflect modes, the playfield resolution in the number of pixels per half screen and the bit alignments for the playfield registers on the left side of the screen.

The graphics registers in the TIA act as parallel to seriel converters. As the electron beam moves left to right across the screen it reads the bits in the specified order. Therefore the first pixel is bit 4 of Playfield register 0 (PF0), the second pixel is bit 5 of PF0, etc.

EXERCISE

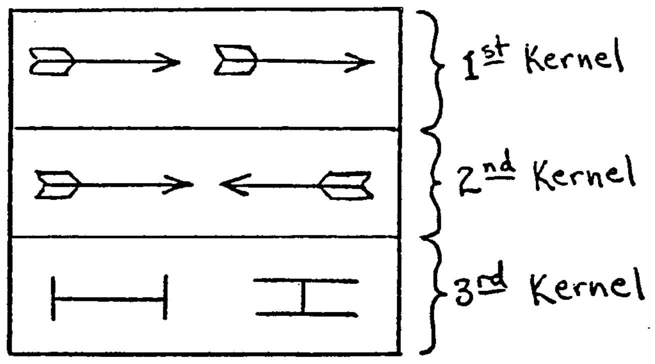


Figure 1.7. The exercises display

On the top third of the screen draw an arrow shaped playfield on the left and repeat the same shape on the right (see Figure 1.7). On the middle third of the screen draw the same shape on the left and its mirror image on the right. On the bottom third of the screen draw a playfield in the shape of a long, narrow I-beam on the left and a playfield in the shape of a short, wide I-beam on the right.

The program will need correct vertical timing, a background color and tables of data to describe the playfield shapes. Use the logic analyzer to verify that the active screen is 192 scan lines and that the frame time is 16,686 microseconds ± 6 microseconds. The data tables may be zero filled to cover a full third of the screen so the kernel does not have logic to determine when to display a blank playfield.

The first two thirds of the screen may use a two-line kernel(see Example 1.3), setting the playfield registers twice every scan line. A one-line kernel must be used for the bottom third of the screen because it is asymmetrical and the playfield registers must be set once every half line (See Example 1.4).

```
;       KERNEL PREPARATION
        LDY     #96.    ;192 ACTIVE LINES/2 ACTIVE LINES PER LOOP
;       FIRST LINE OF KERNEL
LOOP:   STA     WSYNC   ;WAIT FOR BEGINNING OF LINE
        LDA     T1,Y    ;FROM TABLE 1
        STA     PF0
        LDA     T2,Y    ;FROM TABLE 2
        STA     PF1
        LDA     T3,Y    ;FROM TABLE 3
        STA     PF2
;       SECOND LINE OF KERNEL
        STA     WSYNC   ;WAIT FOR BEGINNING OF LINE
        DEY             ;DECREMENT COUNTER
        BPL     LOOP    ;CONTINUE IF NOT ZERO
```

Example 1.3. Two line kernel with playfield tables

```
;          END OF VERTICAL BLANK
           LDA      #0        ;TURN OFF VBLANK
           STA      WSYNC     ;AT THE BEGINNING OF A LINE
           STA      VBLANK
;          KERNEL PREPARATION
           LDY      #192.     ;192 SCAN LINES PER ACTIVE SCREEN
;          KERNEL - COUNTS ACTIVE SCAN LINES
LOOP:      STA      WSYNC     ;WAIT FOR BEGINNING OF LINE
           DEY                ;DECREMENT COUNTER
           BPL      LOOP      ;CONTINUE IF NOT ZERO
;          OVERSCAN BEGINNING
           STA      WSYNC
           LDA      #02       ;TURN VBLANK ON
           STA      VBLANK
```

Example 1.4. Minimal 1-line kernel

The minimal 1-line kernel displays nothing but counts the required 192 scan lines for the active screen.

Registers Needed

- INTIM - The PIA timer read register; polled to determine the end of the vertical interval.
- TIM64T - PIA timer interval set register; set to different values to initiate vertical interval timing.
- VBLANK - Controls vertical blanking for the vertical interval.
- VSYNC - Controls the vertical synchronization signal.
- WSYNC - Synchronizes the processor with the beginning of the next scan line.
- COLUBK - Sets the background color and luminosity.
- COLUPF - Sets the playfield color and luminosity.
- PF0, PF1, PF2 - Set the playfield graphics.
- CTRLPF - Sets the playfield reflect/repeat mode.

Coding Hints

- It is suggested that the programmer use an already constructed register address equates file and the routine to zero RAM (See Example 1.5). The later is most important because the system has an anomoly that if RAM is zeroed from the high addresses to the low addresses the system can lock up (This will happen if the store to VSYNC happens at cycle 76 in a scan line.).

```
INIT:      CLD                    ;INSURE BINARY ARITHMETIC MODE
           SEI                    ;NECESSARY FOR KEYBOARD INPUT
           LDX      #0FF          ;INITIALIZE STACK POINTER
           TXS
;
;          CLEAR RAM
;
           LDA      #0
CLERAM:    DEX                    ;DECREMENT COUNTER
           STA      STACK(X);ZERO RAM
           BNE      CLERAM  ;CONTINUE IF NOT ZERO
```

Example 1.5. Typical program start and RAM zeroing routine

- It is further suggested that the problems are isolated and solved sequentially (i.e. the frame timing first, repeat mode second, etc.) for ease of debugging.
- Use playfield repeat mode for creating asymmetrical playfield to avoid problems assigning values to PF2.
- For the exercise it may be easier to copy the code for the 2-line kernel for the first third of the screen and use it for the 2-line kernel for the second third of the screen.
- In the last third of the screen, a 1-line kernel, the second store to each PF register must not take place until after the entire first value is displayed. So, spread out the stores to the PF registers using NOPs to insure correct timing. The NOPs will be replaced later with loads and stores for other graphics registers.

RELATED READING

Television Protocol (Bound with the Stella Programmers Guide), Steve Wright, 11/29/79
Stella Programmers Guide, Steve Wright, 12/03/79
- (Beginning through section 5.0) General Description, The Registers, Synchronization, Color and Luminosity, and Playfield

The PIA (Bound with the Stella Programmers Guide), Steve Wright, 12/03/79
- (Beginning through section 3.0) General, Interval Timer, Setting the Timer, Reading the Timer, When the Timer Reaches Zero, and RAM.

Stella Package - Bank Switching Without Even Thinking, Carla Meninksy, 5/27/82
Television Interface Adaptor (TIA) Manual
- Descriptions of needed registers

UNIT 2

The material covered in the second unit will include:

- Missile and Ball movement
  - Character Reset (CHRST)
- Missile and Ball enable (M0, M1, BL)
- Advanced Topics
  - Reincarnation
  - Alternate CHRST Loop
- Exercise
  - Registers Needed
  - Programming Hints

At the end of the unit you will be able to display missiles and the ball and move them on the screen.

MISSILE AND BALL MOVEMENT

As a model of the moveable object graphics you may consider the vertical position of the object to be kernel controlled. For an object that is supposed to be on line 100 and is 8 lines tall the kernel counts the lines turning the ball on at line 100 and turning it off at line 108. For the same model you may consider the horizontal position of the objects to be controlled by character reset (CHRST) routines.

CHRST Routines

The character reset routines apply to missiles, the ball and the players. To position an object in a particular horizontal position (a column) you must strobe the character reset register for that object at that position in a scan line, any scan line. Because the usual method uses an entire scan line this is usually done during the vertical interval. For example to move the ball to a column position you must store any value into the RESBL register at the same column position during a scan line, that is its when you store to RESBL that is important not what you store. From that point on whenever the ball is enabled during the kernel it will appear with its left edge in that position.

```
STA     WSYNC    ;WAIT FOR BEGINNING OF LINE
NOP              ;20 NOPS: 40 CYCLES
NOP
.
.
.
NOP
STA     RESBL    ;STROBE RESET BALL
```

Example 2.1. Hard Coded Horizontal Positioning

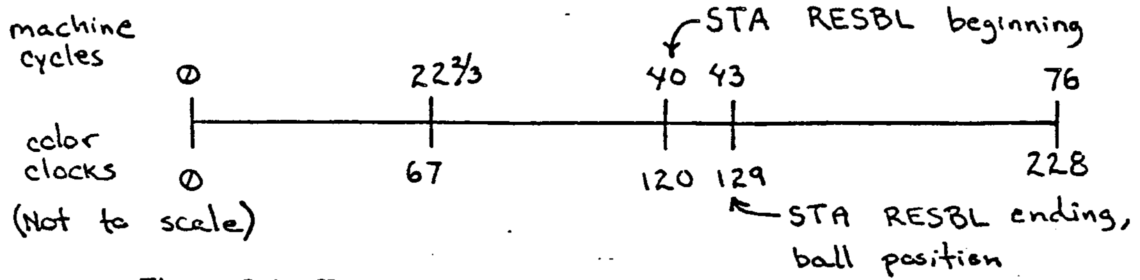


Figure 2.1. Horizontal Position of STA RESBL in Example 2.1

Example 2.1 shows the NOP method for moving the ball to a fixed column location. Notice that this code segment will always put the ball in the same location (See Figure 2.1). It is impractical to use the NOP method for all the possible positions on a scan line so another method of positioning on scan lines is needed.

Example 2.2 shows a flexible method of coarse positioning. In the example each loop takes 2+3=5 cycles until the loop in which the final BPL test fails taking 2+2=4 cycles and the strobe of the reset register takes 3 cycles. For n=(COARSE) time=5n+4+3=5n+7 cycles. The loop allows 5 cycle or 15 clock resolution in the positioning of an object.

```
        LDY    COARSE   ;SET UP COUNTER
        STA    WSYNC    ;WAIT FOR BEGINNING OF LINE
LOOP:   DEY             ;DECREMENT COUNTER - 2 CYCLES
        BPL    LOOP     ;LOOP IF NOT ZERO - 3 CYCLES/2 CYCLES
        STA    RESBL    ;STROBE BALL RESET - 3 CYCLES
```

Example 2.2. Coarse Horizontal Positioning

Because of the 15 clock granularity of positioning with the CHRST loop, the TIA also has horizontal motion registers. The reset registers are used for coarse positioning, the horizontal motion registers are used for fine positioning. Thus the reset registers position an object to 15 clock positions and the horizontal motion registers position the object as much as 7 clocks to the left or 8 clocks to the right of the coarse position giving complete 1 clock resolution. Example 2.3 shows the complete positioning algorithm. In the example the CHRST loop ending with the RESBL does the coarse positioning and setting HMBL to (FINE) and the strobe of HMOVE does the fine positioning.

```
        LDA    FINE     ;SET UP FINE POSITION VALUE
        LDY    COARSE   ;SET UP COARSE POSITION VALUE
        STA    WSYNC    ;WAIT FOR BEGINNING OF LINE
LOOP:   DEY             ;DECREMENT COARSE POSITION COUNTER
        BPL    LOOP     ;LOOP IF NOT ZERO
        STA    RESBL    ;STROBE BALL RESET
        STA    HMBL     ;LOAD FINE POSITION INTO MOTION REGISTER
        STA    WSYNC    ;WAIT FOR BEGINNING OF NEXT LINE
        STA    HMOVE    ;MOVE THE OBJECT
```

Example 2.3. Fine Horizontal Positioning

The effect of HMOVE is accumulative. Repeated strobing of the HMOVE register will cause the value of the fine positioning to be added repeatedly to the current position changing it each time. Also, since the electron beam is turned off for the 24 cycles of HMOVE, if HMOVE is used during the active screen black HMOVE lines 8 clocks long will appear at screen left on the lines where the HMOVE strobe was executed. Therefore, in simple practice, the programmer will position all five objects (ball, missiles and players), set all five motion values, then execute HMOVE, once per frame during the vertical interval. Note that the horizontal motion registers may not be changed for 24 cycles after strobing HMOVE. To do so will produce unpredictable object positioning.

The determination of the COARSE and FINE values from the horizontal position of the object may be done in either of two ways. The values may be calculated from the position in real-time or the values may be retrieved from a pre-defined table using the position as an index. The first method is fairly complicated and is considered a number of times in the Stella Packages. The second method will be discussed briefly here.

HORIZONTAL SCREEN POSITION	COARSE POSITION (CHRST LOOP COUNTER)	FINE POSITION (HORIZONTAL MOTION VALUE)
38	5	-5
39	5	-6
40	5	-7
41	5	-8
42	6	7
43	6	6
44	6	5

Figure 2.2. Partial CHRST values table

Since the coarse position is used as a counter with useful values between 5 and 14 (giving on screen coarse positions from 75 clocks to 210 clocks) it may be in the lower nyble of the table entry. Since the fine position is a horizontal motion value being written into bits 4, 5, 6, and 7 of the register it may be in the upper nyble of the table entry. In this way the horizontal position can serve as an index into a 160 byte table (1 byte per position) of the coarse and fine position values (see Figure 2.2).

CHRST tables are ROM intensive but save time. CHRST calculations save ROM (typically about 40 bytes) but use more time. With the advent of 8K cartridges, CHRST tables have become a more viable solution to horizontal positioning calculations.

Horizontal positioning may be done using only HMOVE (no CHRST loop) as follows. During initialization use a "hard wired" code sequence with NOPs, as per Example 2.1, to coarse position the object to screen left. The HMOVE is then used during the vertical interval to move the object to each of its new positions. This has the limitation that the object may only move a maximum of 8 clocks per frame.

ENABLE REGISTERS

```
             LDY      #0        ;INIT BALL OFF
             LDA      VPOSBL    ;VERTICAL BALL POSITION
             SEC                ;PREPARE FOR SUBTRACTION
             SBC      LNCNT     ;KERNEL'S LINE COUNTER -
                                ;DECREMENTED EACH LINE
             CMP      #7        ;WITHIN BALL HEIGHT?
             BCS      SKIP      ;NO - LEAVE BALL OFF
             LDY      #2        ;YES - TURN BALL ON
             .                  ;OTHER OPTIONAL CODE
             .
             .
SKIP         STA      WSYNC     ;WAIT FOR BEGINNING OF LINE
             STY      ENABL     ;ENABLE/DISABLE BALL
```

Example 2.4. Vertical Positioning Code

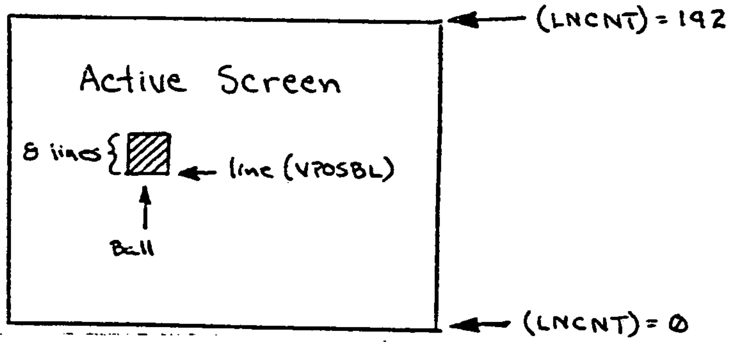


Figure 2.3. Vertical Ball Position from Example 2.4 code

Example 2.4 is the code for vertically positioning the ball on the screen. In the example the VPOSBL variable determines the vertical ball position and the CMP instruction determines the ball height. The two together are sufficient to place the ball on the screen given that it has been previously placed horizontally (See Figure 2.3). This method works will for both missiles and the ball by simple changing ENABL, VPOSBL and #7 to fit the different objects.

The NUSIZ registers contain the bits for setting the number and size of the players and the missile sizes. CTRLPF contains the bits to set ball size. Note that the missiles are directly associated with the players and as multiple copies and color are changed for the players the missiles are changed at the same time. Similarly the color of the ball is associated with the color of the playfield. The horizontal position of the group of missiles or players is the horizontal position of the left most copy of the missile or player.

ADVANCED TOPICS

Reincarnation

Historically, one of the limitations of the VCS was that it had too few sprites (moveable graphics objects). This has been resolved by moving the objects to different screen locations during the kernel allowing the object to be seen two or more times per frame. This is termed "reincarnating" the object.

To reincarnate objects on the screen you may reposition the object using the CHRST and HMOVE method at the cost, in the straightforward case, of 2 scan lines and the last line will have an instruction which strobes the HMOVE register producing a visible HMOVE line. Although the HMOVE lines can not be covered they may be disguised by having black background or a black object at screen left (e.g. Realsports Baseball). HMOVE can also be used during the active screen to cause an object to do incremental motion leaving a trail (e.g. Missile Command).

There is a problem associated with reincarnating objects, it is that the second incarnation and the original incarnation cannot be in the same horizontal band. That is the bottom of the first incarnation cannot be any lower than the CHRST loop for the second incarnation. This is termed vertical seperation.

Alternate CHRST loop

In the horizontal positioning CHRST loop above two entire scan lines are lost because the loop may end at any place in the line. In the situations where time is at a premium, doing reincarnations during the kernel for example, this is not desireable. The following method allows the recovery of almost half of the time lost during CHRST loops.

The scan line is logically divided into two halves, 0 to 37 cycles and 38 to 75 cycles. Obviously any position will lie in one of the two ranges. So the CHRST table entries are considered differently. The high nyble remains the same but in the lower nyble the low bit indicates which half of the scan line the object is in and the other three bits are the coarse position (see Figure 2.4).

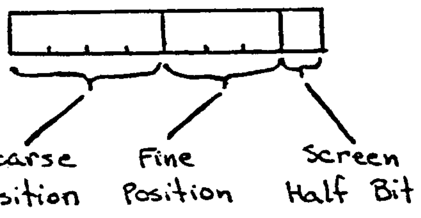


Figure 2.4. CHRST table entry for alternate method.

At the beginning of the scan line the logical half line bit is tested and a decision is made on which of two courses of action to take. If the position is in the right half of the screen then code may be executed, where all paths end at cycle 37, before the CHRST loop is executed with the coarse position. If the position is in the left half of the screen then the CHRST loop is executed immediately and 38 cycles of code may be executed afterward, ending at cycle 75 in the worst case.

Note that this method might be carried one step further. In the lower nyble the low 2 bits would indicate which quarter of the screen the object is in and the upper 2 bits could be the coarse position.

EXERCISE

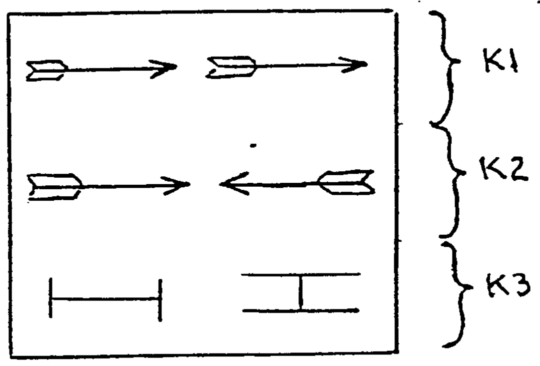


Figure 2.5. First exercise screen

In K3 (See Figure 2.5) move the ball vertically and horizontally continuously using

HMOVE only for the horizontal motion. Seperate the vertical and horizontal movement into different routines. The major issues are limit tests (ball movement only within K3) and ball shaving or compression at the limits.

Move missile 0 (a triple copy missile) and missile 1 (a double copy missile) through K1 and K2 using a CHRST loop. The coarse and fine values may be determined using either CHRST calculations or a CHRST table. For ease of distinguishing them make the missiles different colors. The major issues are limit tests and passing objects between kernels.

Registers Needed

NUSIZ0, NUSIZ1 - Controls the number and size of the missiles.
COLUP0, COLUP1 - Sets the color of the missiles.
RESM0, RESM1 - Resets the coarse horizontal position of the missiles.
RESBL - Resets the coarse horizontal position of the ball
ENAM0, ENAM1 - Enables the missile graphics.
ENABL - Enables the ball graphics.
HMM0, HMM1 - Sets the fine positions of the missiles.
HMBL - Set the fine position of the ball.
HMOVE - Causes the horizontal motion register values to be acted upon setting the fine positions of the missiles, players and the ball.

Programming Hints

- The two 2-line kernels (K1 and K2) will need a line counter to use for positioning the missiles within K1 and K2 which is independent of the kernel specific counter of lines for determining kernel length.
- Turn the background areas for K1, K2 and K3 to different colors to help visually define the limits for the boundary tests.
- Beware that the TIA provides automatic wrap-around side to side. If an object moves off of the screen to the right it will appear at screen left and vice versa.
- The usual method for handling two moving objects in a two-line kernel is to display one on the first scan line and the other on the second scan line.

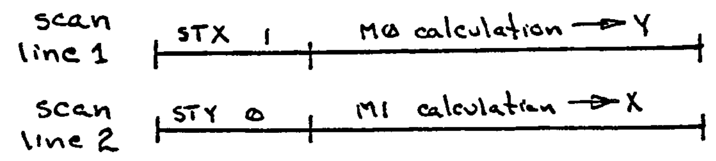


Figure 2.6. Two line kernel displaaying two objects

Figure 2.6 shows that because the graphic stores must happen during the HBLANK a straight-forward method of displaaying two objects in a two-line kernel uses two 6502 registers as well, in this case the X and Y index registers.

Symmetry in two line kernels is desireable to make the passing of graphic objects between the two kernels easier. Objects that are set in the even lines of the first kernel are passed to the even lines of the second kernel. Similarly the objects on odd lines are passed to the odd lines of the second kernel. If there are preoaration lines between the two kernels then it is better to have an even number of them so that the even lines continue to do the even line graphics and the odd lines do the odd line graphics. Another way to make passing objects between kernels easier is to make a line counter global between the two kernels.

RELATED READING

Stella Programmers Guide, Steve Wright, 12/03/79
- (Sections 6.0 to 6.2, 7.0, 8.0, and 9.0) The Moveable Objects Graphics, Missile Graphics, Ball Graphics, Horizontal Positioning, Horizontal Motion, and Object Priorities

Stella Packages - Character Reset Routines For Stella, Richard Maurer,
Stella Packages - Calculate Horizontal Reset (CHRST), Howard Warshaw, 10/13/81
TIA Manual
- Descriptions of needed registers.

<u>UNIT 3</u>

The material covered in the third unit will include:

- Player Graphics
- Collision Detection
- Sound
    - General Sound Algorithm
- Exercise
    - Registers Needed
- Related Reading

At the end of the unit you will be able to display and change player graphics, detect collisions between each of the graphic objects and generate rudimentary sounds.

PLAYER GRAPHICS

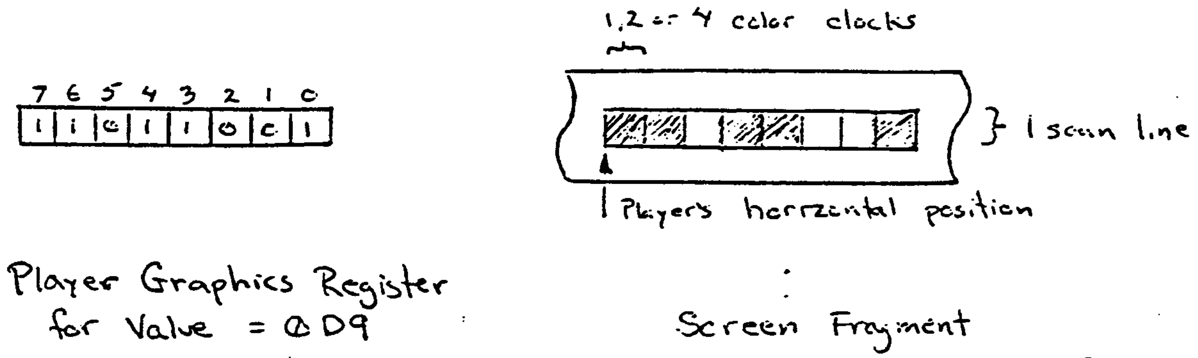


Figure 3.1. Register to Screen Correlation

The TIA controls two player sprites (moveable graphics objects), each has a player graphics register (GRP0 and GRP1). Each register is 8 bits wide and maps to each scan line at the player's position in a conventional bit-map way (See Figure 3.1). The objects are moveable over the entire screen as single units. Historically, this is an advance over the full screen bit map method where to move an object the program must erase the object, redraw the background then redraw the object at its new location. For a player the program simply changes the horizontal position and the object is moved by the TIA. Whereever the player graphics register is a 1 the pixel is the color of the player, whereever the graphics register is a 0 the pixel is the color of the object or field behind the player, usually playfield or background, for that scan line. The COLUP0 and COLUP1 registers are used to set the color of their respective players. The graphics and color registers are latched just like the rest of the TIA write registers. Therefore the graphics must be changed every scan line for 1-line resolution or every two lines for 2-line resolution.

As with the missiles and the ball the horizontal positions of the players are set in hardware using CHRST loops and the vertical positioning of the players is kernel controlled. Typically for two-line resolution players one player's graphics is updated on the even lines and the other player's graphics is updated on the odd lines. This is necessary because conventional programming techniques take 21 to 25 machine cycles to set up one player's graphics, which is one third of a scan line, leaving only two thirds of a scan line to set up two missiles, the ball and the playfield and to maintain counters, etc. Note that when the player reset registers (RESP0, RESP1) are strobed their corresponding player graphics registers are cleared automatically by the TIA.

```
        LDA     RAMTAB,Y        ;FROM A RAM TABLE
        STA     GRPO

        LDA     TABLE,Y         ;FROM A ROM TABLE
        STA     GRPO

        LDA     (POINTER),Y     ;FROM A ROM TABLE
        STA     GRPO
```

Example 3.1. Loading From Player Graphics Tables

For players the kernel must detect the beginning of the area where player graphics are displayed, and generate an index into the graphics table for each line of the player. When a player is not being displayed the graphics register must be cleared to zeros to avoid streaking the player down the screen. The graphics tables, like playfield data tables, are usually inverted in memory because the index is decremented each line. For the case where the player is animated it is necessary to have multiple graphics tables. Example 3.1 shows three ways of loading player graphics. The first method loads the graphics from a RAM table which is updated during the vertical interval. The load from RAM takes only 4 cycles but uses a great deal of RAM (1 byte for each line of graphics). The second method loads the graphics from a ROM table. The load takes 5 cycles and uses no RAM. Unfortunately this method requires seperate code for each animation state and is therefore impractical. The third method loads the graphics from a ROM table using a pointer in RAM which is updated during the vertical interval to point to the proper table of player graphics for the current animation state. The load takes 6 cycles and uses only 2 bytes of RAM.

There are two main methods for determining at which line player graphics begin and which player graphics to display at each line of the player. The kernel test method involves comparing a line counter to the players position during the kernel. The zero fill method involves filling the player graphics table with zeros for the lines where the player is not seen.

```
        LDX     #0
        LDA     VPOSPO  ;LOAD PLAYER 0 VERTICAL POSITION
        SEC             ;PREPARE FOR SUBTRACTION
        SBC     LNCNT   ;SUBTRACT LINE COUNTER FROM VPOSPO
                        ;LNCNT IS DECREMENTED EACH LINE
        CMP     #10     ;10 IS THE PLAYER HEIGHT
        BCS     SKIP    ;ABOVE PLAYER OR BELOW PLAYER
        TAY             ;USE DIFFERENCE AS AN INDEX
        LDA     TABLE,Y ;LOAD NEXT PLAYER GRAPHICS
        TAX             ;SAVE IT IN X
SKIP:   STA     WSYNC   ;WAIT TILL END OF LINE
        STX     GRPO    ;SET PLAYER GRAPHICS
```

Example 3.2. Kernel Test Code

The kernel test method can be used when ROM is at a premium and kernel time is not. It is similar to the kernel test used for the missile and ball positioning in the previous unit (See Example 3.2). Note that the accumulator value after the subtraction for the test is the index needed for retrieving the player graphics from its table during the kernel. The zero loaded at the top of the logic is stored into the graphics register in the case that the player is not to be displayed preventing the player graphics from streaking down the entire screen. This method uses 23 cycles (one third of the scan line) and only one byte of ROM for each scan line that the player is shown.

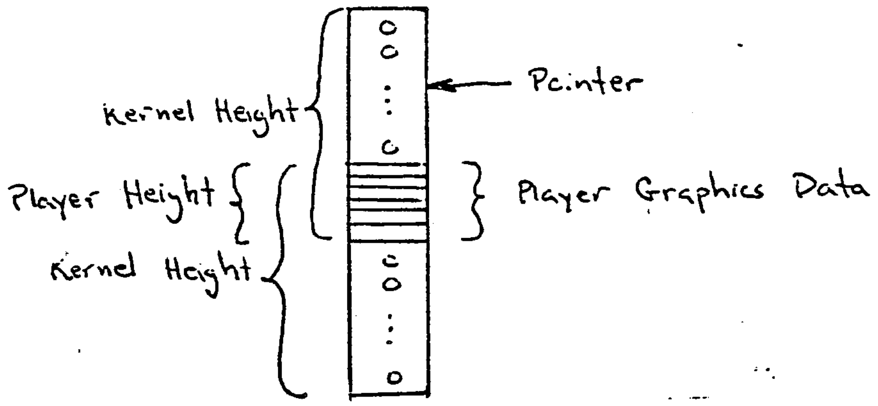


Figure 3.2. Zero Filled Data

```
LDA     (POINTER),Y     ;LOAD GRAPHICS OR ZERO
STA     WSYNC           ;WAIT TILL END OF LINE
STA     GRP1            ;SET PLAYER GRAPHICS
   .
   .                    ;OTHER KERNEL CODE
   .
DEY                     ;DECREMENT LINE COUNTER/INDEX
BPL     top of kernel
```

Example 3.3. Zero Fill Code

The zero fill method can be used when ROM is not at a premium and kernel time is. As shown in Figure 3.2 the player graphics data is surrounded by zeros on each side so that the number of bytes of zeros on one side plus the number of bytes of player graphics data equals the number of iterations in the kernel. For example, for a one-line resolution kernel 128 lines long displaying a 10 line high player the graphics table would be surrounded by 118 zeros on each side. The game logic maintains a pointer into the table to regulate the number of zeros displayed before the player graphics data is displayed and hence the vertical position of the player. The lower below the player graphics in the zeros that the pointer points the higher the player is on the screen. The line counter maintained by the kernel is used as an index into the graphics table starting at the pointer (See Example 3.3). Note that the kernel always uses the same number of bytes of the graphics table and zeros. This method can use nearly as many as 290 bytes of ROM but uses only 6 cylces of kernel time and 2 bytes of RAM. ROM space can be saved when more than 1 player graphics table is needed by overlapping the zero areas.

The registers NUSIZ0 and NUSIZ1 effect the number of copies, their spacing and size of the players. As noted in the previous unit the effect is carried over to the number of missiles as well. There are 3 horizontal resolutions of the players, 8 clocks wide, 16 clocks wide and 32 clocks wide. At 8 clocks 1 bit of data maps to one clock at 16 clocks 1 bit of data maps to two clocks and at 32 clocks 1 bit of data maps to 4 clocks which is the same as for playfield.

The NUSIZ registers can also be used to effect easy game logic actions for multiple copy player games. For example, given a multiple copy player and one copy gets deleted (shot down for example) the deletion may be effected by changing NUSIZ for the target player in all cases except where the left most copy is deleted.

The score bit in the control playfield register (bit 1 of CTRLPF) when set causes the left side of the playfield to be the color of player0 and the right side of the playfield to be the color of player 1 and when clear causes all the playfield to be the color set into COLUPF. Historically this is because the score was drawn in playfield at the top of the screen and it was assumed that the score for a player color as the player itself. Note that the ball stays the color set into COLUPF even when the playfield is set to the players' colors.

There are two registers, REFP0 and REFP1, which are used to reflect player graphics. Usually the graphics data is mapped to the screen in the order bit 0 to bit 7. When the reflect bit (bit 3) is set then the graphics data is mapped to the screen in the order bit 7 to bit 0 which displays the player reflected (See Figure 3.3). This is useful for the situation where a player reverses direction and the graphics needs to be reflected. This can also be used for cheap animation, for example the monsters in Ms Pacman are animated by this method.

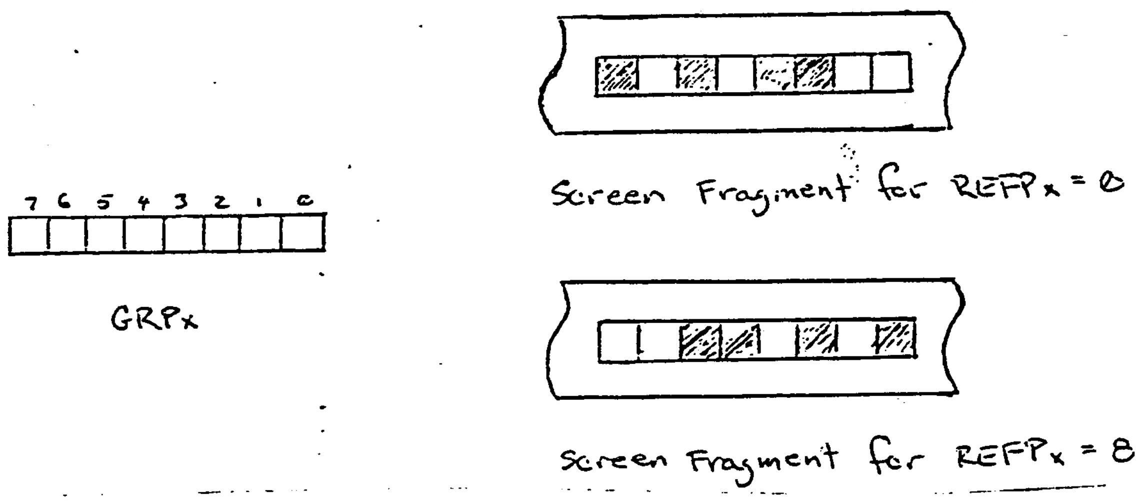


Figure 3.3. Display Using Reflect Player Register

COLLISION DETECTION

Collisions are detected by the TIA in the first frame where any two objects overlap by at least one pixel. The collision indication is latched into the appropriate collision register. For example CXM0P gives the collisions between missile 0 and both players. If player 1 collides with the missile then bit 7 is set and if player 0 collides with the missile then bit 6 is set. Only bits 6 and 7 of the collision registers are used because then the 6502 BIT instruction may be used to test both bits in one instruction. The BIT instruction sets the N flag to bit 7 of the tested byte and the C flag to bit 6 of the tested byte.

Since the collision registers are latched they must be cleared by direct instruction. Strobing the CXCLR register clears all the collision detection registers at once. Generally the collision detection registers are checked right after the kernel causing the game logic changes to be made and the clear collisions register is strobed just before the kernel for the following frame.

There are problems associated with using hardware collision detection. If the two objects have a high relative velocity, greater than the width of each object or border, then it possible that the objects will pass over each other without a collision being detected. In this case the programmer must use a software collision scheme based on the positions of the objects. Similarly if hardware collision detection is used to detect object-border collisions the object passes into the border by one pixel before the collision is detected giving the appearance of penetrating the border. In a maze type game, for example, this is not desireable and the border detection must be done in software. It is also difficult to determine which direction to "bounce" from a border because there is no hardware method to decide if the border is horizontal or vertical. Once again it is necessary to use software to decide. Yet another problem exists when multiple copies or multiple incarnations of the same object are used. Here the problem is to determine which copy or which incarnation collided with the other object. Once again this must be done with software. When oddly shaped objects are used, as in Asteroids for example, the software must be much more sophisticated to detect collisions properly.

SOUND

The TIA has two independent channels for generating sound through the TV's speaker. Each channel has three registers to control the noise generator. The AUDC0 and AUDC1 registers determine the type or content of the sounds generated. The possibilities range from pure tones to polynomial wave forms. The AUDV0 and AUDV1 register determine the volume of the sounds generated. The volume values range from 0 (sound off) to 0F (loudest). The AUDF0 and AUDF1 registers determine the frequency (pitch) of the sounds generated. The 5-bit register values are divided into the base frequency (30 Khz) to produce the primary output frequency. Therefore for a register value of 3 the primary output frequency is 30Khz/3 = 10Khz. The frequency register values range from 0 to 1F giving frequencies from 30 Khz to about 97 Hz. The actual output frequency is derived from the primary output frequency and the waveform function or dividers specified by the AUDC0 and AUDC1 registers.

General Sound Algorithm

This algorithm allows the control of which sounds will be currently sounded and which will not be sounded if there are too many sounds at any given time. This is accomplished by prioritizing the sounds by importance then checking relative priorities at the start of each new sound. It is done to prevent the "clipping" or early termination of the more important sounds in a game situation.

Tables are created for the tone, quality and volume values for each sound as they would be without this algorithm. An additional table is created with the time lengths of each sound sequence and is ordered by importance by the programmer. For example if explosions were not to be clipped by theme music then the explosion would have a higher priority. The first entry of the table, a sound that is never clipped, is then considered to be sound state 1, the second entry is considered to be state 2, etc. Each sound channel requires two bytes of RAM, one to hold the state of the channel (which sound is being generated) and the other for a count down timer (one count per frame).

When the game logic determines that a sound should start it calls the sound initiating routine passing it the sound state that should be started. The sound initiating routine allocates the sound to a channel if the channel is not being used or if the new state has a higher priority than the previous sound. If the channel is allocated then the routine initiates the sound by setting the appropriate registers.

Sound maintainance code executes every frame that decrements the countdown timers and turns off the sounds when their timers are zero.

EXERCISE

Beginning with the screen from Unit 2 add both players to the top two-thirds of the screen in the following manner (See Figure 3.4).

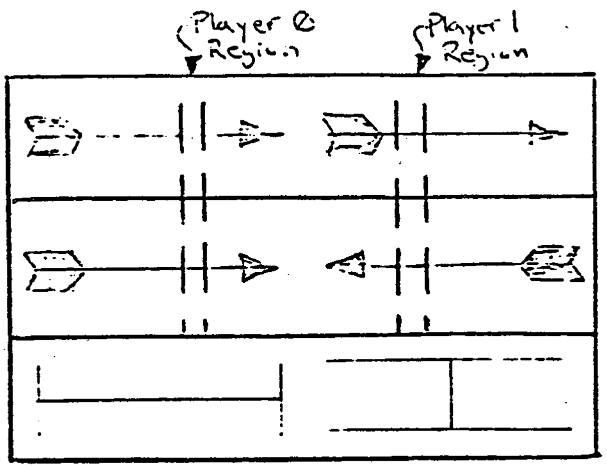


Figure 3.4. Exercise 3 Screen

Put player 0 in a fixed horizontal position approximately one third of the way across the screen from the left moving up and down from the top of the first kernel to the bottom of the second kernel. Make the player two-line resolution and use the kernel test method to vertically position the player. Set the player's NUSIZ for a single width, single copy player. Similarly, put player 1 in a fixed horizontal position approximately two thirds of the way across the screen from the left moving up and down from the top of the first kernel to the bottom of the second kernel. Make the player one-line resolution and use the zero fill method to vertically position the player. The player's NUSIZ should change between a double size single copy player and a quad wide single copy player at some rate. As with the Unit 2 exercise this exercise will have the programmer pass an object from one kernel to another. The difference between the two is that the graphics for each player must span the border of the two kernels. The players may be animated in any way the programmer desires.

Use the hardware collision detection to determine if the missiles collide with the players. If they do then reverse the missile's direction 180 degrees and generate a beep type noise

Registers Needed

RESP0, RESP1 - Resets the horizontal position of the players.
AUDC0 - Sets the channel 0 sound quality.
AUDF0 - Sets the channel 0 frequency (tone).
AUDV0 - Sets the channel 0 volume.
GRP0, GRP1 - Set the player graphics.
HMP0, HMP1 - Sets the fine positions of the players.
CXCLR - Clears the latched collision registers.
CXM0P, CXM1P - Detects the collisions between the missiles and the players.

RELATED READING

Stella Programmers Guide, Steve Wright, 12/03/79
- (6.3, 10.0, 11.0 to 11.3) Player Graphics, Collisions, and Sound
TIA Manual
- Descriptions of needed registers

UNIT 4

The material covered in the fourth unit will include:

- Scraps and Shards
    - VDEL registers
- Controllers
- PAL and SECAM conversions
- Related Reading

Scraps and shards includes the TIA registers not covered so far in the exercises.

SCRAPS AND SHARDS (MISCELLANY)

RSYNC is a register not used in game design. Instead it is used by Carl Neilson's group in the LSI testing of the TIA chip. It causes the HSYNC signals to be scrambled, shearing the picture horizontally. After a few scan-lines have passed the HSYNC signals realign and the picture returns to its normal state.

The HMCLR register clears all of the horizontal motion registers with one instruction.

There are a few registers in the original design of the VCS that are either not used as much now or are used differently than was originally intended. The VCS was designed for 4 or 5 games and it was expected that a new base unit would replace it shortly afterward. The net effect of this is that the VCS is not a generalized base unit. The VCS has had a product life 3 or 4 times as long as it was designed for.

The RESMP0 and RESMP1 are two such registers. Strobing these registers resets the missiles' horizontal position to the center of their corresponding players. The original intent was that in Combat the missile would be reset to the center of the tank and when shot would be moved with the HMOVE technique. In this way a horizontal position variable was not needed.

VDEL registers

Three other such registers are the VDEL registers. Historically, two line kernels were what was expected given the state of the hardware design. With the tests and stores for the player graphics being done every other line the player's vertical position resolution was two lines. When the player is moved vertically at a fast pace then this does not matter but if the player is moved vertically at a slow pace then this causes the motion to be jerky. For this reason the VDEL registers were created so that the graphics of an object could be delayed for one line. This gives the illusion that the player is shifted down one scan line (See Figure 4.1 for an example of VDEL's effect).

FRAME	VDEL	VERTICAL POSITION	APPEARENT VERTICAL POSITION
1	OFF	7	7
2	ON	5	6
3	OFF	5	5
4	ON	3	4
5	OFF	3	3
6	ON	1	2
7	OFF	1	1

Figure 4.1. VDEL values example

There are second parallel registers to the GRP0 and GRP1 registers called DGRP0 and DGRP1, the delayed graphics registers. When VDEL is off then the graphics data goes into the GRP registers. When the TIA displays that player then the data in the GRP register is sent to the screen. If VDEL is on then when graphics are loaded into the GRP register and when the trigger register is strobed then the data from the GRP register is shifted up into the DGRP register. See Figures 4.2, 4.3 and 4.4 for the trigger register effects.

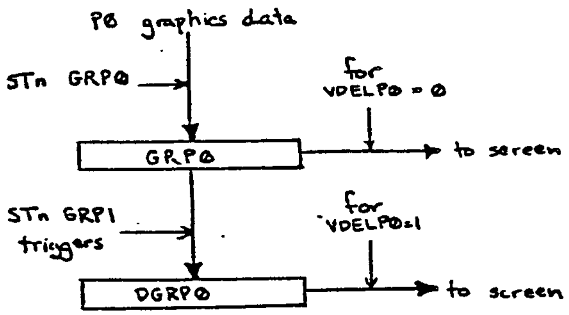


Figure 4.2. Player 0 Graphics Delay

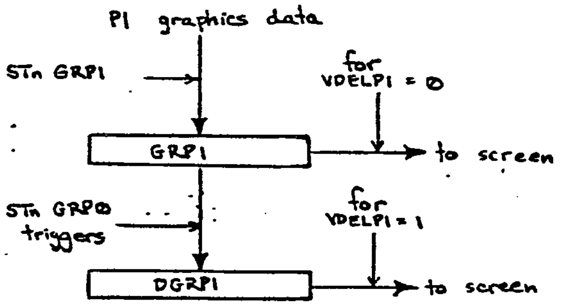


Figure 4.3. Player 1 Graphics Delay

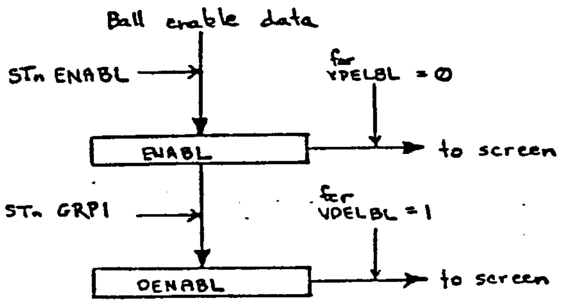


Figure 4.4. Ball Enable Delay

The use of the other graphics objects registers as the triggers is based on the original design concept of using two line kernels. In the two line kernel the other object's graphics would be set up for the next line. The use of this method saved having other strobe registers for that specific purpose.

The most profitable use of the VDEL registers is not getting one line position resolution of the graphics in a two line kernel. There are actually two other uses for the VDEL registers. The first, which uses the delay registers to provide pre-display set-up of two graphics values per player, is the six character kernel. The six character kernel is described in full detail in the Stella Package.

The use of the VDEL registers allows much greater freedom in storing graphics data, that is, it allows the programmer to store graphics data anywhere on a line and have it take effect when he or she chooses by strobing the trigger register at that time. In this way, data may be stored to player graphics registers during the time window in which the player is being displayed because the graphics are not displayed until the next line. Without VDEL, player graphics must be updated during the area of the scan line where the player is not being displayed.

CONTROLLERS

The controller input jack has 4 joystick lines, 2 paddle lines, 1 joystick trigger line, the power line and the ground line. Two of the joystick lines are used as trigger lines when the paddles are used.

Current through the paddle (a potentiometer) charges the capacitor at a varying rate (see Figure 4.5). The less the resistance of the paddle, the more quickly the capacitor charges. When the capacitor charges beyond the Schmidt trigger's upper threshold the trigger latches a logical 1. When the capacitor discharges below the trigger's lower threshold the trigger latches a logical 0. Software counts the amount of time between the logical 0 beginning and the logical 1 beginning. The time length indicates the paddle position. The range of possible values is 0 to 228 counts.

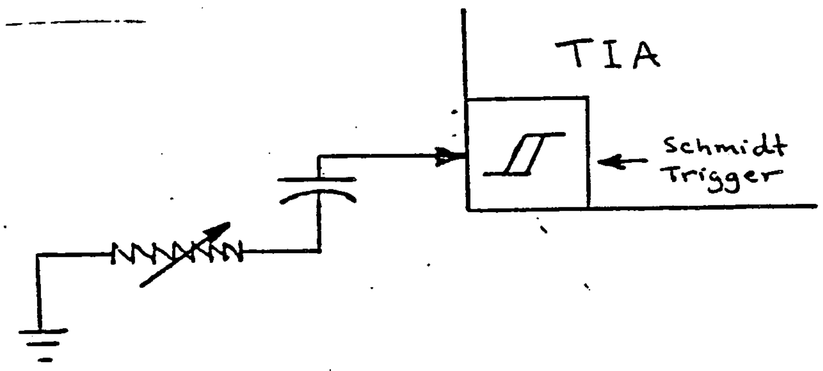


Figure 4.5. Paddle Electronics

Bit 7 of the VBLANK register is used to ground the potentiometers (pots) in the paddles. Setting the bit grounds INPT0, INPT1, INPT2 and INPT3. The program must wait at least 400 microseconds between grounding the capacitor and the first read of the input line to allow the capacitor enough time to completely discharge. When the bit is cleared then the capacitors in the paddles begin to charge. This must be done in the kernel because the counting must happen at regular intervals which would be difficult in the vertical blank time and because the maximum possible charge time (60% to 70% of the frame time) exceeds the length of the vertical blank time. Afterwards the software can determine the paddle position as a function of the length of time that the capacitor took to charge. The rate of the charging of the capacitor in the paddle is a function of the knob position, i.e. the pot position.

The paddle has a position space approximately 300 degrees wide. This gives possible counts from 1 to 228 but since the counting is done during the kernel (one count per scan line) the effective range is from 1 to 196. If the kernel is a two line kernel then the effective range is from 1 to 92. In most programs the effective range is approximately from 1 to 150 or 160 but only a fraction of the range is ever really used in the game.

Because monitoring the paddles is overhead during the kernel, the paddle is unpopular with the programmers. By way of illustration, a survey two years ago showed that 74% of cartridges use a joystick controller while only 24% use a paddle

and since then only one game has been written using a paddle controller.
The driving controller is a rotary switch with a pair of wipes producing a 2-bit grey code input on the joystick lines.

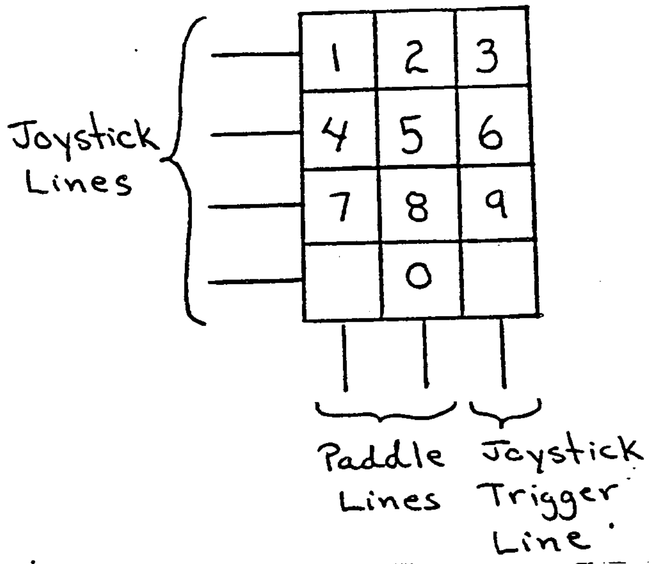


Figure 4.6. Keyboard Controller Data Lines

The keyboard uses the standard sense and scan method. The joystick lines are the "scan" lines and the paddle lines and the joystick trigger line are the "sense" lines. The one joystick line with a zero output indicates the row being tested. The "sense" line, if any, with a 0 input indicates the column of the depressed key. The shortest capacitor charge time for the paddles is about 400 microseconds. So the keyboard (see Figure 4.6) must be read 4 times during the vertical interval, spaced at 400 microsecond intervals.

The joystick has four switches set in the N, S, E, and W positions. The value read when no joystick switches are depressed is 1111_2. When a switch is closed its associated bit goes to 0. With four switches set in the directions N, S, E, and W four positions may be interpolated on the diagonals when 2 bits are 0. From the game design standpoint it is important to note that the diagonals are irregular for the operator so they should not be used in a precise maneuverving design requirement.

Bit 6 in the VBLANK register is used to enable and disable latching in the input registers INPT4 and INPT5 which are the joystick trigger buttons. In the unlatched mode reading the input register gives the program the current status at the time of the read. In the latched mode the pushed button state is saved until it is intensionally cleared by disabling latching then re-enabling latching. When latching is disabled the INPT4 and INPT5 latches are set to logic true.

PAL AND SECAM CONVERSIONS

There are two major considerations in making a PAL conversion. Then there is "fallout" from those two considerations. The first is that PAL has 312 instead of 262 scan lines per frame. The second is that PAL has 50 instead of 60 frames per second. A third, minor, consideration is that PAL colors are different then NTSC colors, PAL colors are more pastel than NTSC colors so the programmer can not get colors as rich as they are on an NTSC TV (See Figure 4.7).

NUMBER SPECIFIED	NTSC COLORS (U. S.)	NUMBER SPECIFIED	PAL COLORS (EUROPEAN EXCEPT FRANCE)
0	GREY	0	GREY
1	GOLD	1	GREY
2	ORANGE	2	GOLD
3	RED ORANGE	3	GREEN
4	PINK	4	ORANGE
5	PURPLE	5	GREEN
6	PURPLE BLUE	6	RED
7	BLUE	7	LIGHT GREEN
8	BLUE	8	PURPLE
9	LIGHT BLUE	9	TURQUOISE
10	TURQUOISE	10	PURPLE BLUE
11	GREEN BLUE	11	LIGHT BLUE
12	GREEN	12	BLUE PURPLE
13	YELLOW GREEN	13	BLUE
14	ORANGE GREEN	14	GREY
15	LIGHT ORANGE	15	GREY

Figure 4.7. NTSC to PAL color comparison

The things that are effected by the change in the number of lines per screen are vertical motion of graphics objects, the kernels, graphics data tables, and boundary or limit tests for moving objects. The things that are effected by the frame rate change are animation rates and vertical and horizontal motion of graphic objects.

The solutions to the last two problems lie in using a fractional movement technique. The following is one of many fractional movement techniques. With this method the position is kept in two bytes. The first is the integer position. The second has the integer position in the lower 3 bits, the fraction in the next 4 bits and the sign in the eighth bit. The velocity is kept in the same single mixed byte format as the second byte of the position. When the object moves the two mixed format bytes are added. Then the integer part is added to the integer position (See Example 4.1). With this method the programmer can move a graphics object from a minimum of 1 line or color clock every 15 frames to a maximum of 7 lines or color clocks every frame. There are some fractional values that produce bad results. If many frames of movement are followed by one frame of not moving or if many frames of not moving are followed by a frame of movement then the motion will be jerky. If the frames of movement and non-movement are alternated fairly evenly then the eye will smooth out the motion. When the PAL conversion is done all that needs to be done is changing the vertical and horizontal velocities by the required 17%.

```
         LDA      VVEL       ;ADD FRACTION PORTION OF VELOCITY TO
                             ; POSITION
         AND      #78        ;GET FRACTIONAL VELOCITY PORTION
         ADC      VFRAC      ;ADD IT TO FRACTION PORTION OF POSITION
         STA      VFRAC
         LDA      VVEL
         BIT      VVEL       ;IS VELOCITY NEGATIVE?
         BPL      POS        ;BRANCH IF IT IS POSITIVE.
         ORA      #78        ;OTHERWISE, SETUP AS A NEGATIVE INTEGER
                             ; FOR ADDITION
         BMI      SKIP       ;UNCONDITIONAL BRANCH AROUND
POS:     AND      #07        ;POSITIVE - CLEAR OUT FRACTION PART FOR
                             ; INTEGER ADDITION
SKIP:    ADC      VPOS       ;ADD INTEGER PORTION TO INTEGER VERTICAL
                             ; POSITION
         STA      VPOS
```

Example 4.1. Fractional Movement

Note also that the clock speed of the PAL system is slightly slower than the NTSC system so that the generated sounds will be of a slightly lower frequency. For sounds this is OK but it is possible that for music it is not.

SECAM cartridges use PAL cartridge vertical timing but the colors are totally different (See Figure 4.8).

NUMBER SPECIFIED	SECAM COLORS (FRANCE)
0	BLACK
2	BLUE
4	RED
6	MAGENTA
8	GREEN
10	CYAN
12	YELLOW
14	WHITE

Figure 4.8. SECAM Color Table

The assignment of colors to the luminosity values is based on the historical fact that games were supposed to be able to be played on black and white television sets. Therefore the different luminosities of grey were used in adjacent objects so that the objects cound be seen clearly. By assigning colors to the different luminosity values the same playability is accomplished on the SECAM color version. Note that the SECAM color version must be the same as the PAL black and white version so that the luminosities for PAL black and white must produce "good" color for the SECAM color version.

RELATED READING

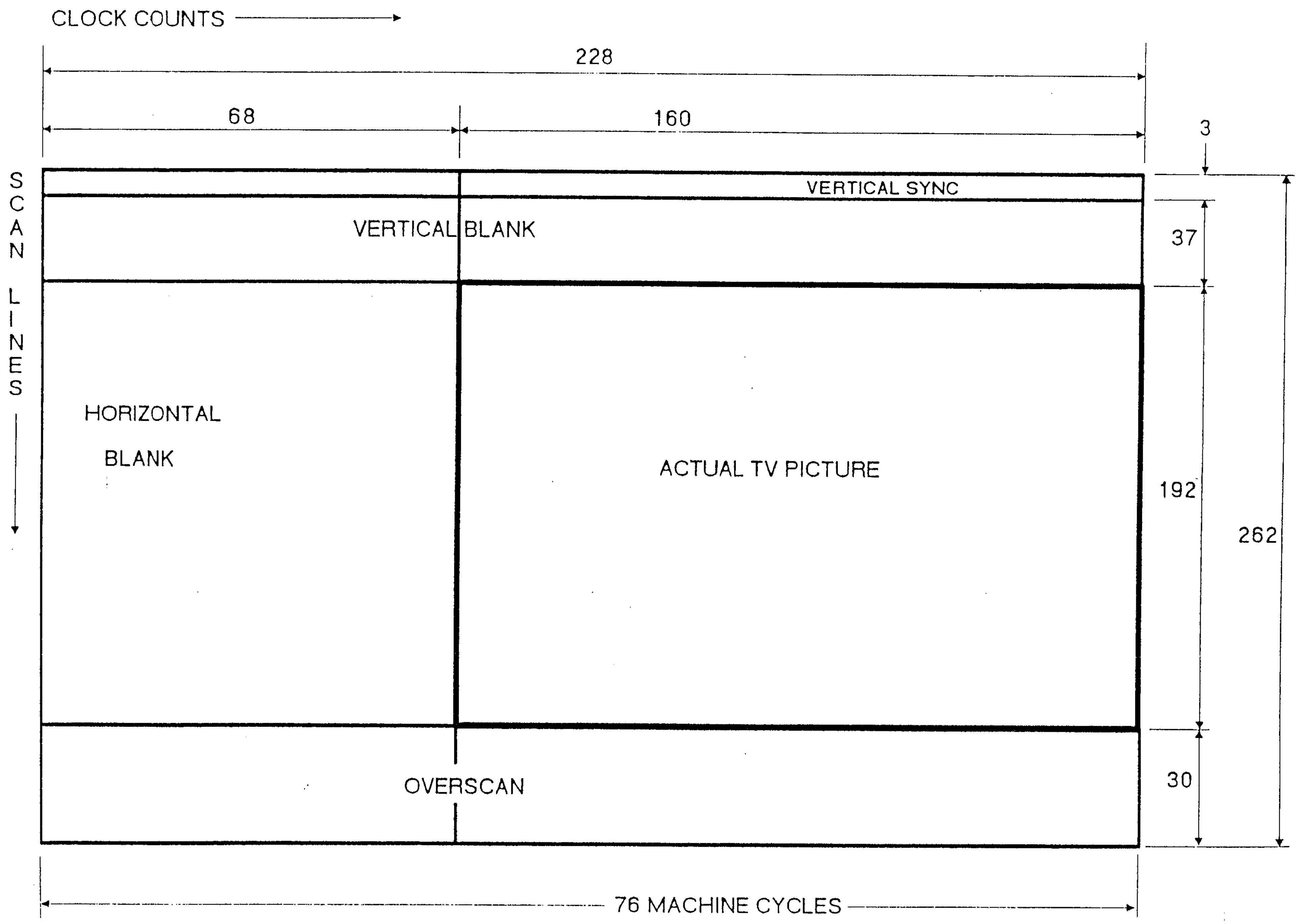
Stella Programmers Guide, Steve Wright, 12/03/79
- (Sections 12.0 to 12.2) Input Ports, Dumped Input Ports, and Latched Input Ports
The PIA, Steve Wright, 12/03/79
- (Sections 4.0 to 4.1, 5.0 to 5.5) The I/O Ports, Port B, Port A, Setting for Input or Output, Inputting and Outputting, Joystick Controllers, Paddle Controllers, and Keyboard Controllers
PAL/SECAM Conversions (Bound with the Stella Programmers Guide)
Stella Package - Stella Hollywood Conversion, Dan Hitchens, 7/14/82
Stella Package - Keyboard Reading Routine, Carla Meninsky
Stella Package - Vertical Delay, Michael Callahan, 5/14/82
Stella Package - Stella Character Maps, Carol Shaw, 12/17/79

2600 (STELLA) Programmer's Guide

By Steve Wright
12/03/79

Updated By Darryl May
7/1/88

TV FRAME



3 CLOCK COUNTS PER MACHINE CYCLE.

DIAGRAM #1

The TIA

(as seen by the programmer)

1.0 GENERAL DESCRIPTION

The TIA is a custom I.C. designed to create the TV picture and sound from the instructions sent to it by the microprocessor. It converts the 8 bit parallel data from the microprocessor into signals that are sent to the video modulation circuits which combine and shape those signals to be compatable with ordinary TV reception. A "playfield" and 5 moveable objects can be created and manipulated by software.

A playfield consisting of walls, clouds, barriers, and other seldom moved objects can be created over a colored background. The 5 moveable objects can be positioned anywhere, and consist of 2 players, 2 missles, and a ball. The playfield, players, missles, and the ball are created and manipulated by a series of registers in the TIA that the microprocessor can address and write into. Each type of object has certain defined capabilities. For example, a player can be moved with one instruction, but the playfield must be completely re-drawn in order to make it "move".

Color and luminosity (brightness) can be assigned to the background, playfield, and 5 moveable objects. Sound can also be generated and controlled for volume, pitch, and type of sound. Collisions between the various objects on the TV screen are detected by the TIA and can be read by the microprocessor. Input ports which can be read by the microprossor give the status of some of the various hand held controllers.

It also generates the signal to turn the beam off (horizontal blanking) during its return time of 68 color clocks. Total round trip for the electrom beam is 160 + 68 = 228 color clocks. Again, all the horizontal timing is taken care of by the TIA without assistance from the microprocessor.

3.2 MICROPROCESSOR SYNCHRONIZATION

The microprocessor's clock is the 3.58 MHz oscillator divided by 3, so one machine cycle is 3 color clocks. Therefore, one complete scan line of 228 color clocks allows only 76 machine cycles (228 / 3 =76) per scan line. The microprocessor must be synchronized with the TIA on a line-by-line basis, but program loops and branches take unpredictable lenghts of time. To solve this software synchronization problem, the programmer can use the WSYNC(Wait for SYNC) strobe register. Simply writting to WSYNC causes the microprocessor to halt until the electron beam reaches the right edge of the screen, then the microprocessor resumes operation at the beginning of the 68 color clocks for horizontal blanking. Since the TIA latches all instructions until altered by another write operation, it could be updated every 2 or 3 lines. The advantage is the programmer gains more time to execute software, but a price is paid with lower vertical resolution in the graphics.

NOTE: WSYNC and all the following addresses' bit structures are itemized in the TIA hardware manual. The purpose of this document is to make them understandable.

For example, if the COLUMP0 register is set for light red, both P0 and M0 will be light red when drawn.

A color-lum register is set for both color and luminosity by writing a single 7 bit instruction to that register. Four of the bits select one of the 16 available colors, and the other 3 bits select on of 8 levels of luminosity (brightness). The specific codes required to create specific color and lum are listed in the the "Detailed Address List" of the TIA hardware manual. As with all registes (except the "strobe" registers) the data written to them is latched until altered by another write operation.

5.0 PLAYFIELD

The PF register is used to create a play field of walls, clouds, barriers, etc., that are seldom moved. This low resolution register is written into to draw the left half of the TV screen only. The right half of the screen is drawn by software selection of either a duplication or a reflection (mirror image) of the left half.

The PF register is 20 bits wide, so the 20 bits are written into 3 addresses: PF0, PF1, and PF2. PF0 is only 4 bits wide and constructs the first 4 "bits" of the playfield, starting at the left edge of the TV screen. PF1 constructs the next 8 "bits", and PF2 the last 8 "bits" which end at the center of the screen. The PF register is scanned from left to right and where a "1" is found the PF color is drawn, and where a "0" is found the BK color is drawn. To clear the playfield, obviously zeroes must be written into PF0, PF1, and PF2.

bits D4 and D5 of the number-size registers (NUSIZ0, NUSIZ1). This has the effect of "stretching" the missiles out over 1, 2, 4, or 8 color clock counts (a full scan line is 160 color clocks).

6.2 BALL GRAPHICS (BL)

The ball graphics register works just like the missile registers. Writing a "1" to the enable ball register (ENABL) enables the ball graphics until the register is disabled. The ball can also be "stretched" to widths of 1, 2, 4, or 8 color clock counts by writing to bits D4 and D5 of the CTRLPF register.

The ball can also be vertically delayed one scan line. For example, if the ball graphics were enabled on scan line 95, it could be delayed to not display on the screen until scan line 96 by writing a "1" to D0 of the vertical delay (VDELBL) register. The reason for having a vertical delay capability is because most programs will update the TIA every 2 lines. This confines all vertical movements of objects to 2 scan line "jumps". The use fo vertical delay allows the objects to move one scan line at a time.

6.3 PLAYER GRAPHICS (P0, P1)

The player graphics are the most sophisticated of all the moveable objects. They have all the capabilities of the missiles and ball graphics, plus three more capabilities. Players can take on a "shape" such as a man or an airplane, and the player can be easily flipped over horizontally to display the mirror image (reflection) instead of the original image, plus multiple copies of the players can be created.

that player. Again, the specifics of all this are laid out in the TIA hardware manual.

Vertical delay for the players works exactly like the ball by writing a "1" to D0 in the players' vertical delay registers (VDELP0, VDELP1). Writing a "0" to these locations disables the vertical delay.

7.0 HORIZONTAL POSITIONING

The horizontal position of each object is set by writing to its' associated reset register (RESP0, RESP1, RESM0, RESM1, RESBL) which are all "strobe" registers (they trigger their function as soon as they are addressed). That causes the object to be positioned wherever the electron beam was in its sweep across the screen when the register was reset. For example, if the electron beam was 60 color clocks into a scan line when RESP0 was written to, player 0 would be positioned 60 color clocks "in" on the next scan line. Whether or not P0 is actually drawn on the screen is a function of the data in the GP0 register, but if it were drawn, it would show up at 60. Resets to these registers anywhere during horizontal blanking will position objects at the left edge of the screen(color clock 0). Since there are 3 color clocks per machine cycle, and it can take up to 5 machine cycles to write to a register, the programmer is confined to positioning the objects at 15 color clock intervals across the screen. This "course" positioning is "fine tuned" by the Horizontal Motion, explained in section 8.0.

before the electron beam starts drawing the next scan line. Also, for mysterious internal hardware considerations, the motion registers should not be modified for at least 24 machine cycles after an HMOVE command.

9.0 OBJECT PRIORITIES

Each object is assigned a priority so when any two objects everlap the one with the highest priority will appear to move in front of the other. To simplify hardware logic, the missles have the same priority as their associated player, and the ball has the same priority as the playfield. The background, of course, has the lowest priority. The following table illustrates the normal (default) priority assignments:

PRIORITY	OBJECTS
1	P0, M0
2	P1, M1
3	BL, PF
4	BK

This priority assignment means that players and missiles will move in front of the playfield. To make the players and missiles move <u>behind</u> the playfield, a "1" must be written to D2 of the CTRLPF register.

11.0 SOUND

There are two audio channels for sound generation. They are identical but completely independent and can be operated simultaneously to produce sound effects through the TV's speaker. Each audio channel has three registers that control a noise-tone generator (what kind of sound), a frequency selection (high or low pitch of the sound), and a volume control.

11.1 NOISE-TONE GENERATOR

The noise-tone generator is controlled by writing to the 4 bit audio control registers (AUDC0, AUDC1). The values written cause different kinds of sounds to be generated. Some are pure tones like a flute, other have various "noise" content like a rocket motor or explosion. Even though the TIA hardware manual list the sounds created by each value, some experimentation will be necessary to find "your sound".

11.2 FREQUENCY CONTROL

Frequency selection is controlled by writing to a 5 bit audio frequency register (AUDF0, AUDF1). The value written is used to divide a 30KHz reference frequency creating higher or lower pitch of whatever type of sound is created by the noise-tone generator. By combining the pure tones available from the noise-tone generator with a frequency selection, a wide range of tones can be generated.

12.2 LATCHED INPUT PORTS (INPT4, INPT5)

These two ports have latches that are both enabled by wring a "1" or disabled by writing a "0" to D6 of VBLANK. When disabled the microprocessor reads the logic level of the port directly. When enabled, the microprocessor is reading the latch, not the port. When enabled, the latch is set for logic one and remains that way until its′ port goes LOW. When the port goes LOW the latch goes LOW and remains that way regardless of what the port does. The trigger buttons of the joystick controllers connect to these ports.

2.2 READING THE TIMER

The timer may be read any number of times after it is loaded of course, but the programmer is usually interested in whether or not the timer has reached 0. The timer is read by reading INTIM at HEX address $284.

2.3 WHEN THE TIMER REACHES ZERO

The PIA decrements the value loaded into it once each interval until it reaches 0. It holds that 0 for one interval, then the value is flipped over to $FF and decrements once each clock cycle, rather than once per interval. The purpose of this feature is to allow the programmer to determine how long ago the timer zeroed itself out in the event the timer was read after it passed zero.

3.0 RAM

The PIA has 128 bytes of RAM located in the STELLA map from HEX address $80 to $FF. The microprocessor stack is normally located from $FF downward, and variables are normally located from $80 upward (hoping the two never meet).

4.0 THE I/O PORTS

The two ports (Port A and Port B) are 8 bits wide and can be set for either input or output. Port A is used to interface to various hand-held controllers but Port B is dedicated to reading the status of the STELLA console switches.

4.1 PORT B - Console Switches (Read only)

Port B is hardwired to be an input port only. Port B is read by addressing SWCHB ($282) to determine the status of all the console switches according to the following table:

5.3 JOYSTICK CONTROLLERS

Two joysticks can be read by configuring the entire port as input and reading the data at SWCHA ($280) according to the following table:

DATA BIT	DIRECTION	PLAYER
D7	Right	P0 (Left Player)
D6	Left	P0
D5	Down	P0
D4	Up	P0
D3	Right	P1 (Right Player)
D2	Left	P1
D1	Down	P1
D0	Up	P1

A "0" in a data bit indicates the joystick has been moved to close that switch. All "1"s in a player's "nibble" indicates the joystick is not moving.

NOTE: The trigger buttons do not go to the PIA. They are read on bit 7 of INPT4 and INPT5 of the <u>TIA</u>.

5.4 PADDLE (Pot) CONTROLLERS

Only the paddle triggers are read from the PIA. The paddles themselves are at INPT0 through INPT3 of the <u>TIA</u>. The data bit is set to 0 when the trigger is pressed. The paddle <u>triggers</u> can be read at SWCHA according to the following table:

DATA BIT	PADDLE NUMBER
D7	Paddle 0
D6	Paddle 1
D5	Not used.
D4	Not used.
D3	Paddle 2.
D2	Paddle 3.
D1	Not used.
D0	Not used.

TIA 1A

TELEVISION INTERFACE ADAPTOR (MODEL 1A)

GENERAL DESCRIPTION

The TIA1A is an MOS integrated circuit designed to interface between an eight (8) bit microprocessor and a television video modulator and to convert eight (8) bit parallel data into serial outputs for the color, luminosity, and composite sync required by a video modulator.

This circuit operates on a "line by line" basis, always outputing the same information every television line unless new data is written into it by the microprocessor.

A hardware sync counter produces horizontal sync timing independent of the microprocessor.

Vertical sync timing is supplied to this circuit by the microprocessor and combined into composite sync.

Horizontal position counters are used to trigger the serial output of five (5) horizontally moveable objects; two players, two missiles, and a ball. The microprocessor can add or subtract from these position counters to move these objects right or left.

The microprocessor determines all vertical position and motion by writing zeros or ones into object registers before each appropriate horizontal line.

Walls, clouds and other seldom moved objects are produced by a low resolution data register called the playfield register.

A fifteen (15) bit collision register detects all fifteen possible two object collisions between these six (6) objects (five moveable and one playfield). This collision register can be read and reset by the microprocessor. Six input ports are also provided on this chip that can be read by the microprocessor. These input ports and the collision register are the only chip addresses that can be read by the microprocessor. All other addresses are WRITE only.

Color luminosity registers are included that can be programmed by the microprocessor with eight (8) luminosity and fifteen (15) color values. A digital phase shifter is included on this chip to provide a single color output with fifteen (15) phase angles.

Two (2) independent audio generating circuits are included, each with programmable frequency, noise content, and volume control registers.

three counter on this chip whose output (1.19 MHZ) is buffered to drive an output pad called $\emptyset 0$. This pad provides the input phase zero clock to the microprocessor which then produces the system $\phi 2$ clock (1.19 MHZ).
Software program loops require different lengths of time to run depending on branch decisions made within the program. Additional synchronization (Shown in Figure 2) is, therefore, required between the software and hardware. This is done with a one bit latch called WSYNC (wait for sync). When the microprocessor finishes a routine such as loading registers for a horizontal line, or computing new vertical locations during vertical blank, it can address WSYNC, setting this latch high. When this latch is high, it drives an output pad to zero connected to the microprocessor ready line (RDY). A zero on this line causes the microprocessor to halt and wait.As shown in figure 2, WSYNC latch is automatically reset to zero by the leading edge of the next horizontal blank timing signal, releasing the RDY line, allowing the microprocessor to begin its computation and register writing for this horizontal television line or line pair.

3. Playfield graphics Register

A. Description

Objects,(such as walls, clouds, and score) which are not required to move, are written into a 20 bit register called the playfield register. This register (Figure 5) is loaded from the data bus by three separate write addresses (PF0, PF1, PF2). Playfield may be loaded at any time. To clear the playfield, zeros must be written into all three addresses.

B. Normal Serial Output

The playfield register is automatically scanned (and converted to serial output) by a bi-directional shift register clocked at a rate which spreads the twenty (20) bits out over the left half of a horizontal line. This scanning is initiated by the end of horizontal blank (left edge of television screen). Normally the same scan is then repeated, duplicating the same twenty (20) bit sequence over the right half of the horizontal line.

C. Reflected Serial Output

A relected playfield may be requested by writing crossing decode and therefore cannot trigger multiple copies of the ball graphics.

C. Player Position Counters

Each player position counter has three decodes in addition to the zero crossing decode. These decodes are controlled by bits 0,1,2 of the number size control registers (NUSIZ0, NUSIZ1), and trigger 1,2, or 3 copies of the player (at various spacings) across a horizontal line as shown on page 20. These same control bits are used for the decodes on the missile position counter, insuring an equal number of players and missiles.

D. Missile Position Counters

Missile position counters are identical to player positon counters except that they have another type of reset in addition to the previously discussed horizontal position reset. These extra reset addresses (RESMP0, RESMP1) write data bit 1 into a one bit register whose output is used to position the missile (horizontally) directly on top of its corresponding player, and to disable the missile serial output.

5. Horizontal Motion Registers

A. General Description

There are five write only registers on this chip that contain the horizontal motion values for each of the five moving objects. A typical horizontal motion register is shown in figure 4. The data bus (bits4 through7) is written into these addresses (HMP0, HMP1, HMM0, HMM1, HMBL) to load these registers with motion values. These registers supply extra (or fewer) clocks to the horizontal position counters only when commanded to do so by an HMOVE address from the microprocessor. These registers may all be cleared to zero (no motion) simultaneously by an HMCLR command from the microprocessor, or individually by loading zeros into each register.
These registers are each four bits in length and may be loaded with positive (left motion), negative (right motion) or zero (no motion) values. Negative values are represented in twos complement format.

scan counter that converts the parallel data
into serial output.
A one bit control register (REFP0, REFP1) determines the direction (reflection) of the
parallel to serial scan, outputing either
D7 through D0, or D0 through D7. This allows
reflection (horizontal flipping) of player
serial graphics data without having to flip
the microprocessor data.
The clock into the scan counter can be controlled
(three bits of NUSIZ0 and NUSIZ1) to slow
the scan rate and stretch the eight bits of
serial graphics out over widths of 8, 16,
or 32 clocks of horizontal line time. These
same control bits are used in the player-missile motion counters to control multiple
copies, so only three player widths (scan
rates) are available.

D. <u>Vertical Delay</u>,

Each of the player graphics registers actually
consists of two 8 bit parallel registers.
The first (GRP0, GRP1) is loaded (written)
from the microprocessor 8 bit data bus. The
second is automatically loaded from the output of the first. The reason for this is a
complex subject called vertical delay.
A large amount of microprocessor time is required to generate player, missile and playfield graphics (table look up, masking,
comparisons,-ect.) and load these into this
chip's registers. For most game programs this
time is just too large to fit into one horizontal
line time. In fact for most games it will
barely fit into two line times (127 microseconds).
Therefore, individual graphics registers are
loaded (written) every two lines, and used
twice for serial output between loads.
This type of programming will obviously limit
the vertical height resolution of objects
to multiples of two lines. It also will
limit the resolution of vertical motion to
two lines jumps.
Nothing can be done about the vertical
height resolution; however, vertical motion
can be resolved to a single line by addition
of a second graphics register that is automatically parallel loaded from the output
of the first, one line time after the first
was loaded from the data bus..This second graphics
register output is therefore always delayed
vertically by one line. A control bit called

they will overlap (collide) on the screen. There are six objects generated on this chip (five moving and playfield allowing fifteen possible two object collisions. These overlaps (collisions) are detected by fifteen "and" gates whenever they occur, and are stored in fifteen individual latch register bits, as shown in figure 6.

B. Reading Collisions

The microprocessor can read these fifteen collision bits on data lines 6 and 7 by addressing them two at a time. This could be done at any time but is usually done between frames (during vertical blank) after all possible collisions have serially occured.

C. Reset

All collision bits are reset simultaneously by the microprocessor using the reset address CXCLR. This is usually done near the end of vertical blank, after collisions have been tested.

8. Input ports

A. General Description

There are 6 input ports on this chip whose logic state may be read on data line 7 with read addresses INPT0 through INPT5. These 6 ports are divided into two types, "dumped" and "latched". See Figure 8.

B. Dumped Input Ports (I0 through I3)

These 4 input ports are normally used to read paddle position from an external potentiometer-capacitor circuit. In order to discharge these capacitors each of these input ports has a large transistor, which may be turned on (grounding the input ports) by writing into bit 7 of the register VBLANK. When this control bit is cleared the potentimeters begin to recharge the capacitors and the microprocessor measures the time required to detect a logic 1 at each input port.

As long as bit 7 of register VBLANK is zero, these four ports are general purpose high inpedance input ports. When this bit is a 1 these ports are grounded.

C. Latched Input ports (I4, I5)

These two input ports have latches which can be enabled or disabled by writing into bit 6 of register VBLANK.

When disabled, these latches are removed from the circuit completly and these ports become two general purpose input ports, whose present logic state can be read directly by the microprocessor.

When enabled, these latches will store negative (zero logic level) signals appearing on these two input ports, and the input port addresses will read the latches instead of the input ports.

Players will then move behind playfield (clouds, wall, etc.).
When a one is written into the score control bit, the playfield is forced to take the color-lum of player 0 in the left half of the screen and player 1 in the right half of the screen. This is used when displaying score and identifies the score with the correct player.
The priority encoder produces 4 register select lines (shown in figure 3) that are mutually exclusive. These 4 lines select either background, player 0, player 1 or playfield, and only one of them can be true at a time.

9. Color Luminance Registers

A. Description

There are four registers (shown in figure 3) that contain color-lum codes. Four bits of color code and three bits of luminance code may be written into each of these registers (COLUP0, COLUP1, COLUPF, COLUBK) by the microprocessor at any time. These codes (representing 16 color values and 8 luminance values) are given in the Detailed Address List.

B. Multiplexing

The serial graphics output from all six objects is examined by the priority encoder which activates one of the four select lines into a 4 X 7 multiplexer. This multiplexer (shown in figure 3) then selects one of the four color-lum registers as a 7 line output. Three of these lines are binary coded luminosity and go directly to chip output pads. The other four lines go to the color phase shifter.

10. Color Phase Shifter

This portion of the chip (shown in figure 3) produces a reference color output (color burst) during horizontal blank and then during the unblanked portion of the line it produces a color output shifted in phase with respect to the color burst. The The amount of phase shift determines the color and is selected by the four color code lines from the Color-lum multiplexer. Binary code 0 selects no color. Code 1 selects gold (same phase as color burst). Codes 2 (0010) through 15 (1111) shift the phase from zero through almost 360 degrees allowing selection of 15 total colors around the television color wheel.

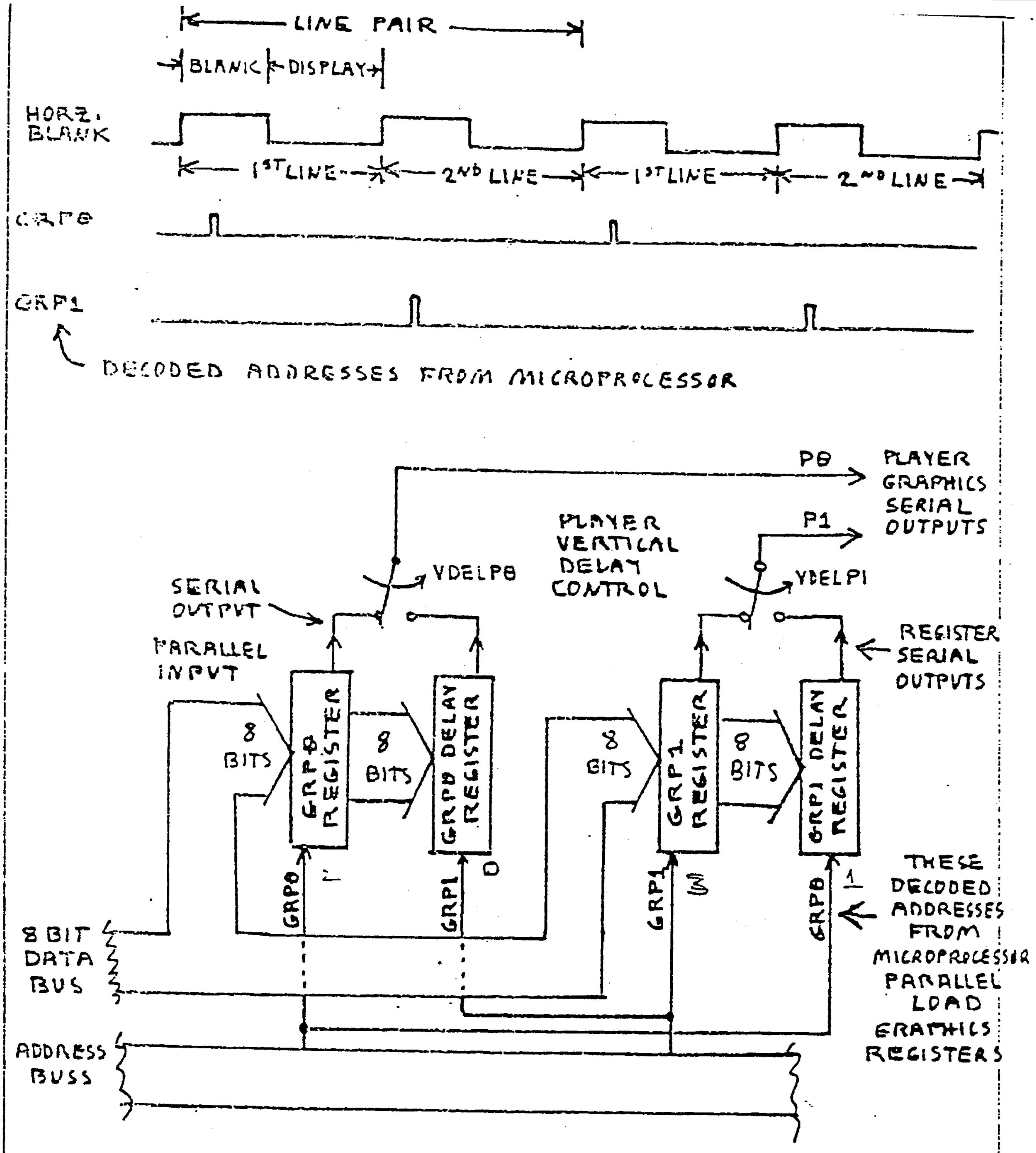


FIGURE 1. VERTICAL DELAY

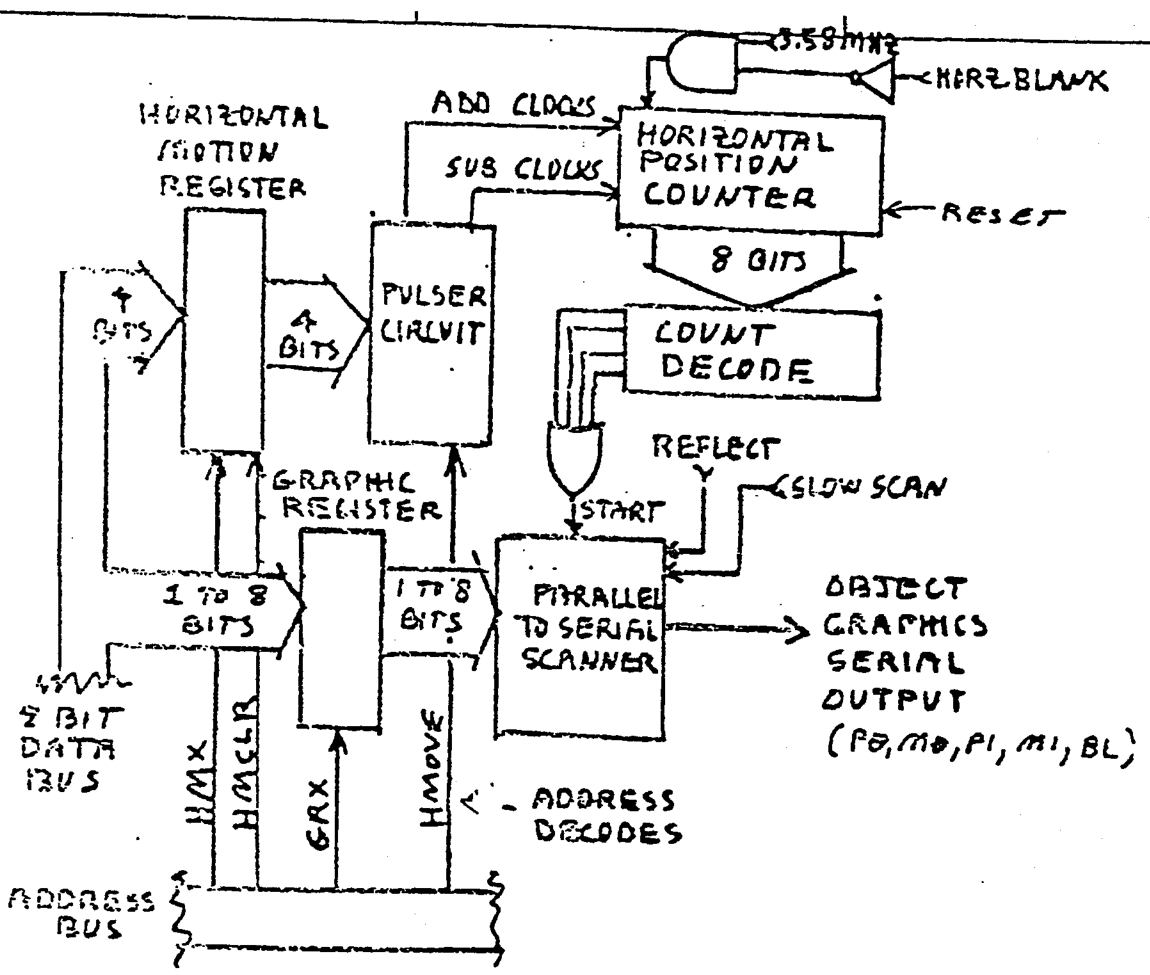


FIGURE 4. TYPICAL HORIZONTAL MOTION CIRCUIT

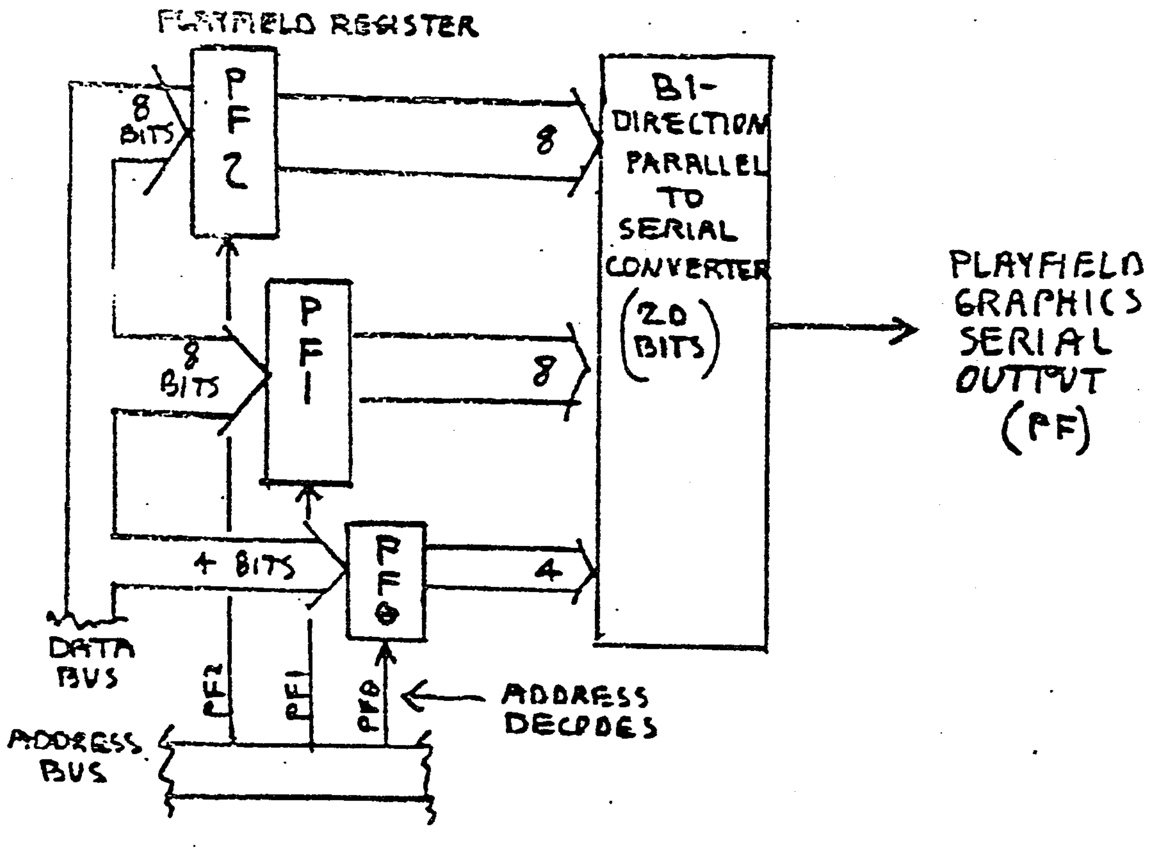


FIGURE 5. PLAYFIELD GRAPHICS

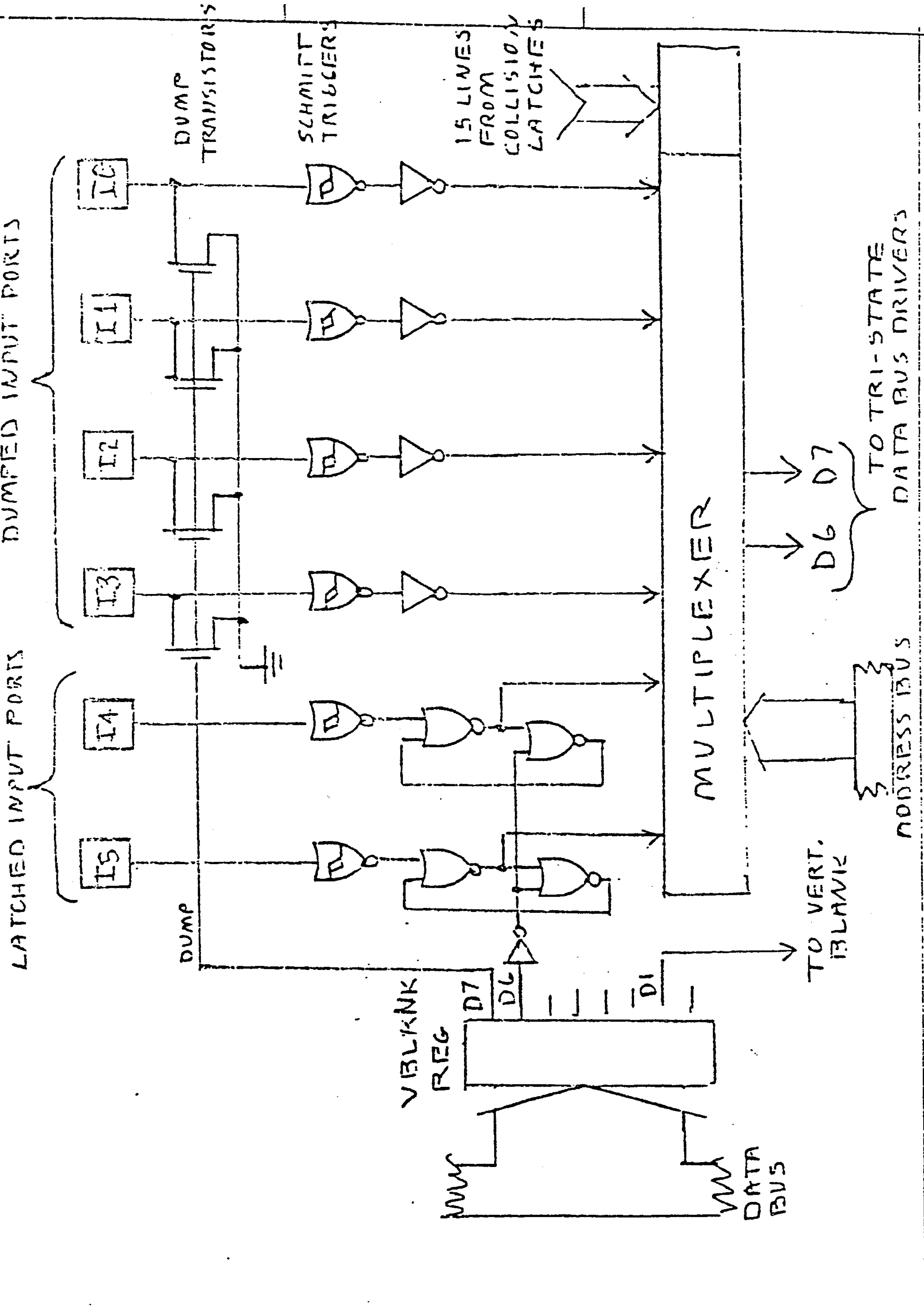
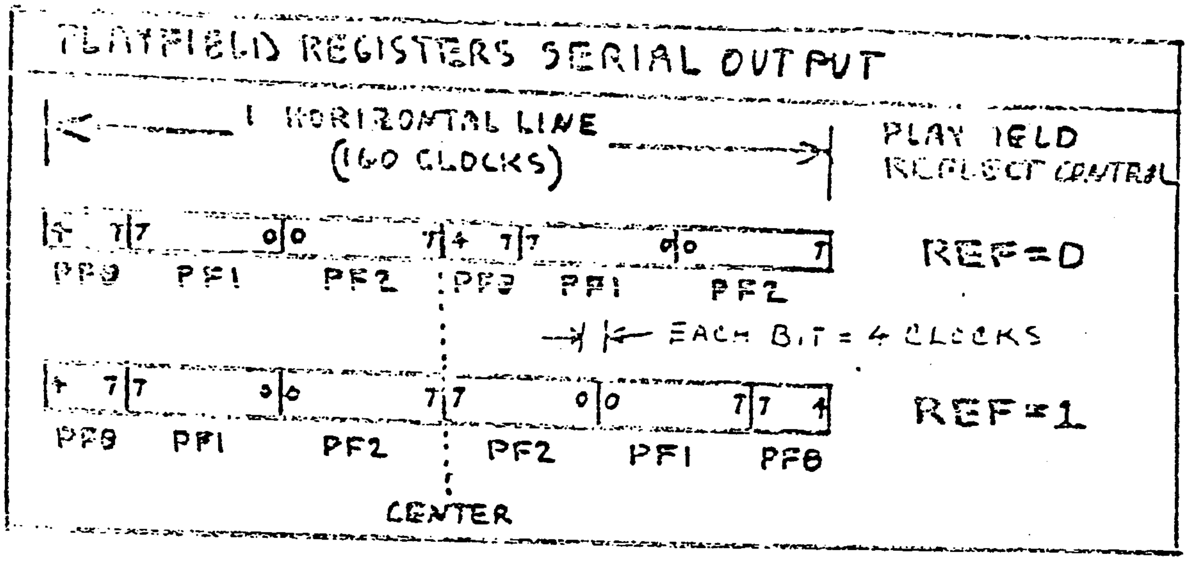


FIGURE 8. INPUT PORTS

PF0 (PF1, PF2)

THESE ADDRESSES ARE USED TO WRITE INTO THE PLAYFIELD REGISTERS

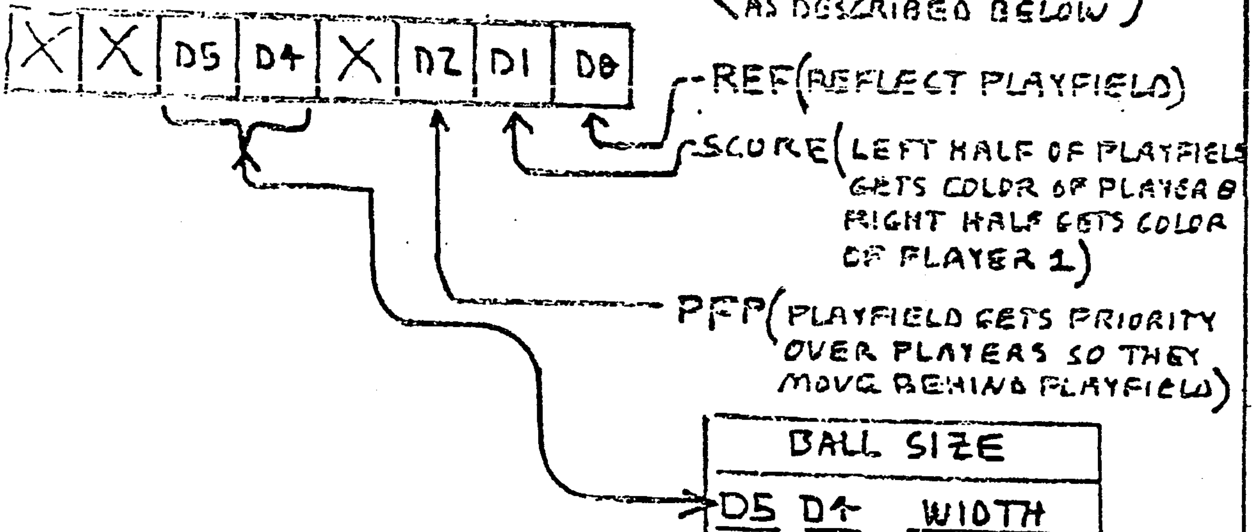
D7	D6	D5	D4	X	X	X	X	PF0
D7	D6	D5	D4	D3	D2	D1	D0	PF1
D7	D6	D5	D4	D3	D2	D1	D0	PF2



CTRLPF

THIS ADDRESS IS USED TO WRITE INTO THE PLAYFIELD CONTROL REGISTER (A LOGIC 1 CAUSES ACTION AS DESCRIBED BELOW)

X	X	D5	D4	X	D2	D1	D0



D0 — REF (REFLECT PLAYFIELD)

D1 — SCORE (LEFT HALF OF PLAYFIELD GETS COLOR OF PLAYER 0 RIGHT HALF GETS COLOR OF PLAYER 1)

D2 — PFP (PLAYFIELD GETS PRIORITY OVER PLAYERS SO THEY MOVE BEHIND PLAYFIELD)

D5, D4 —

BALL SIZE

D5	D4	WIDTH
0	0	1 CLOCK
0	1	2 CLOCKS
1	0	4 CLOCKS
1	1	8 CLOCKS

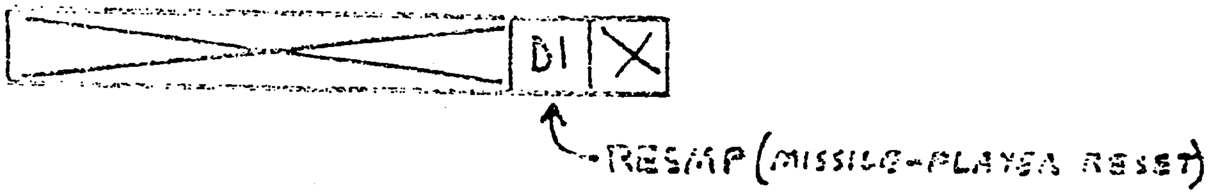
RESP0 (RESP1, RESM0, RESM1, RESBL)

THESE ADDRESSES ARE USED TO RESET PLAYERS MISSILES AND THE BALL. THE OBJECT WILL BEGIN ITS SERIAL GRAPHICS AT THAT TIME OF A HORIZONTAL LINE AT WHICH THE RESET ADDRESS OCCURS.

NO DATA BITS ARE USED

RESMP0 (RESMP1)

THESE ADDRESSES ARE USED TO RESET THE HOR. LOCATION OF A MISSLE TO THE CENTER OF IT'S CORRESPONDING PLAYER. AS LONG AS THIS CONTROL BIT IS TRUE (1) THE MISSILE WILL REMAIN LOCKED TO THE CENTER OF IT'S PLAYER AND THE MISSILE GRAPHICS WILL BE DISABLED. WHEN A ZERO IS WRITTEN INTO THIS LOCATION THE MISSILE IS ENABLED, AND CAN BE MOVED INDEPENDENTLY FROM THE PLAYER.



HMOVE

THIS ADDRESS CAUSES THE HORIZONTAL MOTION REGISTER VALUES TO BE ACTED UPON DURING THE HORIZONTAL BLANK TIME IN WHICH IT OCCURS. IT MUST OCCUR AT THE BEGINNING OF HORIZ. BLANK IN ORDER TO ALLOW TIME FOR GENERATION OF EXTRA CLOCK PULSES INTO THE HORIZONTAL POSITION COUNTERS. IF MOTION IS DESIRED THIS COMMAND MUST IMMEDIATELY FOLLOW A WSYNC COMMAND IN THE PROGRAM.

NO DATA BITS ARE USED

HMCLR

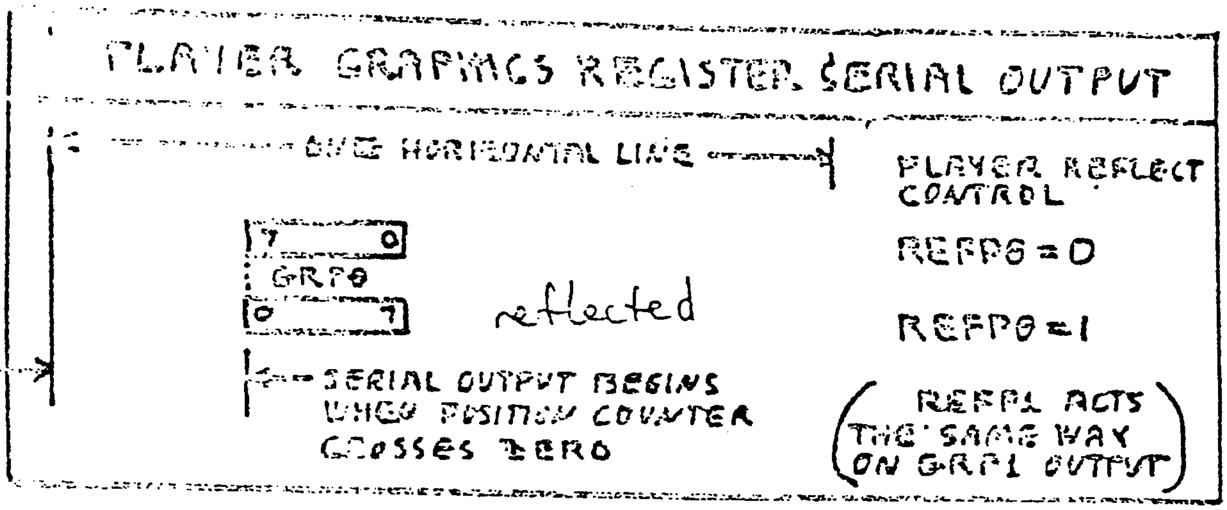
THIS ADDRESS CLEARS ALL HORIZONTAL MOTION REGISTERS TO ZERO (NO MOTION)

NO DATA BITS ARE USED

GRP0(GRP1)

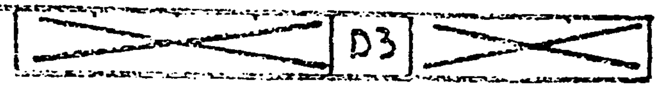
THESE ADDRESSES WRITE DATA INTO THE PLAYER GRAPHICS REGISTERS

D7	D6	D5	D4	D3	D2	D1	D0


PLAYER GRAPHICS REGISTER SERIAL OUTPUT
ONE HORIZONTAL LINE
7 0
GRP0
0 7
reflected
SERIAL OUTPUT BEGINS WHEN POSITION COUNTER CROSSES ZERO
PLAYER REFLECT CONTROL
REFP0=0
REFP0=1
REFP1 ACTS THE SAME WAY ON GRP1 OUTPUT

REFP0(REFP1)

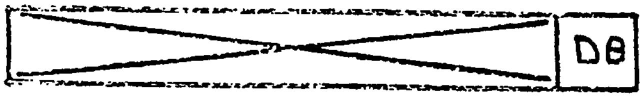
THESE ADDRESSES WRITE D3 INTO THE 1 BIT PLAYER REFLECT REGISTERS.



0 NO REFLECT (D7 OF GRP ON LEFT)
1 REFLECT (D0 OF GRP ON LEFT)

VDELP0(VDELP1, VDELBL)

THESE ADDRESSES WRITE D0 INTO THE 1 BIT VERTICAL DELAY REGISTERS, TO DELAY PLAYERS OR BALL BY ONE VERTICAL LINE.



0 NO DELAY
1 DELAY

CXCLR

THIS ADDRESS CLEARS ALL COLLISION LATCHES TO ZERO (NO COLLISION)

DATA BITS NOT USED

AUDC0 (AUDC1)

THESE ADDRESSES WRITE DATA INTO THE AUDIO CONTROL REGISTERS WHICH CONTROL THE NOISE CONTENT AND ADDITIONAL DIVISION OF THE AUDIO OUTPUT.

HEX CODE	D3	D2	D1	D0	TYPE OF NOISE OR DIVISION	
0	0	0	0	0	SET TO 1	
1	0	0	0	1	4 BIT POLY	
2	0	0	1	0	÷15 → 4-BIT POLY	
3	0	0	1	1	5 BIT POLY → 4 BIT POLY	
4	0	1	0	0	÷2	
5	0	1	0	1	÷2	pure tone
6	0	1	1	0	÷31	
7	(	1	1	1	5 BIT POLY → ÷2	
8	1	0	0	0	9 BIT POLY (WHITE NOISE)	
9	1	0	0	1	5 BIT POLY	
A	1	0	1	0	÷31	pure tone
B	1	0	1	1	SET LAST 4 BITS TO 1	
C	1	1	0	0	÷6	
D	1	1	0	1	÷6	pure tone
E	1	1	1	0	÷93	
F	1	1	1	1	5 BIT POLY ÷6	

AUDV0 (AUDV1)

THESE ADDRESSES WRITE DATA INTO THE AUDIO VOLUME REGISTERS WHICH SET THE PULL DOWN IMPEDANCE DRIVING THE AUDIO OUTPUT PADS.

D3	D2	D1	D0	AUDIO OUTPUT PULLDOWN CURRENT
0	0	0	0	NO OUTPUT CURRENT
0	0	0	1	LOWEST
0	0	1	0	
–	–	–	–	
1	1	1	0	
1	1	1	1	HIGHEST

WRITE ADDRESS SUMMARY

PG 2 OF 2

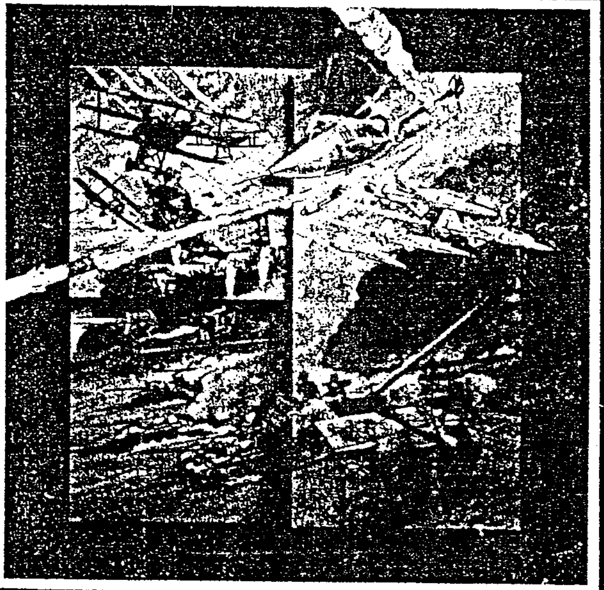
6 BIT ADDRESS		ADDRESS NAME	DATA BITS USED								FUNCTION
OCT	HEX		7	6	5	4	3	2	1	0	
27	17	AUDF0				1	1	1	1	1	AUDIO FREQUENCY 0
30	18	AUDF1				1	1	1	1	1	AUDIO FREQUENCY 1
31	19	AUDV0					1	1	1	1	AUDIO VOLUME 0
32	1A	AUDV1					1	1	1	1	AUDIO VOLUME 1
33	1B	GRP0	1	1	1	1	1	1	1	1	GRAPHICS PLAYER 0
34	1C	GRP1	1	1	1	1	1	1	1	1	GRAPHICS PLAYER 1
35	1D	ENAM0							1		GRAPHICS (ENABLE) MISSILE 0
36	1E	ENAM1							1		GRAPHICS (ENABLE) MISSILE 1
37	1F	ENABL							1		GRAPHICS (ENABLE) BALL
40	20	HMP0	1	1	1	1					HORIZONTAL MOTION PLAYER 0
41	21	HMP1	1	1	1	1					HORIZONTAL MOTION PLAYER 1
42	22	HMM0	1	1	1	1					HORIZONTAL MOTION MISSILE 0
43	23	HMM1	1	1	1	1					HORIZONTAL MOTION MISSILE 1
44	24	HMBL	1	1	1	1					HORIZONTAL MOTION BALL
45	25	VDELP0								1	VERTICAL DELAY PLAYER 0
46	26	VDELP1								1	VERTICAL DELAY PLAYER 1
47	27	VDELBL								1	VERTICAL DELAY BALL
50	28	RESMP0							1		RESET MISSILE 0 TO PLAYER 0
51	29	RESMP1							1		RESET MISSILE 1 TO PLAYER 1
52	2A	HMOVE	STROBE								APPLY HORIZONTAL MOTION
53	2B	HMCLR	STROBE								CLEAR HORIZ. MOTION REGS.
54	2C	CXCLR	STROBE								CLEAR COLLISION LATCHES

01 combat

	Game No.	Straight Missile	Guided Missile	Machine Guns	Direct Hit	Billiard Hit		Open Field	Easy Maze	Complex Maze	Clouds
TANK	1		▨					▨			
	2		▨						▨		
	3	▨							▨		
	4		▨							▨	
	5	▨								▨	
TANK-PONG	6				▨	▨			▨		
	7				▨	▨				▨	
	8							▨			
	9								▨		
INVISIBLE TANK	10		▨					▨			
	11		▨								
INVISIBLE TANK-PONG	12				▨	▨			▨		
	13							▨			
	14								▨		
BI-PLANE	15		▨								▨
	16	▨									▨
	17			▨							▨
	18			▨				▨			
2 vs. 2	19		▨					▨			
1 vs. 3	20	▨						▨			
JET	21		▨								▨
	22	▨									▨
	23		▨					▨			
	24	▨						▨			
2 vs. 2	25		▨								▨
1 vs. 3	26		▨					▨			
2 vs. 2	27	▨						▨			

LITHO IN U.S.A.

COMBAT
GAME PROGRAM™ INSTRUCTIONS



ATARI®

A Warner Communications Company

ATARI, INC., Consumer Division
1195 Borregas Ave., Sunnyvale, CA 94086

C011402-01

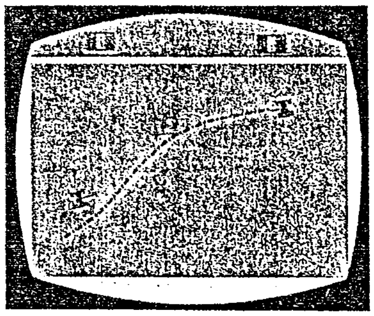


Fig. G —Tank Open Playfield

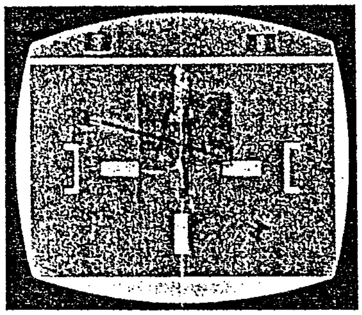


Fig. H —Tank Easy Maze Playfield

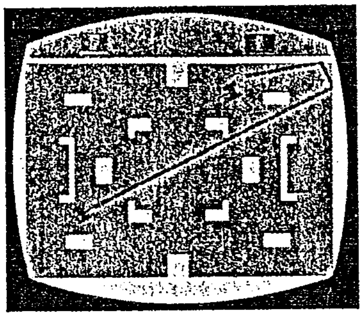


Fig. I —Tank Complex Maze Playfield

TANK GAMES

The object of TANK is to hit your opponent as many times as you can before the game ends. You score one point for each hit.

Game No.		
1.	Open Field (Fig. G)	Guided Missile (Fig. E)
2.	Easy Maze (Fig. H)	Guided Missile (Fig. E)
3.	Easy Maze (Fig. H)	Straight Missile (Fig. D)
4.	Complex Maze (Fig. I)	Guided Missile (Fig. E)
5.	Complex Maze (Fig. I)	Straight Missile (Fig. D)

TANK-PONG GAMES

TANK-PONG is a unique series of games from Atari. The missile will bounce off the walls and barriers of the playfield. In the "Direct Hit" games, you score by hitting your opponent either head on, or by bouncing your missile. In "Billiard Hit," your missile must bounce at least once before hitting your opponent to score. If, after firing, your own missile hits your tank, it will not score against you.

Game No.		
6.	Easy Maze (Fig. H)	Direct Hit or Billiard (Fig. F)
7.	Complex Maze (Fig. I)	Direct Hit or Billiard (Fig. F)
8.	Open Field (Fig. G)	Billiard Hit (Fig. F)
9.	Easy Maze (Fig. H)	Billiard Hit (Fig. F)

INVISIBLE TANK GAMES

You and your opponent are invisible to each other, except when a missile is fired or when a hit is made. In addition, the tanks become visible whenever they bump into a wall or barrier.

Game No.		
10.	Open Field (Fig. G)	Guided Missiles (Fig. E)
11.	Easy Maze (Fig. H)	Guided Missiles (Fig. E)

INVISIBLE TANK-PONG GAMES

These games combine the invisible play feature with the missile action of TANK-PONG.

Game No.		
12.	Easy Maze (Fig. H)	Direct or Billiard (Fig. F)
13.	Open Field (Fig. G)	Billiard Hit (Fig. F)
14.	Easy Maze (Fig. H)	Billiard Hit (Fig. F)

SPACE WAR

	SPACE WAR							SPACE SHUTTLE									
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
No. of Players	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Single Module									▨			▨		▨	▨	▨	▨
Galaxy Boundary	▨	▨		▨		▨					▨	▨	▨			▨	▨
Warp Drive			▨		▨		▨	▨	▨	▨				▨	▨		
Space Sun				▨	▨					▨	▨				▨		▨
Hyperspace		▨	▨	▨	▨	▨	▨										
Starbase						▨	▨										

SPACE WAR
GAME PROGRAM™ INSTRUCTIONS

Model CX2604

ATARI

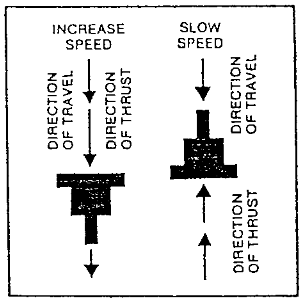
A Warner Communications Company

ATARI, INC., Consumer Division
1195 Borregas Ave., Sunnyvale, CA 94086

© 1978 ATARI, INC. C011402-04

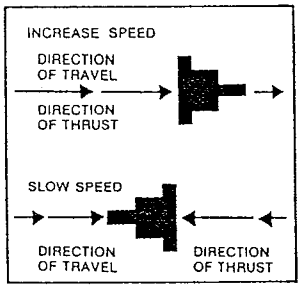
EXERCISE 2

Push the Game Reset button. By pushing the Joystick forward (towards the television screen), give your Star Ship three quick short bursts of "thrust". Notice that your Star Ship is now travelling in a forward motion towards the bottom of the playfield. By turning the Star Ship either clockwise or counter clockwise, turn the Star Ship so it is facing **away** from the forward motion. Give the Star Ship three short quick bursts of thrust. Your Star Ship will slow almost to a stop. Push the Game Reset and try again. Practice this exercise until you can stop the Star Ship completely.



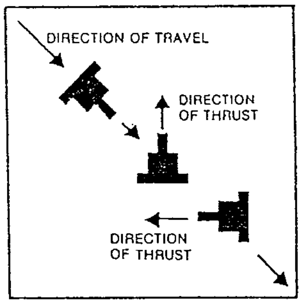
EXERCISE 3

Push the Game Reset button. Turn your Star Ship so it is facing to your right. Give your Star Ship continuous "thrust" by pushing the Joystick forward and holding it in position. When the Star Ship is travelling rapidly across the playfield, turn your Star Ship in the opposite direction of travel and push the Joystick forward, giving your Star Ship reverse "thrust". Your Star Ship will slow. Practice this exercise until you can bring your Star Ship to a complete halt.



EXERCISE 4

Push the Game Reset button. Turn your Star Ship so it is facing to your right and down (approximately 45°). Give your Star Ship continuous "thrust" until it is moving rapidly across the playfield. Alternate using horizontal "thrust" and vertical "thrust" to bring the Star Ship to a near standstill in the middle of the playfield. After mastering the above exercises, you should be an experienced Star Ship captain, ready to do battle among the stars.



HANDICAP Difficulty Switches

The left and right Difficulty switches must be in the "B" position during all Space War games. In Space Shuttle games, slide the Difficulty switch "A" and you must exactly match your Star Ship's velocity to the Space Module's velocity. In "B" position, your Star Ship does not have to travel at the same speed to dock with the Space Module.

SCORING

During Space War games (1 through 7) you score one point when your opponent's Star Ship explodes. A Star Ship will explode when:

- A direct hit is made by firing a missile.
- The Star Ship collides with the Space Sun (games 4 and 5).
- The Star Ship runs out of fuel while in Hyperspace (games 2 through 7).
- The Star Ship tries to enter Hyperspace when out of fuel (games 2 through 7).

In one and two-player Shuttle games (8 through 17) one point is scored each time the Star Ship is successfully docked with the Space Module. You have ten minutes to score a maximum ten points.

GAME 2
Engage in combat in a galaxy which features Galaxy Boundaries and Hyperspace.

GAME 3
Oppose your space opponent in a galaxy which has Warp Drive. Use Hyperspace as a defensive move.

GAME 4
The Space Sun in the center of this galaxy exerts gravity during combat. Avoid your opponent or collision with the Space Sun by using Hyperspace. You also fight within Galaxy Boundaries.

GAME 5
The Space Sun, Warp Drive, and Hyperspace are the features of the galaxy playfield.

GAME 6
You can refuel and receive more missiles at any time during this game. Steer your Star Ship to the Starbase. This galaxy also features Galaxy Boundaries and Hyperspace.

GAME 7
Steer your Star Ship to the Starbase at any time during the game to refuel or receive more missiles. This galaxy also features Warp Drive and Hyperspace.

SPACE SHUTTLE GAMES

If you have mastered the exercises, you are ready to try Space Shuttle. Connect your Star Ship with the Space Module to score. Recommended strategy is to first match your Star Ship's speed to the Space Module's speed. Then slowly maneuver your Star Ship towards the Space Module. During Shuttle games the Star Ships have an unlimited supply of fuel.

In one-player games, you control one Star Ship with the left Joystick controller and compete against the clock. You have ten minutes to score a maximum ten points. During two-player games each player maneuvers his Star Ship to score. In two-player games with two Space Modules, the target Space Module will be the same color as your Star Ship. First player to score ten points or the most points in ten minutes wins.

2-PLAYER GAMES

GAME 8
Two players each control one Star Ship and attempt to connect with the Module which is color coordinated with the Ship. Warp Drive is present in this galaxy.

GAME 9
Two players each control one Star Ship and compete to connect with the same Space Module. This galaxy features Warp Drive.

GAME 10
Each player controls a Star Ship and attempts to connect with the color coordinated Space Module. A Space Sun and Warp Drive add extra dimension to the strategy you will use.

GAME 11
Each player controls a Star Ship and attempts to connect with a color coordinated Space Module. This galaxy has a Space Sun and Galaxy Boundaries.

GAME 12
Galaxy Boundaries characterize this galaxy. Each player controls a Star Ship and attempts to connect to the same Space Module.

GAME 13
Each player controls a Star Ship and attempts to connect with the Space Module that is color coordinated to the Star Ship. Galaxy Boundaries are featured.

SLOT RACERS™ GAME PROGRAM™ INSTRUCTIONS

ATARI

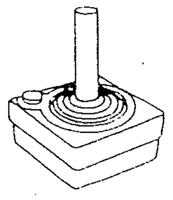
A Warner Communications Company

ATARI, INC., Consumer Division
1195 Borregas Ave., Sunnyvale, CA 94086

C011402-06

SLOT RACERS™

Use your Joystick Controllers with this Game Program™. Be sure the controllers are firmly connected to your Video Computer System™. See the Owner's Manual for details. Hold the controllers with the red button to your upper left towards the television screen.



NOTE: To prolong the life of your Atari Video Computer System and protect the electronic components, the console unit should be OFF when inserting a Game Program.

HOW TO PLAY

Screech! Pow! Smash! This is the super chase scene, and you're in it----right behind the wheel of a Super Chasemobile car equipped with power and incredible gadgets.

Console Controls:

To start the game action, use the console controls to program the type of game you want to play.

- Press the Game Select to choose the game. The numbers of each game appear in the upper left corner. (See Game Descriptions of the game number of each game).
- Press the Game Select to choose one of the four chase mazes.

There are four chase mazes. Each player steers one car through the maze. Chase your opponent and attempt to hit him with one of the secret missiles fired from your car headlights. You score a point each time you hit your opponent with a missile.

- The missile on the screen must hit your opponent's car

OR

- You must retrieve the missile on the screen by steering your car into it.

GAME DESCRIPTIONS

GAMES 1-4

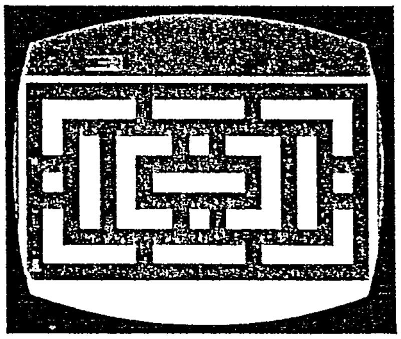
Select your favorite maze pattern. These games feature missiles that travel faster than the cars. Note that the speed of both the missiles and cars increase with each game number. For example, Game 1 has the slowest moving missiles and cars; Game 4 features the fastest moving missiles and cars.

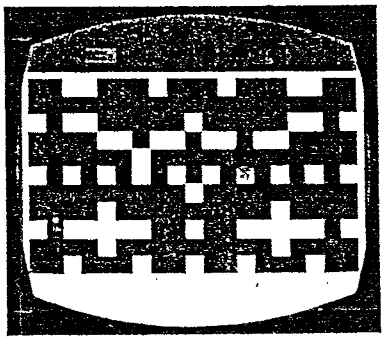
GAMES 5-7

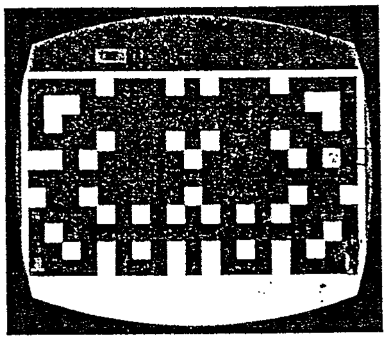
Drive your car fast on these mazes. This time, the cars travel faster than the missiles during these games. Note that the speed of the cars increase with each game number. For example, Game 5 features the slowest moving cars; Game 7 offers plenty of speed.

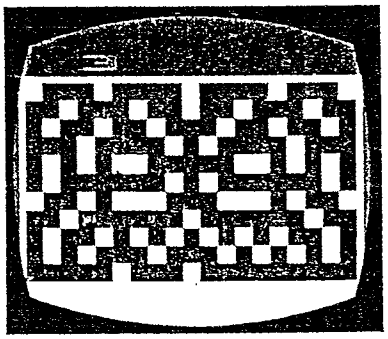
GAMES 8 AND 9

Missiles do not automatically turn corners during these games. That's why some of your missiles may become trapped in front of a wall. In Game 9 you're driving race cars; Game 8 features slower cars.

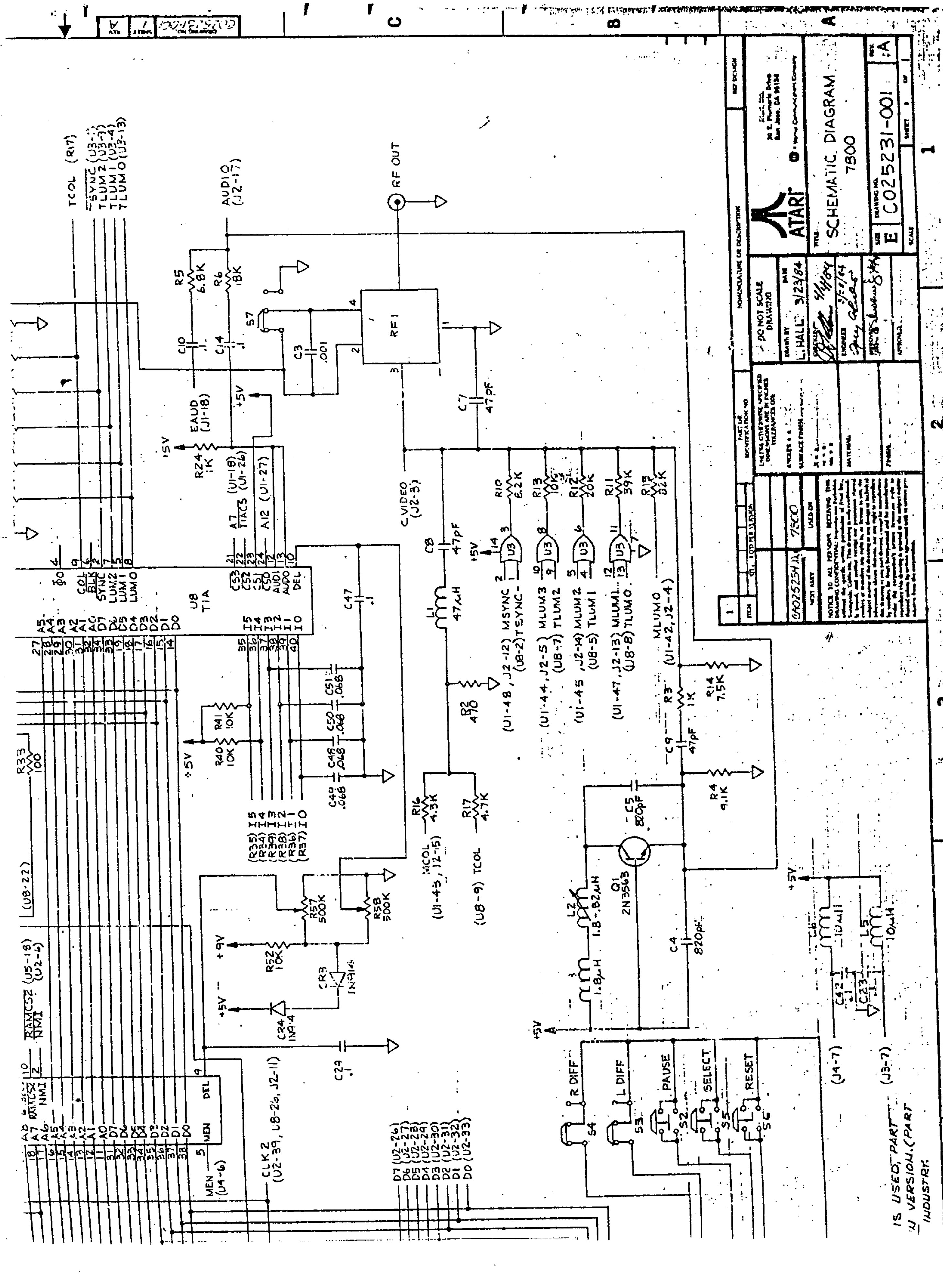

Maze 1


Maze 2


Maze 3


Maze 4

Printed in U.S.A.



DO NOT SCALE DRAWING

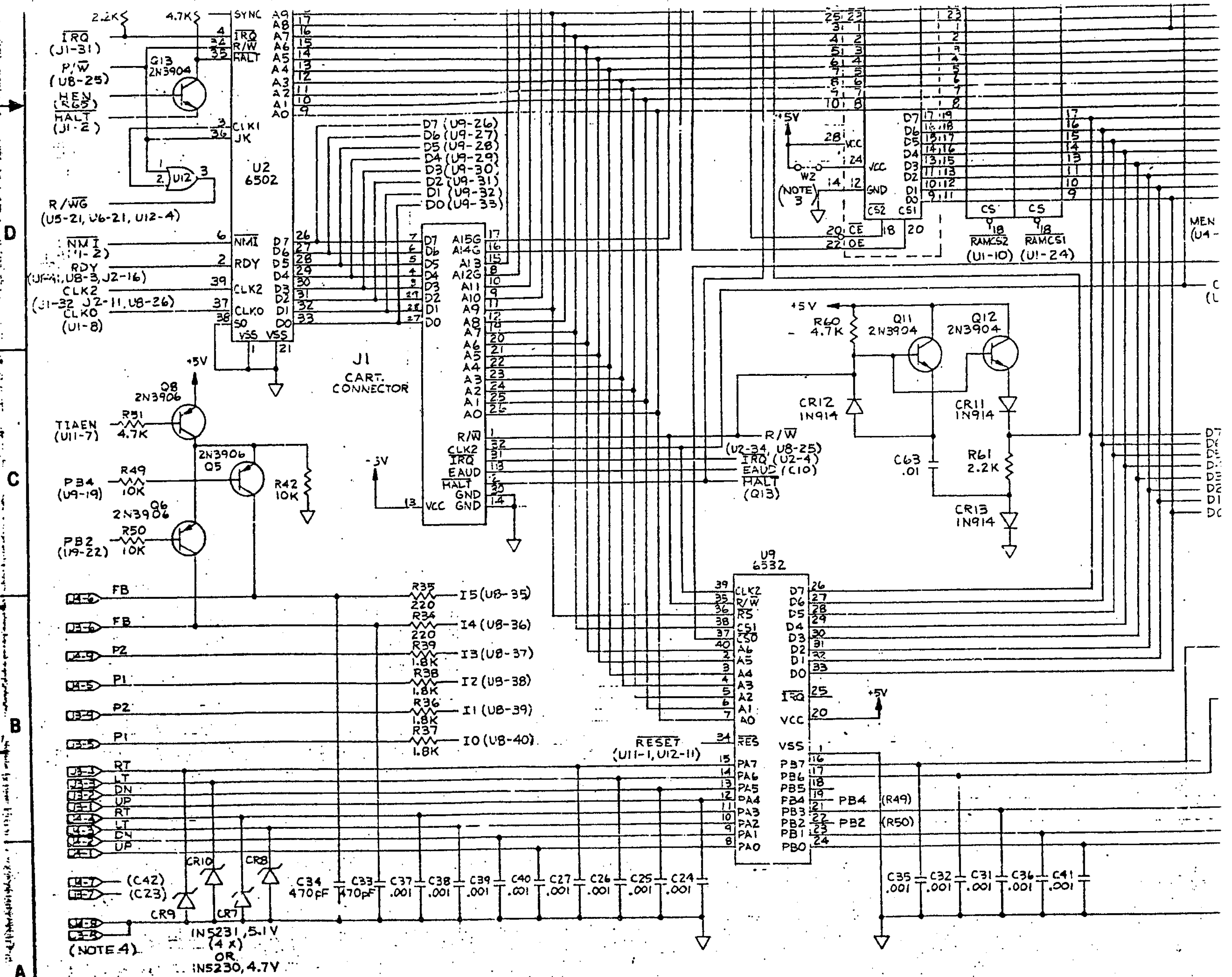
DRAWN BY L. HALL 3/23/84

ATARI

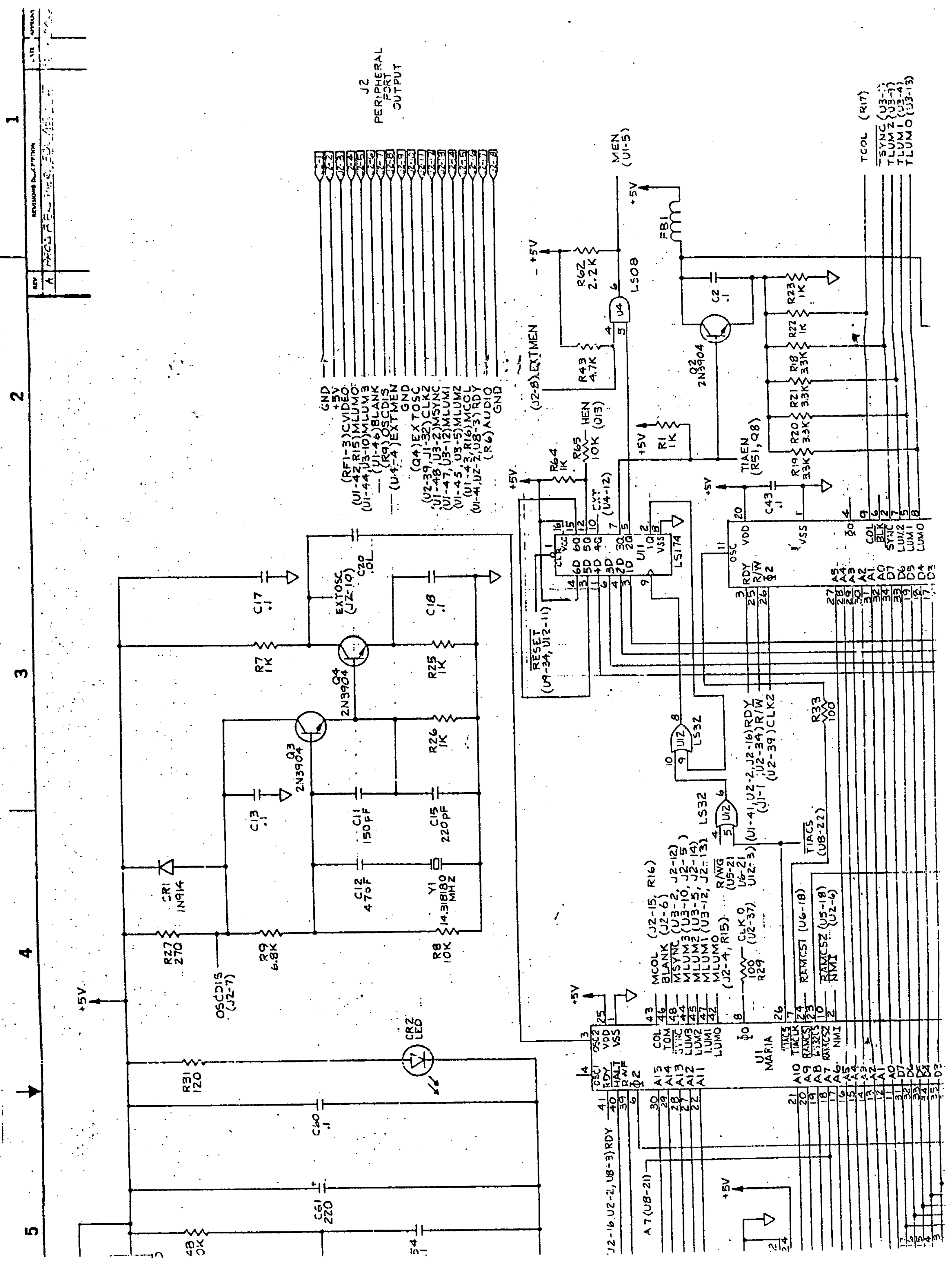
SCHEMATIC DIAGRAM 7800

SIZE E DRAWING NO. C025231-001 REV A

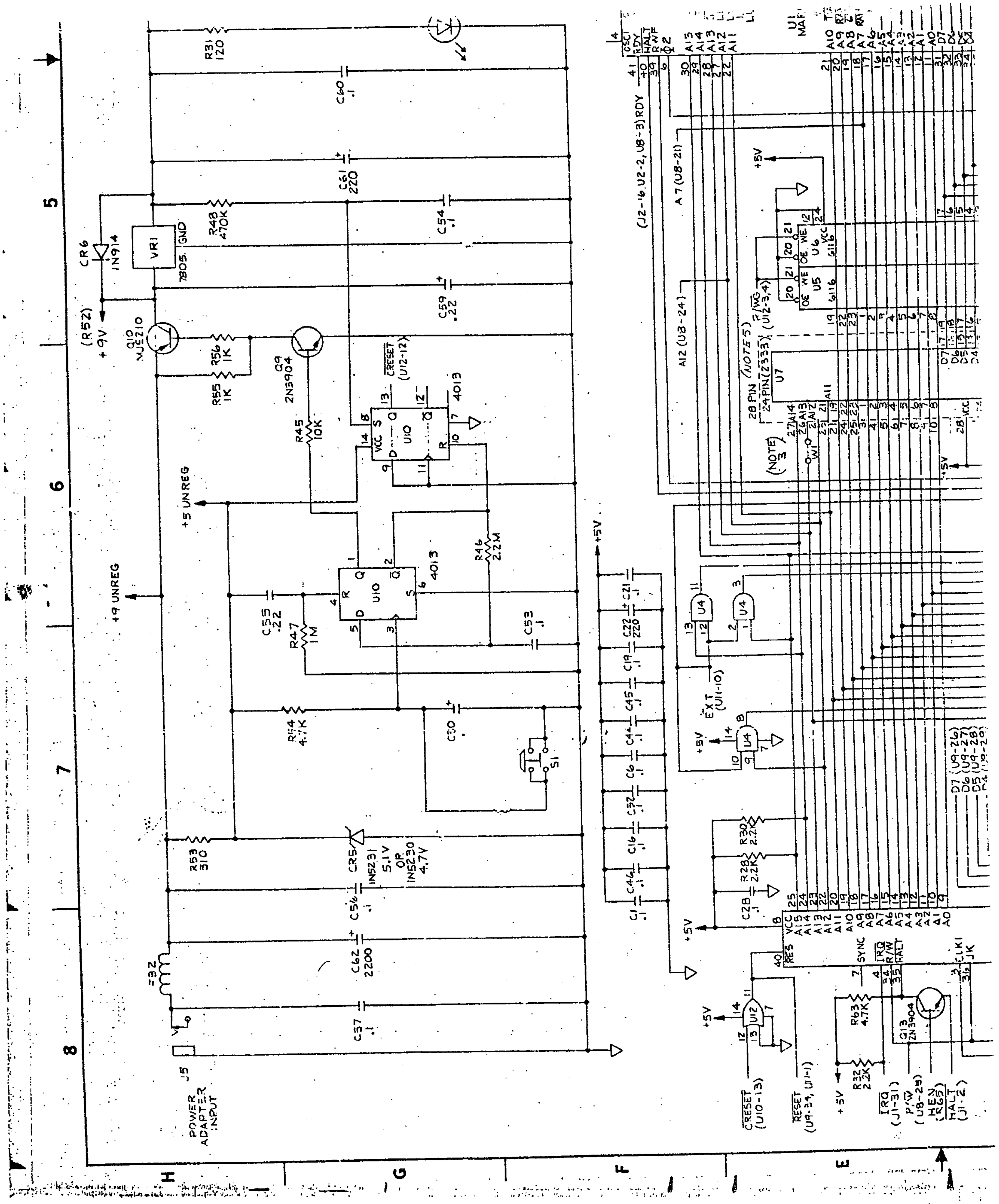
ITEM 1 — C025234 — 7800



NOTES (UNLESS OTHERWISE SPECIFIED)
1. RESISTANCE VALUES ARE IN OHMS 1/4W, 5%.
2. CAPACITANCE VALUES ARE MICROFARADS.
3. W1: USE WITH 28 PIN ROM
W2: USE WITH 24 PIN ROM
4. J3 IS LEFT PORT; J4 IS RIGHT PORT.
5. WHEN 28 PIN 256K BIT ROM IS USED
IS A 23256 TYPE POWER DOWN VERSIO
IS NOT STANDARDIZED IN THE INDUSTR



J2 PERIPHERAL PORT OUTPUT
GND
+5V
(RF1-3)CVIDEO
(U1-42,R15)MLUM0
(U1-44,U3-10)MLUM3
(U1-46)BLANK
(R9)OSCDIS
(U4-4)EXTMEN
GND
(Q4)EXTOSC
(U2-39,J1-32)CLK2
(U1-48,U3-2)MSYNC
(U1-47,U3-12)MLUM1
(U1-45,U3-5)MLUM2
(U1-43,R16)MCOL
(U1-4,U2-2,U8-3)RDY
(R6)AUDIO
GND
MEN (U1-5)
+5V
FB1
R62 2.2K
LS08
U4
R43 4.7K
(J2-8) EXTMEN
C2 .1
R23 1K
R22 1K
R18 3.3K
R21 3.3K
R20 3.3K
R19 3.3K
Q2 2N3904
TIAEN (R51,Q8)
R1 1K
R64 1K
R65 10K
HEN (Q13)
EXT (U4-12)
C43 .1
TCOL (R17)
SYNC (U3-1)
TLUM2 (U3-3)
TLUM1 (U3-4)
TLUM0 (U3-13)
U11 LS174
RESET (U9-34,U12-11)
C20 .01
EXTOSC (J2-10)
C17 .1
C18 .1
R7 1K
Q4 2N3904
R25 1K
Q3 2N3904
R26 1K
C13 .1
C11 150PF
C15 220pF
CR1 1N914
C12 47oF
Y1 14.318180 MHZ
R27 270
R9 6.8K
R8 10K
OSCDIS (J2-7)
+5V
CR2 LED
R31 120
C60 .1
C61 220
LS32
U12
(U1-41,U2-2,J2-16)RDY
(J1-1,U2-34)R/W
(U2-39)CLK2
R33 100
TIACS (U8-22)
RAMCS1 (U6-18)
RAMCS2 (U5-18)
NMI (U2-6)
MCOL (J2-15, R16)
BLANK (J2-6)
MSYNC (U3-2, J2-12)
MLUM3 (U3-10, J2-5)
MLUM2 (U3-5, J2-14)
MLUM1 (U3-12, J2-13)
MLUM0 (J2-4, R15)
CLK0 (U2-37)
R29 100
R/WG (U5-21 U6-21 U12-3)
U1 MARIA
A7 (U8-21)
(J2-16,U2-2,U8-3)RDY



POWER ADAPTER INPUT
J5
+9 UNREG
+5 UNREG
+5V
VR1 7805
CR6 1N914
(R52) +9V
Q10 MJE210
Q9 2N3904
Q13 2N3904
R31 120
R48 470K
R55 1K
R56 1K
R45 10K
R46 2.2M
R47 1M
R44 4.7K
R53 510
R30 2.2K
R28 2.2K
R63 4.7K
R32 2.2K
C60 .1
C61 220
C54 .1
C59 .22
C55 .22
C53 .1
C30
C56 .1
C62 2200
C57 .1
C1 .1
C44 .1
C16 .1
C52 .1
C6 .1
C45 .1
C19 .1
C22 220
C21 .1
C28 .1
CR5 1N5231 5.1V OR 1N5230 4.7V
S1
U10 4013
CRESET (U12-12)
CRESET (U10-13)
RESET (U9-34, U11-1)
EXT (U11-10)
U4
U12
U5 6116
U6 6116
U7
28 PIN (NOTE 5) 24 PIN (2332)
(NOTE 3)
R/WG (U12-3,4)
A12 (U8-24)
A12 (U8-21)
A7 (U8-21)
(J2-16, U2-2, U8-3) RDY
U1
IRQ (J1-31)
R/W (U8-25)
HEN (R65)
HALT (J1-2)
D7 (U9-26)
D6 (U9-27)
D5 (U9-28)
D4 (U9-29)
VCC RES A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
SYNC IRQ R/W HALT CLK1 JK
H
G
F
E
5
6
7
8