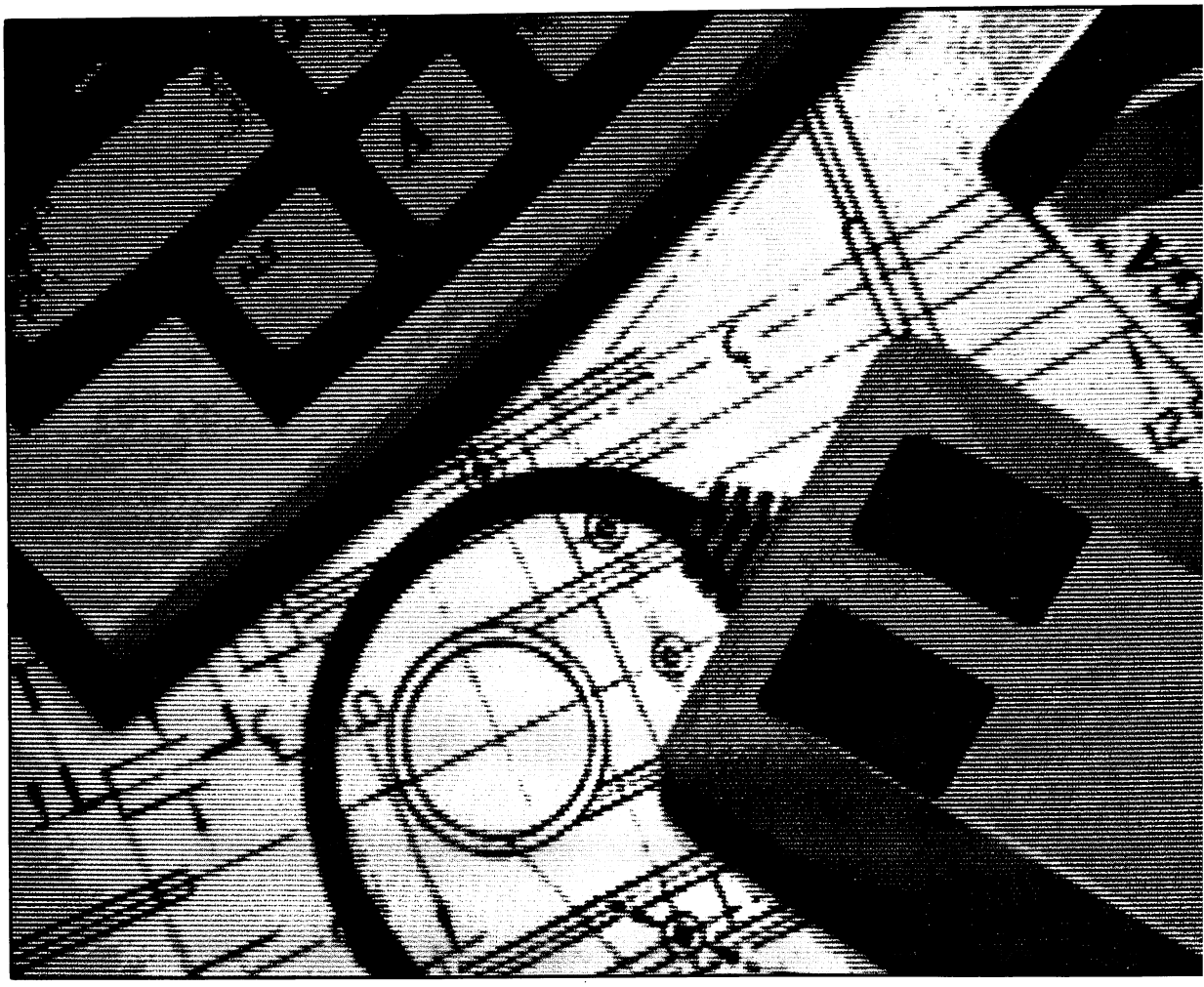
AMIGA™



Hardware Manual

AMIGA

HARDWARE MANUAL

COMMODORE-AMIGA, INC.

The text of this manual was written by Robert Peck, Susan Deyl, and Jay Miner.

DISCLAIMER

COMMODORE-AMIGA, INC., ("COMMODORE") MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THE PROGRAM DESCRIBED HEREIN, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. THIS PROGRAM IS SOLD "AS IS." THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAM PROVE DEFECTIVE FOLLOWING ITS PURCHASE, THE BUYER (AND NOT THE CREATOR OF THE PROGRAM, COMMODORE, THEIR DISTRIBUTORS OR THEIR RETAILERS) ASSUMES THE ENTIRE COST OF ALL NECESSARY DAMAGES. IN NO EVENT WILL COMMODORE BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME LAWS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

CBM Product Number 327272-01 rev 1.0 8.27.85

PREFACE

This manual provides information about the Amiga[tm] graphics and audio hardware and about how the Amiga talks to the outside world through peripheral devices. A portion of this manual is a tutorial on writing assembly language programs to directly control the Amiga's graphics and hardware.

This book is intended for the following audiences:

- Assembly language programmers who need a more direct way of interacting with the system than the routines described in the *Amiga ROM Kernel Manual.* You can find information here to help you make your programs run faster or do things that the ROM kernel routines don't do.
- Anyone who wants to add new peripherals to the Amiga or just wants to know how the hardware works

We suggest that you use this book according to your level of familiarity with the Amiga system. Here are some suggestions:

- If this is your initial exposure to the Amiga, Chapter 1 gives a survey of all the hardware features and a brief rundown of graphics and audio effects created by hardware interaction.
- If you are already familiar with the system and want to acquaint yourself with how the various bits in the hardware registers govern the way the system functions, browse through chapters 2 through 8. Examples are included in these chapters.
- For advanced users, the Appendices give a concise summary of the entire register set and the uses of the individual bits. Once you are familiar with the effects of changes in the various bits, you may wish to refer more often to the Appendices than to the explanatory chapters.

Here is a brief overview of the contents:

Chapter 1, *Introduction.* An overview of the hardware and survey of the Amiga's graphics and audio features.

Chapter 2, *Coprocessor Hardware.* Using the Copper coprocessor to control the entire graphics and audio system; directing mid-screen modifications in graphics displays and directing register changes during the time between displays.

Chapter 3, *Playfield Hardware.* Creating, displaying and scrolling the playfields, one of the basic display elements of the Amiga; how the Amiga produces multi-color, multi-graphical bit-mapped displays.

Chapter 4, *Sprite Hardware.* Using the 8 sprite direct-memory access (DMA) channels to make sprite movable objects; creating their data structures, displaying and moving them, reusing the DMA channels.

Chapter 5, *Audio Hardware.* Overview of sampled sound; how to produce quality sound, simple and complex sounds, and modulated sounds.

Chapter 6, *Blitter Hardware.* Using the blitter DMA channel to create animation effects and draw lines into playfields.

Chapter 7, *System Control Hardware.* Using the control registers to define depth arrangement of graphics objects, detect collisions between graphics objects, control direct memory access, and control interrupts.

Chapter 8, *Interface Hardware.* How the Amiga talks to the outside world through controller ports, keyboard, audio jacks and video connectors, serial and parallel interfaces; information about the disk controller and RAM expansion slot.

Appendices. Alphabetical and address-order listings of all the graphics and audio system registers and the functions of their bits; system memory map; descriptions of internal and external connectors; specifications for the peripheral interface ports; specifications for the expansion connector; and specifications for the keyboard.

Glossary. After the appendices, there is a glossary of important terms.

You may wish to look at the following books and manuals for further information about the Amiga:

- The *Amiga ROM Kernel Manual* contains information about the Exec multi-tasking routines and is the source for all the C language primitives for Amiga graphics, animation, and audio.
- The following manuals contain information about the AmigaDOS operating system:
  - *AmigaDOS User's Manual*
  - *AmigaDOS Developer's Manual*
  - *AmigaDOS Technical Reference Manual*

It is our policy to make certain that the information contained here is accurate, consistent, and up-to-date. If you should find any material confusing, inaccurate, or incomplete, please feel free to contact Amiga with your questions or comments.

AMIGA HARDWARE MANUAL

Table of Contents

Chapter 1 INTRODUCTION **1-1**
1.1. COMPONENTS OF THE AMIGA 1-1
1.2. SYSTEM EXPANDABILITY AND ADAPTABILITY 1-5

Chapter 2 COPROCESSOR HARDWARE **2-1**
2.1. INTRODUCTION 2-1
2.2. WHAT IS A COPPER INSTRUCTION? 2-3
2.3. THE MOVE INSTRUCTION 2-4
2.4. THE WAIT INSTRUCTION 2-5
2.5. USING THE COPPER REGISTERS 2-7
2.5.1. Location Registers 2-7
2.5.2. Jump Strobe Address 2-8
2.5.3. Control Register 2-8
2.6. PUTTING TOGETHER A COPPER INSTRUCTION LIST 2-9
2.7. STARTING AND STOPPING THE COPPER 2-11
2.8. ADVANCED TOPICS 2-13
2.8.1. The SKIP Instruction 2-13
2.8.2. Copper Loops and Branches and Comparison Enable 2-13
2.8.3. Using the Copper in Interlace Mode 2-15
2.8.4. Using the Copper with the Blitter 2-16
2.8.5. The Copper and the 68000 2-16
2.9. SUMMARY OF COPPER INSTRUCTIONS 2-17

Chapter 3 PLAYFIELD HARDWARE **3-1**
3.1. INTRODUCTION 3-1
3.2. FORMING A BASIC PLAYFIELD 3-7
3.2.1. Height and Width of the Playfield 3-7
3.2.2. Bit-Planes and Color 3-7
3.2.3. Selecting Horizontal and Vertical Resolution 3-10
3.2.4. Allocating Memory for Bit-Planes 3-12
3.2.5. Coding the Bit-Planes for Correct Coloring 3-14
3.2.6. Defining the Size of the Display Window 3-16
3.2.7. Telling the System How to Fetch and Display Data 3-19
3.2.8. Displaying and Redisplaying the Playfield 3-22
3.2.9. Enabling the Color Display 3-23

3.2.10. Summary 3-23
3.3. FORMING A DUAL PLAYFIELD DISPLAY 3-27
3.3.1. How Bit-Planes are Assigned in Dual Playfield Mode 3-29
3.3.2. How Color Registers are Assigned in Dual Playfield Mode 3-31
3.3.3. Dual Playfield Priority and Control 3-32
3.3.4. Activating Dual Playfield Mode 3-33
3.3.5. Summary 3-33
3.4. BIT-PLANES AND DISPLAY WINDOWS OF ALL SIZES 3-33
3.4.1. When the Big Picture is Larger than the Display Window 3-34
3.4.2. Maximum Display Window Size 3-42
3.5. MOVING (SCROLLING) PLAYFIELDS 3-43
3.5.1. Vertical Scrolling 3-43
3.5.2. Horizontal Scrolling 3-45
3.5.3. Summary 3-50
3.6. ADVANCED TOPICS 3-52
3.6.1. Interactions Between Playfields and Other Objects 3-52
3.6.2. Hold and Modify Mode 3-52
3.6.3. Forming a Display with Several Different Playfields 3-53
3.6.4. Using an External Video Source 3-53
3.7. SUMMARY OF PLAYFIELD REGISTERS 3-54
3.8. SUMMARY OF COLOR SELECTION 3-56
3.8.1. Color Register Contents 3-56
3.8.2. Some Sample Color Register Contents 3-56
3.8.3. Color Selection in Low Resolution Mode 3-57
3.8.4. Color Selection in High Resolution Mode 3-59

Chapter 4 SPRITE HARDWARE **4-1**
4.1. INTRODUCTION 4-1
4.2. FORMING A SPRITE 4-2
4.2.1. Screen Position 4-2
4.2.2. Shape of Sprites 4-5
4.2.3. Sprite Color 4-7
4.2.4. Designing a Sprite 4-9
4.2.5. Building the Data Structure 4-9
4.3. DISPLAYING A SPRITE 4-14
4.3.1. Selecting the Sprite DMA Channel and Setting the Pointers 4-14
4.3.2. Resetting the Address Pointers 4-15
4.4. MOVING A SPRITE 4-15
4.5. CREATING ADDITIONAL SPRITES 4-16
4.6. REUSING SPRITE DMA CHANNELS 4-18
4.7. OVERLAPPED SPRITES 4-20
4.8. ATTACHED SPRITES 4-23
4.9. MANUAL MODE 4-25
4.10. SPRITE HARDWARE DETAILS 4-25
4.11. SUMMARY OF SPRITE REGISTERS 4-29

4.11.1. Pointers 4-29
4.11.2. Control Registers 4-29
4.11.3. Data registers 4-31
4.12. SUMMARY OF SPRITE COLOR REGISTERS 4-32

Chapter 5 AUDIO HARDWARE **5-1**
5.1. INTRODUCTION 5-1
5.2. FORMING AND PLAYING A SOUND 5-6
5.2.1. Deciding Which Channel to Use 5-6
5.2.2. Creating the Waveform Data 5-6
5.2.3. Telling the System About the Data 5-7
5.2.4. Selecting the Volume 5-8
5.2.5. Selecting the Data Output Rate 5-9
5.2.6. Playing the Waveform 5-12
5.2.7. Stopping the Audio DMA 5-13
5.2.8. Summary 5-14
5.2.9. Example 5-14
5.3. PRODUCING COMPLEX SOUNDS 5-16
5.3.1. Joining Tones 5-16
5.3.2. Playing Multiple Tones at the Same Time 5-17
5.3.3. Modulating Sound 5-18
5.4. PRODUCING QUALITY SOUND 5-21
5.4.1. Making Waveform Transitions 5-21
5.4.2. Sampling Rate 5-21
5.4.3. Efficiency 5-22
5.4.4. Noise Reduction 5-23
5.4.5. Aliasing Distortion 5-23
5.4.6. Low Pass Filter 5-26
5.5. USING DIRECT (NON-DMA) AUDIO OUTPUT 5-28
5.6. THE EQUAL-TEMPERED MUSICAL SCALE 5-29
5.7. DECIBEL VALUES FOR VOLUME RANGES 5-31

Chapter 6 BLITTER HARDWARE **6-1**
6.1. INTRODUCTION 6-1
6.2. DATA COPYING 6-2
6.3. MULTIPLE SOURCES 6-3
6.4. LOGIC OPERATIONS 6-3
6.4.1. Blitter Logic Operations - Combining Minterms 6-4
6.4.2. Table of Commonly Used Equations 6-5
6.4.3. Equation to Minterm Conversion 6-6
6.5. MULTIPLE MODULOS 6-7
6.6. ASCENDING AND DESCENDING ADDRESSING 6-8
6.7. SHIFTING 6-8
6.8. MASKING 6-9
6.9. ZERO DETECTION 6-10

6.10. AREA FILLING 6-10
6.10.1. Inclusive (Normal) Area Filling 6-10
6.10.2. Exclusive Area Filling 6-12
6.11. LINE DRAWING 6-13
6.12. VENN DIAGRAMS 6-16

Chapter 7 SYSTEM CONTROL HARDWARE **7-1**
7.1. INTRODUCTION 7-1
7.2. VIDEO PRIORITIES 7-1
7.2.1. Fixed Sprite Priorities 7-1
7.2.2. How Sprites Are Grouped 7-2
7.2.3. Understanding Video Priorities 7-2
7.2.4. Setting the Priority Control Register 7-4
7.3. COLLISION DETECTION 7-6
7.3.1. How Collisions Are Determined 7-6
7.3.2. How to Interpret the Collision Data 7-6
7.3.3. How Collision Detection is Controlled 7-7
7.4. BEAM POSITION DETECTION 7-9
7.4.1. Using the Beam Position Counter 7-9
7.5. INTERRUPTS 7-10
7.5.1. Non-Maskable Interrupt 7-10
7.5.2. Maskable Interrupts 7-10
7.5.3. User Interface to the Interrupt System 7-10
7.5.4. Interrupt Control Registers 7-10
7.5.5. Setting and Clearing Bits 7-11
7.6. DMA CONTROL 7-15

Chapter 8 INTERFACE HARDWARE **8-1**
8.1. INTRODUCTION 8-1
8.2. CONTROLLER PORT INTERFACE 8-1
8.2.1. How to Read The Controller Port 8-2
8.3. DISK CONTROLLER 8-13
8.3.1. Disk Selection, Control and Sensing 8-13
8.3.2. Other Registers Involved with Disk Operations 8-16
8.3.3. Disk Interrupts 8-19
8.4. THE KEYBOARD 8-19
8.4.1. How the Keyboard Data is Received 8-20
8.4.2. Type of Data Received 8-20
8.4.3. Limitations of the Keyboard 8-23
8.5. PARALLEL INPUT/OUTPUT INTERFACE 8-25
8.6. SERIAL INTERFACE 8-25
8.6.1. Introduction to Serial Circuitry 8-25
8.6.2. Setting the Baud Rate 8-25
8.6.3. Setting the Receive Mode 8-25
8.6.4. Contents of the Receive Data Register 8-26

8.6.5. How Output Data is Transmitted 8-27
8.6.6. How to Specify the Register Contents 8-29
8.7. AUDIO OUTPUT CONNECTIONS 8-29
8.8. DISPLAY OUTPUT CONNECTIONS 8-30

Appendix A REGISTER SET SUMMARY -- ALPHABETICAL ORDER **A-1**

Appendix B REGISTER SET SUMMARY -- ADDRESS ORDER **B-1**

Appendix C PIN ALLOCATION LIST **C-1**

Appendix D SYSTEM MEMORY MAP **D-1**

Appendix E INTERFACES **E-1**

Appendix F PERIPHERAL INTERFACE ADAPTERS **F-1**

Appendix G EXPANSION CONNECTOR **G-1**

Appendix H KEYBOARD **H-1**

GLOSSARY Glossary-1

Chapter 1

INTRODUCTION

The Amiga is a low-cost, high-performance computer with advanced graphics and sound features. This chapter lists and describes the Amiga's hardware components and gives a brief overview of its graphics and sound features.

1.1. COMPONENTS OF THE AMIGA

These are the hardware components of the Amiga:

- Motorola MC 68000 16/32-bit main processor
- 256K bytes of internal RAM, expandable to 512K
- 256K bytes of ROM containing a real-time, multi-tasking operating system with sound, graphics, and animation support routines
- built-in 3 1/2-inch double-sided disk drive
- expansion disk port for connecting up to 3 additional disk drives, which may be either 3-1/2 inch or 5-1/4 inch, double-sided
- fully programmable serial port
- fully programmable parallel port
- two-button opto-mechanical mouse
- two reconfigureable controller ports (for mice, joysticks, paddles, or custom controllers)
- detached 89-key keyboard with calculator pad, function keys, and cursor keys
- ports for simultaneous composite video, and analog or digital RGB output
- ports for audio output to left and right stereo channels from 4 special-purpose audio channels
- expansion connector that allows you to add RAM, additional disk drives (floppy or hard disks), peripherals, or coprocessors

The MC 68000 and the Special-Purpose Hardware

The Motorola 68000 is a 16/32-bit microprocessor operating at 7.16 megahertz. In the Amiga, the 68000 can address over 8 megabytes of contiguous random access memory (RAM).

Performance of the 68000 is enhanced by a system design that gives it every alternate bus cycle, allowing it to run at full rated speed most of the time. As described in the section below, the special purpose hardware can steal time from the 68000 for jobs it can do more efficiently than the 68000. Even then, such cycle stealing only blocks the 68000's access to the shared memory. When using ROM or external memory, the 68000 always runs at full speed.

Among other functions, the special-purpose hardware provides the following:

- bit-plane generated high-resolution graphics typically producing 320 by 200 non-interlaced displays and 320 by 400 interlaced displays in 32 colors, and 640 by 200 non-interlaced or 640 by 400 interlaced displays in 16 colors. There is also a special mode that allows you to have up to 4096 colors on-screen simultaneously.

- a custom display coprocessor, allowing changes to most of the special-purpose registers in synchronization with the position of the video beam. This allows such special effects as mid-screen changes to the color palette, splitting the screen into multiple horizontal slices each having different video resolutions and color depths, beam-synchronized interrupt generation for the 68000, and more. The coprocessor can trigger many times per screen. It can trigger in the middle of lines, as well as at the beginning or during the blanking interval. The coprocessor itself can directly affect most of the registers of the special-purpose hardware, freeing the 68000 for general purpose computing tasks.

- 32 system color registers, each of which contains a 12-bit number as 4 bits of RED, 4 bits of GREEN, and 4 bits of BLUE intensity information. This allows a system color palette of 4096 different choices of color for each register. Although an RGB monitor provides the best available output for the system graphics, test, and color, the composite video signal has been carefully designed to provide maximum NTSC compatibility. This signal may be video-taped or fed to a standard composite video monitor.

- 8 reusable 16-bit-wide sprites with up to 15 color choices per sprite pixel (when sprites are paired). A sprite is an easily movable graphics object whose display is entirely independent of the background (called a playfield); sprites can be displayed "over" or "under" this background. A sprite is 16 low-resolution pixels wide and an arbitrary number of lines tall. After producing the last line of a sprite on the screen, a sprite DMA (direct memory access) channel may be used to produce yet another sprite image elsewhere on-screen (with at least one horizontal line between each reuse of a sprite processor). Thus, you can produce many, many small sprites by simply reusing the sprite processors appropriately.

- dynamically-controllable inter-object priority, with collision detection. This means that the system can dynamically control the video priority between the sprite objects and the bit-plane backgrounds (playfields). You can control which object or objects appear "on top" at any time.

  Additionally, you can use system hardware to detect collisions between objects and have your program react to such collisions.

- custom bit-blitter used for high speed data movement, adaptable to bit-plane animation. The blitter has been designed to efficiently retrieve data from up to 3 sources, combine the data in one of 256 different possible ways, and optionally store the combined data in a destination area. This is one of the situations where the 68000 gives up memory cycles to a DMA channel that can do the job more efficiently. The bit-blitter, in a special mode, draws patterned lines into rectangularly organized memory regions at a speed of about 1 million dots per second; and it can efficiently handle area fill.

- audio consisting of 4 low-noise digital channels with independently programmable volume and sampling rate. The audio channels retrieve their control and data via direct memory access. Once started, each channel can automatically play a specified waveform without further processor interaction. Two channels are directed into each of the two stereo audio outputs. The audio channels may be linked together if desired to provide amplitude or frequency modulation or both forms of modulation simultaneously.

- DMA-controlled floppy disk read and write on a full-track basis. This means that the built-in disk can read something over 5.6K bytes of data in a single disk revolution (11 sectors of 512 bytes each).

All of the special functions described above are produced by three custom-designed VLSI circuits, which work in concert with the 68000. These circuits and the 68000 use the shared memory on a fully interleaved basis. Since the 68000 only needs to access the memory bus during each alternate clock cycle in order to run full-speed, the rest of the time the memory bus is free for other activities.

The special purpose hardware uses the memory bus during these free cycles, effectively allowing the 68000 to run at full rated speed "most of the time". We say "most of the time" because there are some occasions when the special purpose hardware steals memory cycles from the 68000, but with good reason. Specifically, the coprocessor and the data-moving DMA channel called the blitter can each steal time from the 68000 for jobs they can do better than the 68000. Thus, the system DMA channels are designed with maximum performance in mind, where the job to be done is performed by the most efficient hardware element available. In addition, sprites, audio, and disk DMA also steal cycles when in operation.

Another primary feature of the Amiga hardware is the ability to dynamically control which part of memory is used for the background display, audio, and sprites. The Amiga is not limited to a small, specific area of RAM for a frame buffer. Instead, the system allows display bit-planes, sprite-processor control lists, coprocessor instruction lists, or audio channel control lists to be located anywhere within the lowest 512K of the memory map.

This same region of memory can be accessed by the bit-blitter. This means, for example, that the user can store partial images at scattered areas of memory and use these images for animation effects by means of rapid replacement of on-screen material while saving and

restoring background images. In fact, the Amiga includes firmware support for display definition and control as well as support for animated objects embedded within playfields.

VCR and Direct Camera Interface

In addition to the connections for NTSC composite Amiga video, and both digital and analog RGB monitors, the system can be expanded to include a VCR or camera interface. This system is capable of synchronizing with an external video source and replacing the system background color with the external image. This allows for the development of fully integrated video images with computer-generated graphics. Laser disk input is accepted in the same manner.

Primary and Secondary Memory

Primary memory in the Amiga consists of 256K bytes of ROM and 256K bytes of RAM. A RAM expansion cartridge is available as an option. Secondary memory is provided by a built-in 3 1/2-inch floppy disk drive. Disks are 80-track, double-sided, formatted as 11 sectors per track, 512 bytes per sector (over 900,000 bytes per disk). A special utility can read and write disk files compatible with the Apple II[tm]. In addition, the disk controller can read and write 320/360K IBM PC[tm] formatted disks. External 3 1/2-inch or 5 1/4-inch disk drives can be added to the system through the expansion connector.

Peripherals

Circuitry for some of the peripherals resides on one of the custom chips; other chips handle various signals not specifically assigned to any of the custom chips, including modem controls, disk status sensing, disk motor and stepping controls, ROM enable, parallel input/output interface, and keyboard interface.

The Amiga includes a standard RS-232 serial port for external serial input/output devices.

A detached, professional quality keyboard is included in the base system. You can store the keyboard beneath the system cabinet. For maximum flexibility, both key-down and key-up signals are sent.

For those who prefer incremental cursor control, there are cursor keys on the keyboard. You can attach many other types of controllers through the two controller ports on the side of

the base unit. You can use a mouse, joystick, keypad, trackball, or steering wheel controller in either of the controller ports. (A light pen can be attached to Port 0.)

1.2. SYSTEM EXPANDABILITY AND ADAPTABILITY

You can add peripheral devices to the Amiga's expansion connector, and add additional external RAM on the same expansion connector or upgrade internal RAM to 512K. Additional disk units may be daisy-chained from a connector at the rear of the unit for a total of 3 extra drives.

The system software, as well, is highly adaptable to other host operating systems. The Amiga's graphics support routines are designed to make the user interface as friendly as possible. New peripheral devices are recognized and used by system software through a well defined, well documented linking procedure.

Chapter 2

COPROCESSOR HARDWARE

2.1. INTRODUCTION

The Copper is a general purpose coprocessor which resides in one of the Amiga's custom chips. It retrieves its instructions via direct memory access (DMA). The Copper can control nearly the entire graphics system, freeing the 68000 to execute program logic; and it can directly affect the contents of most of the chip control registers. It is a very powerful tool for directing mid-screen modifications in graphics displays and for directing the register changes that must occur during the vertical blanking periods. Among other things, it can control register updates, reposition sprites, change the color palette, update the audio channels, and control the blitter.

One of the features of the Copper is its ability to WAIT for a specific video beam position, then MOVE data into a system register. During the WAIT period, the Copper examines the contents of the video beam position counter directly. This means that while the Copper is waiting for the beam to reach a specific position, it does not use the memory bus at all. Therefore, the bus is freed for use by the other DMA channels or by the 68000.

When the WAIT condition has been satisfied, the Copper steals memory cycles from either the blitter or the 68000 to move the specified data into the selected special-purpose register.

The Copper is a 2-cycle processor that requests the bus only during odd-numbered memory cycles. This prevents collision with audio, disk, refresh, sprites, and most low-resolution display DMA access, all of which use only the even-numbered memory cycles. The Copper, therefore, only needs priority over the 68000 and the blitter (DMA channel that handles animation, line drawing, and polygon filling).

As with all the other DMA channels in the Amiga system, the Copper can only retrieve its instructions from the lowest 512K of system memory.

About This Chapter

In this chapter, you will learn how to use the special Copper instruction set to organize mid-screen register value modifications and pointer register setup during the vertical blanking interval. The chapter shows how to organize Copper instructions into Copper lists, how to use Copper lists in interlace mode, and how to use the Copper with the blitter. The Copper is discussed in this chapter in a general fashion. The chapters that deal with playfields,

sprites, audio, and the blitter contain more specific suggestions for using the Copper.

2.2. WHAT IS A COPPER INSTRUCTION?

As a coprocessor, the Copper adds its own instruction set to the instructions already provided by the 68000. The Copper has only three instructions, but you can do a lot with them:

- WAIT for a specific screen position specified as x and y coordinates.
- MOVE an intermediate value into one of the special purpose registers.
- SKIP the next instruction if the video beam has already reached a specified screen position.

All Copper instructions consist of two 16-bit words in sequential memory locations. Each time the Copper fetches an instruction, it fetches both words. The MOVE and SKIP instructions require 2 memory cycles and 2 instruction words. Since only the odd memory cycles are requested by the Copper, 4 memory cycle times are required per instruction. The WAIT instruction requires 3 memory cycles and 6 memory cycle times; it takes one extra memory cycle to wake up.

Although the copper can directly affect only machine registers, it can affect the memory by setting up a blitter operation. More information about how to use the Copper in controlling the blitter can be found in the sections called "Control Register" and "Using the Copper with the Blitter".

The WAIT and MOVE instructions are described below. The SKIP instruction is described in the "Advanced Topics" section.

2.3. THE MOVE INSTRUCTION

The MOVE instruction transfers data from RAM to a register destination. The transferred data is contained in the second word of the MOVE instruction; the first word contains the address of the destination register. This is shown in detail in the section called "Summary of Copper Instructions".

FIRST INSTRUCTION WORD (IR1)

Bit 0	Always set to 0.
Bits 8 - 1	Register destination address (DA8-1).
Bits 15 - 9	Not used, but should be set to 0.

SECOND INSTRUCTION WORD (IR2)

Bits 15 - 0	16 bits of data to be transferred (moved) to the register destination.

The Copper can store data into the following registers:

- Any register whose address is hex 20 or above.
- Any register whose address is between hex 10 and hex 20 if the Copper danger bit is a 1. The Copper danger bit is in the Copper's control register, COPCON, which is described in the "Control Register" section.
- The Copper cannot write into any register whose address is lower than hex 10.

Appendix B at the end of this manual contains all the machine register addresses.

2.4. THE WAIT INSTRUCTION

The WAIT instruction causes the Copper to wait until the video beam counters are equal to (or greater than) the coordinates specified in the instruction. While waiting, the Copper is off the bus and not using memory cycles.

The first instruction word contains the vertical and horizontal coordinates of the beam position. The second word contains enable bits that are used to form a "mask" that tells the system which bits of the beam position to use in making the comparison.

FIRST INSTRUCTION WORD (IR1)

Bit 0	Always set to 1.
Bits 15 - 8	Vertical beam position (called VP).
Bits 7 - 1	Horizontal beam position (called HP).

SECOND INSTRUCTION WORD (IR2)

Bit 0	Always set to 0.
Bit 15	The Blitter Finished Disable bit. Normally, this bit is a 1. (See the "Advanced Topics" section below.)
Bits 14 - 8	Vertical position compare enable bits (called VE).
Bits 7 - 1	Horizontal position compare enable bits (called HE).

The following notes apply to both the WAIT instruction and to the SKIP instruction, which is described below in the "Advanced Topics" section.

Horizontal Beam Position

The horizontal beam position has a value of $0 to $E2 (hex).[1] The least significant bit is not used in the comparison, so there are 113 positions available for Copper operations. This corresponds to 4 pixels in low resolution and 8 pixels in high resolution. Horizontal blanking falls in the range of $0F to $35. The standard screen (320 pixels wide) has an unused horizontal portion of $04 to $47 (during which only the background color is displayed).

[1] In this manual hexadecimal numbers are distinguished from decimal numbers by the $ prefix.

Vertical Beam Position

The vertical beam position can be resolved to 1 line, with a maximum value of 255. There are actually 262 possible vertical positions. Some minor complications can occur if you want something to happen within these last 6 or 7 scan lines. Since there are only 8 bits of resolution for vertical beam position (allowing 256 different positions), one of the simplest ways to handle this is shown in the Copper sequence below.

INSTRUCTION	EXPLANATION
[... other instructions ...]	
WAIT for position (0,255)	*At this point, the vertical counter appears to wrap to 0, since the comparison works on the least significant bits of the vertical count.*
WAIT for any horizontal position with vertical position 0 through 6, covering the last 6 lines of the scan before vertical blanking occurs.	*Thus the total of 256 + 6 = 262 lines of video beam travel during which Copper instructions can be executed.*

The Comparison Enable Bits

Bits 14-1 are normally set to all 1's. The use of the comparison enable bits is described later in the "Advanced Topics" section.

2.5. USING THE COPPER REGISTERS

There are several machine registers and strobe addresses dedicated to the Copper:

- Location registers
- Jump address strobes
- Control register

2.5.1 Location Registers

The Copper has two sets of location registers, which are:

COP1LCH	High 3 bits of first Copper list address
COP1LCL	Low 16 bits of first Copper list address
COP2LCH	High 3 bits of second Copper list address
COP2LCL	Low 16 bits of second Copper list address

In accessing the hardware directly, you often have to write to a pair of registers that contain the address of some data. The register with the lower address always has a name ending in "H" and contains the most significant data, or high 3 bits of the address. The register with the higher address has a name ending in "L" and contains the least significant data, or low 15 bits of the address. Therefore, you write the 18-bit address by moving one long word to the register whose name ends in "H". This is because when you write long words with the 68000, the most significant word goes in the lower addressed word.

In the case of the copper location registers, you write the address to COP1LCH. In the following text, for simplicity, both addresses are referred to as COP1LC or COP2LC.

The copper location registers contain the two indirect jump addresses used by the Copper. The Copper fetches its instructions by using its program counter and increments the program counter after each fetch. When a jump address strobe is written, the corresponding location register is loaded into the Copper program counter. This causes the Copper to jump to a new location from which its next instruction will be fetched. Instruction fetch continues sequentially until the Copper is interrupted by another jump address strobe.

NOTE

At the start of each vertical blanking interval, COP1LC is automatically used to start the program counter. That is, no matter what the Copper is doing, when vertical blanking reset occurs, the Copper is automatically freed to restart its operations at the address contained in COP1LC.

2.5.2. Jump Strobe Address

When you write to a strobe address, the Copper reloads its program counter from the corresponding location register. The Copper can write its own location registers to perform programmed jumps. For instance, you might MOVE an indirect address into the COP2LC location register. Then, any MOVE instruction that addresses COPJMP2, loads this indirect address into into the program counter.

There are two jump strobe addresses:

COPJMP1	Restart Copper from address contained in COP1LC
COPJMP2	Restart Copper from address contained in COP2LC

2.5.3. Control Register

The Copper can access:

A. Some special-purpose registers all of the time,
B. Some registers only when a special control bit is set to a 1,
C. Some registers not at all.

The registers that the Copper can always affect are numbered from $20 through $FF inclusive. Those it cannot affect at all are numbered from $00 to $0F inclusive. (See Appendix B at the end of this manual for a list of registers in address order.) The Copper control register is within this third, always protected, group. This means that it takes deliberate action on the part of the 68000 to allow the Copper to write into a specific range of the special-purpose registers.

The Copper control register, called COPCON, contains only one bit, Bit #1. This bit, called CDANG (for Copper Danger Bit) protects all registers numbered between $10 and $1F inclusive. This includes the blitter control registers. When CDANG is a 0, these registers cannot be written by the Copper. When CDANG is a 1, these registers can be written by the Copper. Preventing the Copper from accessing the blitter control registers prevents a "runaway" Copper (caused by a poorly formed instruction list), from accidentally affecting system memory.

NOTE

The CDANG bit is cleared after a reset.

2.6. PUTTING TOGETHER A COPPER INSTRUCTION LIST

The Copper instruction list contains all the register resetting done during the vertical blanking interval and the register modifications necessary for making mid-screen alterations. As you are planning what will happen during each display field, you may find it easier to think of each aspect of the display as a separate subsystem, such as playfields, sprites, audio, interrupts, and so on. Then you can build a separate list of things that must be done for each subsystem individually at each video beam position.

When you have created all these intermediate lists of things to be done, you must merge them together into a single instruction list to be executed by the Copper once for each display frame. The alternative is to create this all-inclusive list directly, without the intermediate steps.

For example, the bit-plane pointers used in playfield displays and the sprite pointers must be rewritten during the vertical blanking interval so the data will be properly retrieved when the screen display starts again. This can be done with a Copper instruction list that does the following:

```
WAIT until first line of the display
MOVE data to bit-plane pointer 1
MOVE data to bit-plane pointer 2
MOVE data to sprite pointer 1
and so on
```

As a further example, the sprite DMA channels that create movable objects can be reused multiple times during the same display field. You can change the size and shape of the multiple reuses of a sprite; however, every reuse will normally use the same set of colors during a full display frame. You can change sprite colors mid-screen with a Copper instruction list that waits until the last line of the first use of the sprite processor and changes the colors before the first line of the next use of the same sprite processor:

```
WAIT for first line of display
MOVE firstcolor1 to COLOR17
MOVE firstcolor2 to COLOR18
MOVE firstcolor3 to COLOR19
WAIT for last line +1 of sprite's first use
MOVE secondcolor1 to COLOR17
MOVE secondcolor2 to COLOR18
MOVE secondcolor3 to COLOR19
and so on
```

As you create Copper instruction lists, note that the final list must be in the same order as the order in which the video beam creates the display. The video beam traverses the screen from position (0,0) in the upper left hand corner of the screen to the end of the display (226,263) in the lower right hand corner. The first 0 in (0,0) represents the x, position. The second 0 represents the y position. For example, an instruction that does something at

position (0,100) should come after an instruction that affects the display at position (0,60).

Note that because of the form of the WAIT instruction, you can sometimes get away with not sorting the list in strict video beam order. The WAIT instruction causes the Copper to wait until the value in the beam counter is equal to or greater than the value in the instruction. This means, for example, if you have instructions following each other like this:

```
WAIT for position (64,64)
MOVE data

WAIT for position (60,60)
MOVE data
```

the Copper will perform both moves, even though the instructions are out of sequence. The reason for the "greater than" specification is to prevent the Copper from locking up if the beam has already passed the specified position. A side effect is that the second MOVE below will be performed:

```
WAIT for position (60,60)
MOVE data

WAIT for position (60,60)
MOVE data
```

At the time of the second WAIT in this sequence, the beam counters will be greater than the position shown in the instructions. Therefore, the second MOVE will also be performed.

Note also that the above sequence of instructions can just as easily be:

```
WAIT for position (60,60)
MOVE data
MOVE data
MOVE data
```

since multiple moves can follow a single WAIT.

Loops and Branches

Loops and branches in Copper lists are covered in the "Advanced Topics" section below.

2.7. STARTING AND STOPPING THE COPPER

Starting the Copper After Reset

At power-on or reset time, you must initialize one of the copper location registers (COP1LC or COP2LC) and write to its strobe address before Copper DMA is turned on. This ensures a known start address and known state. Usually, COP1LC is used because this particular register is reused each vertical blanking time. The following sequence of 68000 assembly instructions shows how to initialize a location register.

```
                                ;It is assumed that the
                                ;user has already created
                                ;correct Copper instruction
                                ;list at location "mycoplist"
  move.l mycoplist, a0
  move.l a0, COP1LCH            ;Write both COP1LCH and COP1LCL
  move.w COPJMP1, d0            ;Any access to this location forces
                                ;load from COP1LC to Copper
                                ;program counter
  move.w #SETBIT + COPPERDMA, d0
  move.w d0, DMACONW            ;Enable Copper DMA
```

Now, if the contents of COP1LC are not changed, every time vertical blanking occurs the Copper will restart at the same location for each subsequent video screen. This forms a repeatable loop which, if the list is correctly formulated, will cause the displayed screen to be stable.

Stopping the Copper

There is no stop instruction provided for the Copper. To ensure that it will stop and do nothing until the screen display ends and the program counter starts again at the top of the instruction list, the last instruction should be to WAIT for an event that cannot occur. A typical instruction is to WAIT for VP = $FF and HP = $FE. An HP of greater than $E2 is not possible. When the screen display ends and vertical blanking starts, the Copper will automatically be pointed to the top of its instruction list and this final WAIT instruction never finishes.

You can also stop the Copper by disabling its ability to use DMA for retrieving instructions or placing data. The register called DMACON controls all of the DMA channels. Bit 7,

COPEN, enables Copper DMA when set to 1.
For information about controlling the DMA, see Chapter 7, "System Control Hardware".

2.8. ADVANCED TOPICS

2.8.1. The SKIP Instruction

The SKIP instruction causes the Copper to skip the next instruction if the video beam counters are equal to, or greater than, the value given in the instruction.

The contents of the SKIP instruction's words are shown below. They are identical to the WAIT instruction, except that bit 0 of the second instruction word is a 1 to identify this as a SKIP instruction.

FIRST INSTRUCTION WORD (IR1)

Bit 0	Always set to 1.
Bits 15 - 8	Vertical position (called VP).
Bits 7 - 1	Horizontal position (called HP).
	Skip if the beam counter is equal to or greater than these combined bits (bits 15 through 1).

SECOND INSTRUCTION WORD (IR2)

Bit 0	Always set to 1.
Bit 15	The Blitter Finished Disable bit. (See "Using the Copper with the Blitter" below.)
Bits 14 - 8	Vertical position compare enable bits (called VE).
Bits 7 - 1	Horizontal position compare enable bits (called HE).

The notes about horizontal and vertical beam position found in the discussion of the WAIT instruction apply also to the SKIP instruction.

2.8.2. Copper Loops and Branches and Comparison Enable

You can change the value in the location registers at any time and use this value to construct loops in the instruction list. Before the next vertical blanking time, however, the COP1LC registers must be repointed to the beginning of the appropriate Copper list. The value in the COP1LC location registers will be restored to the Copper's program counter at the start of the vertical blanking period.

Bits 14-1 of instruction word 2 in the WAIT and SKIP instructions specify which bits of the horizontal and vertical position are to be used for the beam counter comparison. The

position in instruction word 1 and the compare enable bits in instruction word 2 are tested against the actual beam counters before any further action is taken. A position bit in instruction word 1 is used in comparing the positions with the actual beam counters if and only if the corresponding enable bit in instruction word 2 is set to 1. If the corresponding enable bit is 0, the comparison is always true. For instance, if you only care about the value in the last 4 bits of the vertical position, you only set the last 4 compare enable bits, bits (11-8) in instruction word 2.

As another example, suppose you want to issue an interrupt each time a total of 16 vertical scan lines has occurred. In addition, you want the interrupts only between lines 80 and 160. Here is a Copper instruction sequence which would do this. The enable "masks" are specified with the instructions.

Before the Copper is told to begin this set of instructions, you use the 68000 to write the address of LOOP to COP2LC.

The Copper instruction stream for the interrupt example includes instructions to do the following:

(Word 1)	WAIT for	Vertical Position (VP) = 80 Horizontal Position (HP) = 0.
(Word 2)		Mask for vertical = 111111 Mask for horizontal = 111111 *All bits of the mask are important; we want y = 80.*
LOOP:		
(Word 1)	WAIT for	VP = 0 HP = 0
(Word 2)		Mask for VP = 001111 Mask for HP = 000000 *Only the lowest 4 bits are important; we want an interrupt every 16 lines.*
(Word 1)	MOVE to	(INTREQ)
(Word 2)		Interrupt word contents *Issue an interrupt to the 68000..*
(Word 1)	SKIP	(next instruction if) VP = 160 HP = 0
(Word 2)		Mask for VP = 111111 Mask for HP = 000000 *We want this loop executed only between lines 80 and 160.*
(Word 1)	MOVE to	COPJMP2
(Word 2)		(anything)

Program counter set to address of loop.

2.8.3. Using the Copper in Interlace Mode

An interlaced bit-plane display has twice the normal number of vertical lines on the screen. Whereas a normal display has 200 lines, an interlaced display has 400 lines. In interlace mode, the video beam scans the screen twice from top to bottom, displaying 200 lines at a time. During the first scan, the odd-numbered lines are displayed. During the second scan, the even-numbered lines are displayed and interlaced with the odd-numbered ones. The scanning circuitry thus treats an interlaced display as two display fields, one containing the even-numbered lines and one containing the odd-numbered lines. The diagram below shows how an interlaced display is stored in memory.

DATA AS DISPLAYED	DATA IN MEMORY
odd field	line 1
even field	line 2
odd field	line 3
even field	line 4
.	.
.	.
.	.
odd field	line 399
even field	line 400

Figure 2-1: Interlaced Bit-Plane in RAM - 400 Lines Long

The system retrieves data for bit-plane displays by using pointers to the starting address of the data in memory. As you can see, the starting address for the even-numbered fields is one line greater than the starting address for the odd-numbered fields. Therefore, the bit-plane pointer must contain a different value for alternate fields of the interlaced display. This means that two separate Copper instruction lists are required.

To get the Copper to execute the correct list, you set an interrupt to the 68000 just after the first line of the display. When the interrupt is executed, you change the contents of the COP1LC location register to point to the second list. Then, during the vertical blanking interval, COP1LC will be automatically reset to point to the original list.

For more information about interlaced displays, see Chapter 3, "Playfield Hardware".

2.8.4. Using the Copper with the Blitter

If the Copper is used to start up a sequence of blitter operations, it must wait for the blitter-finished interrupt before starting another blitter operation. Changing blitter registers while the blitter is operating causes unpredictable results. For just this purpose, the WAIT instruction includes an additional control bit, called BFD (for blitter finished disable). Normally, this bit is a 1 and only the beam counter comparisons control the WAIT.

When the BFD bit is a 0, the logic of the Copper WAIT instruction is modified. The Copper will WAIT until the beam counter comparison is true and the blitter has finished. The blitter has finished when the Blitter Finish flag is set. This bit should be unset with caution. It could possibly prevent some screen displays or prevent objects from being displayed correctly.

For more information about using the blitter, see Chapter 6, "Blitter Hardware".

2.8.5. The Copper and the 68000

On those occasions when the Copper's instructions do not suffice, you can interrupt the 68000 and use its instruction set instead. The 68000 can poll for interrupt flags set in the INTREQ register by various devices. To interrupt the 68000, use the copper MOVE instruction to store a 1 into the following bits of INTREQ:

BIT NUMBER	NAME	FUNCTION
15	SET/CLR	Set/Clear control bit. Determines if bits written with a 1 get set or cleared.
4	COPEN	Coprocessor interrupting 68000.

See Chapter 7, "System Control Hardware", for more information about interrupts.

2.9. SUMMARY OF COPPER INSTRUCTIONS

The table below shows a summary of the bit positions for each of the Copper instructions.

	MOVE		WAIT		SKIP	
BIT#	IR1	IR2	IR1	IR2	IR1	IR2
15	X	RD15	VP7	BFD	VP7	BFD
14	X	RD14	VP6	VE6	VP6	VE6
13	X	RD13	VP5	VE5	VP5	VE5
12	X	RD12	VP4	VE4	VP4	VE4
11	X	RD11	VP3	VE3	VP3	VE3
10	X	RD10	VP2	VE2	VP2	VE2
09	X	RD09	VP1	VE1	VP1	VE1
08	DA8	RD08	VP0	VE0	VP0	VE0
07	DA7	RD07	HP8	HE8	HP8	HE8
06	DA6	RD06	HP7	HE7	HP7	HE7
05	DA5	RD05	HP6	HE6	HP6	HE6
04	DA4	RD04	HP5	HE5	HP5	HE5
03	DA3	RD03	HP4	HE4	HP4	HE4
02	DA2	RD02	HP3	HE3	HP3	HE3
01	DA1	RD01	HP2	HE2	HP2	HE2
00	0	RD00	1	0	1	1

X = don't care, but should be a 0 for upward compatibility
IR1 = first instruction word
IR2 = second instruction word
DA = destination address
RD = RAM data to be moved to destination register
VP = vertical beam position bit
HP = horizontal beam position bit
VE = enable comparison (mask bit)
HE = enable comparison (mask bit)
BFD = blitter-finished disable

Chapter 3

PLAYFIELD HARDWARE

3.1. INTRODUCTION

The screen display of the Amiga consists of two basic parts—playfields, which are sometimes called backgrounds, and sprites, which are easily movable graphics objects. This chapter describes how to directly access hardware registers to form playfields.

About this Chapter

This chapter begins with a brief overview of playfield features, including definitions of some fundamental terms, and continues with the following major topics:

- Forming a single "basic" playfield, which is a playfield the same size as the display screen. This section includes concepts that are fundamental to forming any playfield.
- Forming a dual-playfield display, where one playfield is superimposed upon another. This differs from a basic playfield in some of the details.
- Forming playfields of various sizes, and displaying only part of a larger playfield.
- Moving playfields by scrolling them vertically and horizontally.
- Advanced topics to help you use playfields in special situations.

The following topics are relevant to playfields and are discussed elsewhere. For information about the movable sprite objects, see Chapter 4, "Sprite Hardware". There are also movable playfield objects, which are subsections of a playfield. To move portions of a playfield, you use a technique called playfield animation, which is described in Chapter 6, "Blitter Hardware".

Playfield Features

This section gives an overview of playfield features and how playfields are displayed and colored.

The Amiga produces its video displays with raster display techniques. You create a graphic display by defining one or more bit-planes in memory and filling them with 1's and 0's to determine the colors in your display. The picture you see on the screen is made up of a series of horizontal video lines displayed one after the other.

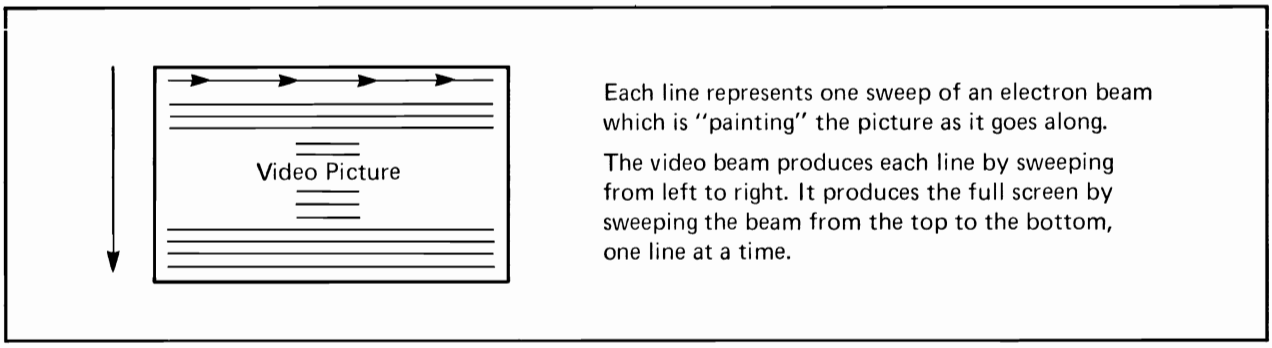


Figure 3-1: How the Video Display Picture is Produced

The video beam produces about 262 video lines from top to bottom, of which about 200 normally are visible on the screen. Each complete set of 262 lines is called a display field. A complete display field is produced in approximately 1/60th of a second; this is known as the field time. Between display fields, the video beam is traversing the lines you don't see and is returning to the top of the screen to produce another display field.

The display area is defined as a grid of pixels. A pixel is a single picture element, the smallest addressable part of a screen display. The drawings below show what a pixel is and how pixels form displays.

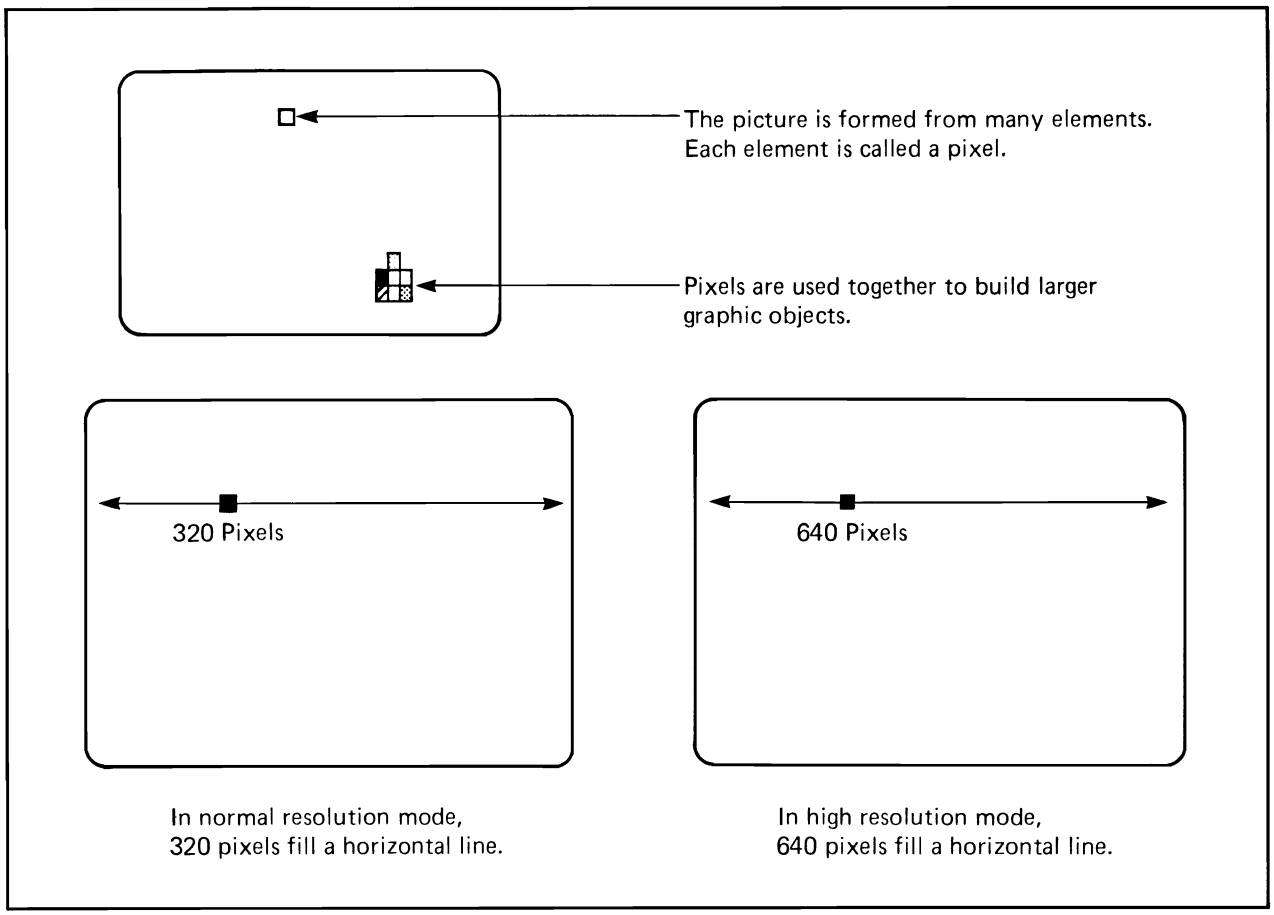


Figure 3-2: What is a Pixel

The Amiga has four basic display modes — interlace, non-interlace, low resolution, and high resolution. In non-interlace mode, the normal playfield has a height of 200 video lines. Interlace mode gives finer vertical resolution — 400 lines in the same physical display area. In low resolution mode, the normal playfield has a width of 320 pixels. High resolution mode gives finer horizontal resolution — 640 pixels in the same physical display area. These modes can be combined so you can have, for instance, an interlace, high resolution display.

Note that the dimensions referred to as "normal" in the previous paragraph are nominal dimensions and represent the normal values you should expect to use. Actually, you can display larger playfields; the maximum dimensions are given in the section called "Bit-Planes and Playfields of All Sizes". Also, the dimensions of the playfield in memory are often larger than the playfield displayed on the screen. You chose which part of this larger memory

picture to display by specifying a different size for the display window.

A playfield taller than the screen can be scrolled, or moved smoothly, up or down. A playfield wider than the screen can be scrolled horizontally, from left to right or right to left. Scrolling is described in the section called "Moving (Scrolling) Playfields".

In the Amiga graphics system, you can have up to 32 different colors in a single playfield, using normal display methods. You can control the color of each individual pixel in the playfield display by setting the bit or bits that control each pixel. A display formed in this way is called a bit-mapped display. For instance, in a 2-color display, the color of each pixel is determined by whether a single bit is on or off. If the bit is a 0, the pixel is one user-defined color; if the bit is a 1, the pixel is another color. For a 4-color display, you build two bit-planes in memory. When the playfield is displayed, the two bit-planes are overlapped, which means that each pixel is now 2 bits deep. You can combine up to 5 bit-planes in this way. Displays made up of 3, 4, or 5 bit-planes allow a choice of 8, 16, or 32 colors, respectively.

The color of a pixel is always determined by the binary combination of the bits that define it. When the system combines bit-planes for display, the combination of bits formed for each pixel corresponds to the number of a color register. This method of coloring pixels is called color indirection. The Amiga has 32 color registers, each containing bits defining a user-selected color (from a total of 4,096 possible colors).

Figure 3-3 shows how the combination of up to 5 bit-planes forms a code that selects which one of the 32 registers to use to display the color of a playfield pixel.

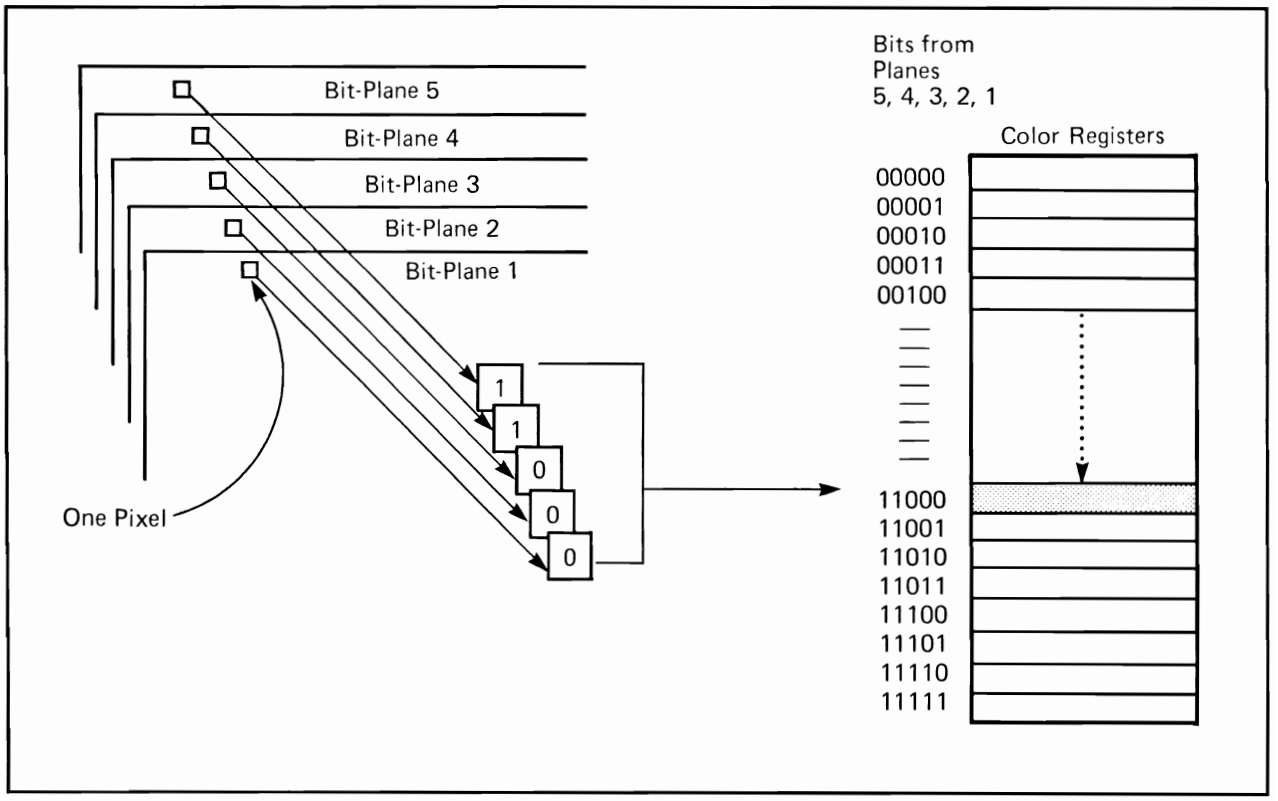


Figure 3-3: How Bit-Planes Select a Color

Values in the highest numbered bit-plane have the highest significance in the binary number. As shown in Figure 3-4, the value in each pixel in the highest-numbered bit-plane forms the leftmost digit of the number. The value in the next highest-numbered bit-plane forms the next bit, and so on.

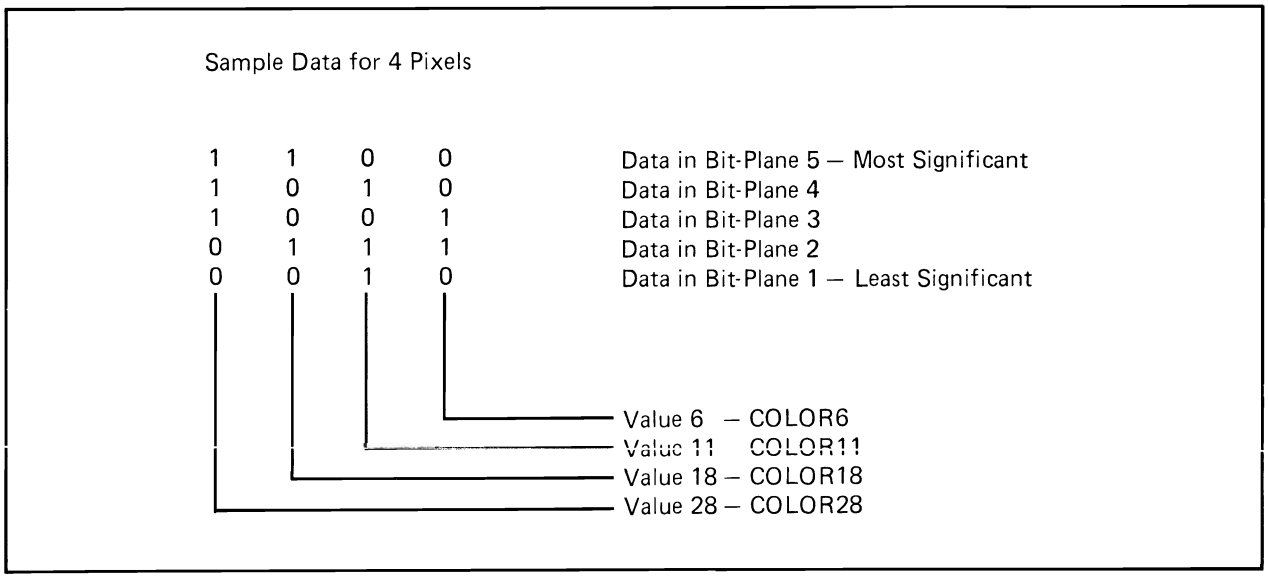


Figure 3-4: Significance of Bit-Plane Data in Selecting Colors

You also have the choice of defining 2 separate playfields, each formed from up to 3 bit-planes. Each of the two playfields uses a separate set of 8 different colors. This is called dual playfield mode.

3.2. FORMING A BASIC PLAYFIELD

To get you started, this section describes how to directly access hardware registers to form a single basic playfield that is the same size as the video screen. Here, "same size" means that the playfield is the same size as the actual display window. This will leave a small border between the playfield and the edge of the video screen. The playfield usually does not extend all the way to the edge.

To form a playfield, you need to define these characteristics:

- height and width of the playfield and size of the display window (that is, how much of the playfield actually appears on the screen)
- color of each pixel in the playfield
- horizontal resolution
- vertical resolution, or interlacing
- data fetch and modulo, which tell the system how much data to put on a horizontal line and how to fetch data from memory to the screen

In addition, you need to allocate memory to store the playfield, set pointers to tell the system where to find the data in memory, and (optionally) write a Copper routine to handle redisplay of the playfield.

All these topics are described in this section, and some of them are expanded upon in the more specialized sections that follow. At the end of this section there is a short summary of all the steps and there are two examples showing how to form a playfield that is the same size as the screen.

3.2.1. Height and Width of the Playfield

Because this is a basic playfield that is the same size as the screen, the width is either 320 pixels or 640 pixels, depending upon the resolution you choose. The height is either 200 lines or 400 lines, depending upon whether or not you choose interlace mode.

3.2.2. Bit-Planes and Color

You define playfield color by:

1. Deciding how many colors you need and how you want to color each pixel.
2. Loading the colors into the color registers.
3. Allocating memory for the number of bit-planes you need and setting a pointer to each bit-plane.
4. Writing instructions to place a value in each bit in the bit-planes to give you the correct color.

Table 3-1 shows how many bit-planes to use for the color selection you need.

Table 3-1: Colors in a Single Playfield

NUMBER OF COLORS	NUMBER OF BIT-PLANES
1 - 2	1
3 - 4	2
5 - 8	3
9 - 16	4
17 - 32	5

The Color Table

The color table contains 32 registers and you may load a different color into each of the registers. Here is a condensed view of the contents of the color table:

Table 3-2: Portion of the Color Table

REGISTER NAME	CONTENTS	MEANING
COLOR0	12 bits	User defined — color for the background area and borders.
COLOR1	12 bits	User defined color number 1. (For example, the alternate color selection for a two-color playfield.
COLOR2	12 bits	User defined color number 2.
.		
.		
.		
COLOR31	12 bits	User defined color number 31.

COLOR0 is always reserved for the background color. The background color shows in any area on the display where there is no other object present and is also displayed outside the defined display window, in the border area.

If you are using the optional genlock board for video input from a camera, VCR, or laser disk, the background color will be replaced by the incoming video display.

Twelve bits of color selection allow you to define, for each of the 32 registers, one of 4096 possible colors as shown below.

Table 3-3: Contents of the Color Registers

CONTENTS OF EACH COLOR REGISTER	
Bits 15 - 12	Unused
Bits 11 - 8	Red
Bits 7 - 4	Green
Bits 3 - 0	Blue

Here are some sample color register bit assignments and the resulting colors. At the end of the chapter is a more extensive list.

Table 3-4: Sample Color Register Contents

CONTENTS OF THE COLOR REGISTER	RESULTING COLOR
$FFF	white
$6FE	sky blue
$DB9	tan
$000	black

Some sample instructions for loading the color registers are shown below:

```
lea COLOR0, a0          ;get address of color register 0 into a0
move.w #$FFF, (a0)      ;load white into color register 0
move.w #$6FE, 2(a0)     ;load sky blue into color register 1
```

Note that the color registers are write-only. Only by looking at the screen can you find out the contents of each color register. As a standard practice, then, for these and certain other write-only registers, you may wish to keep a "backup" RAM copy. As you write to the color register itself, you should update this RAM copy. In this way, you will always know what value each register contains.

Selecting Number of Bit-Planes

After deciding how many colors you want and how many bit-planes are required to give you those colors, you tell the system how many bit-planes to use.

You select the number of bit-planes by writing the number into the register BPLCON0 (for Bit Plane Control Register 0) The relevant bits are bits 14, 13, 12, named BPU2, BPU1, BPU0 (for Bit Planes Used). The table below shows the values to write to these bits, and how the system assigns bit-plane numbers.

Table 3-5: Setting the Number of Bit-Planes

VALUE	NUMBER OF BIT-PLANES	NAME(S) OF BIT PLANES
000	None *	
001	1	PLANE 1
010	2	PLANES 1 and 2
011	3	PLANES 1 - 3
100	4	PLANES 1 - 4
101	5	PLANES 1 - 5
110	6	PLANES 1 - 6 **
111		Value not used.

* Shows only a background color; no playfield is visible.

** Sixth bit-plane is used only in dual-playfield mode and in hold and modify mode, (described in the section called "Advanced Topics").

NOTE

The bits in the BPLCON0 register are not independently settable. To set any bits, you must reload them all.

3.2.3. Selecting Horizontal and Vertical Resolution

Standard home television screens are best suited for low resolution displays. Low resolution mode provides 320 pixels for each horizontal line. High-resolution monochrome and RGB monitors can produce displays in high-resolution mode, which provides 640 pixels for each horizontal line. If you define an object in low resolution mode and then display it in high resolution mode, the object will be only half as wide.

To set horizontal resolution mode, you write to Bit 15, HIRES, in register BPLCON0:

High resolution mode — write 1 to bit 15
Low resolution mode — write 0 to bit 15

Note that in high resolution mode, you can have up to 4 bit-planes in the playfield and, therefore, up to 16 colors.

Interlacing allows you to double the number of lines appearing on the video screen. In non-interlaced mode, normally, 200 lines fill the screen and a normal size playfield appears full-size. In interlace mode, normally, a maximum of 400 lines fill the screen. Twice as much data is displayed in the same vertical area as in non-interlace mode.

In interlace mode, the scanning circuitry vertically offsets the start of every other field by 1/2 of a scan line.

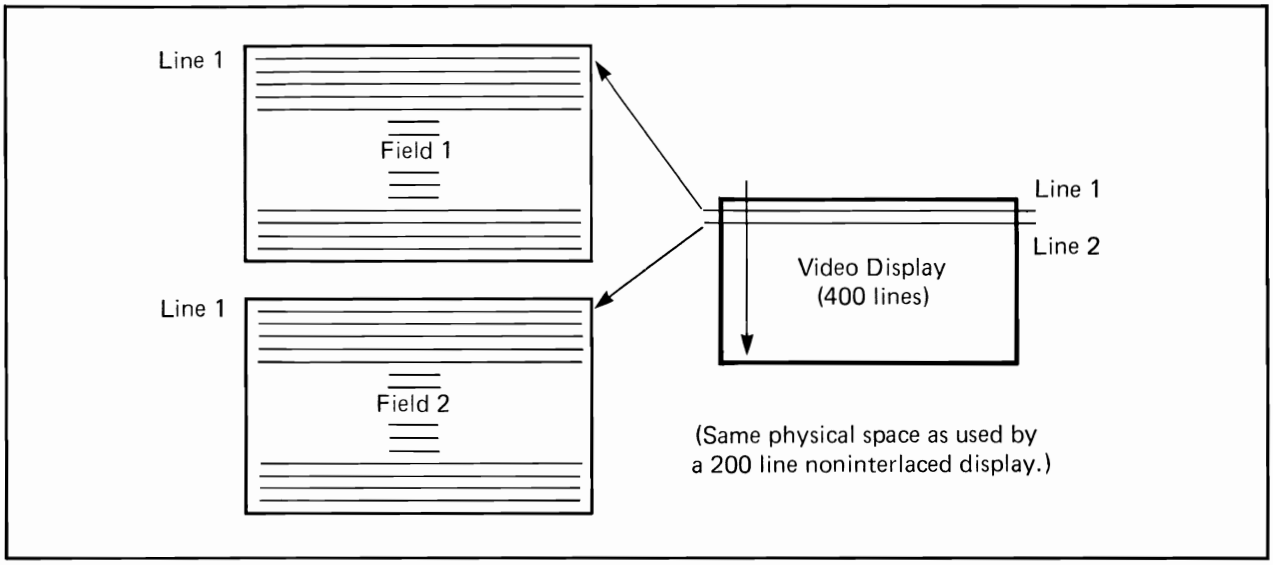


Figure 3-5: Interlacing

Even though interlace mode requires a modest amount of extra work in setting registers (as you will see later on in this section), it provides needed fine tuning for certain graphics effects. Consider the diagonal line in the picture below as it appears in non-interlaced and interlaced modes. Interlacing eliminates much of the jaggedness or "staircasing" in the edges of the line.

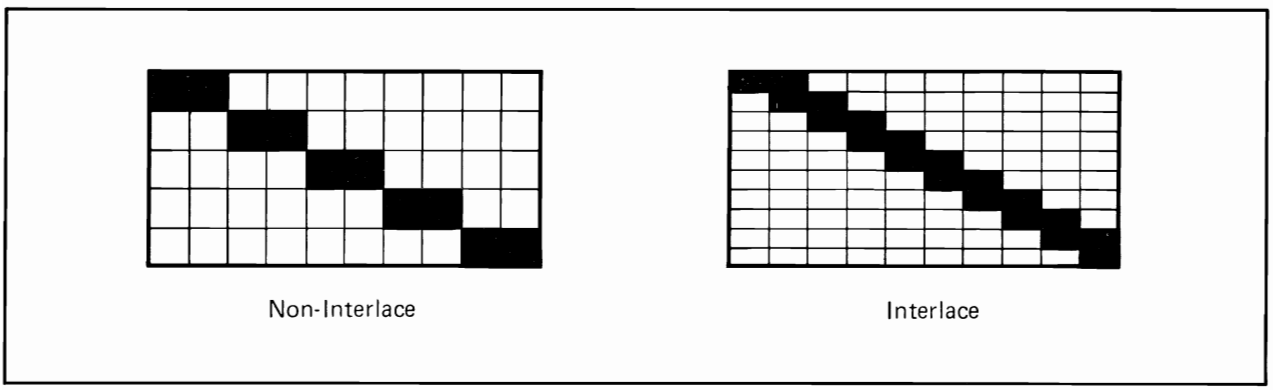


Figure 3-6: Effect of Interlace Mode on Edges of Objects

When you use the special blitter DMA channel to draw lines or polygons onto an interlaced playfield, the playfield is treated as one display, rather than as odd and even fields. Therefore, you still get the smoother edges provided by interlacing.

To set interlace or non-interlace mode, you write to bit 2, LACE, in register BPLCON0:

Interlace mode — write 1 to bit 2
Non-interlace mode — write 0 to bit 2

As explained under Section 3.2.2, bits in BPLCON0 are not independently settable.

The amount of memory you need to allocate for each bit-plane depends upon the resolution modes you have selected, as high-resolution or interlaced playfields contain more data and require larger bit-planes. The following sub-section describes memory requirements.

3.2.4. Allocating Memory for Bit-Planes

After you set the number of bit-planes and specify resolution modes, you are ready to allocate memory. A bit-plane consists of an end-to-end sequence of words at consecutive memory locations. To allocate memory, you set the registers that point to the starting memory address of each bit-plane you are using. The starting address is the memory word that contains the bits of the upper left-hand corner of the bit-plane.

Table 3-6 shows how much memory is needed for basic playfields. You may need to balance your color and resolution requirements against the amount of available memory you have.

Table 3-6: Playfield Memory Requirements

PICTURE SIZE	MODES	NUMBER OF BYTES PER BIT-PLANE
320 X 200	low-resolution, non-interlaced	8,000
320 X 400	low-resolution, interlaced	16,000
640 X 200	high-resolution, non-interlaced	16,000
640 X 400	high-resolution, interlaced	32,000

A normal low-resolution, non-interlaced display has 320 pixels across each display line and a total of 200 display lines. Each line of the bit-plane for such a display requires 40 bytes (320 bits divided by 8 bits per byte = 40).

A low-resolution, non-interlaced playfield made up of two bit-planes requires 16,000 bytes of memory area. The memory for each bit-plane must be continuous so you need to have two 8,000-byte blocks of available memory. The figure below shows an 8,000-byte memory area organized as 200 lines of 40 bytes each, providing 1 bit for each pixel position in the display plane.

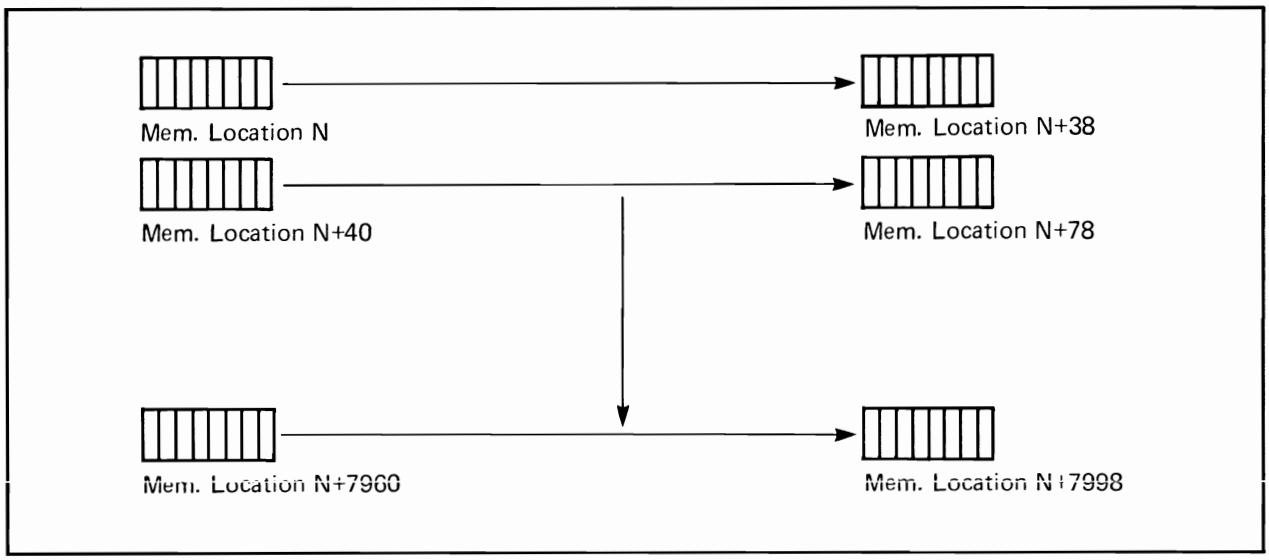


Figure 3-7: Memory Organization for a Basic Bit-Plane

Access to bit-planes in memory is provided by two address registers, BPLxPTH and BPLxPTL, for each bit-plane (12 registers in all). The "x" position in the name holds the bit-plane number; for example BPL1PTH and BPL1PTL hold the starting address of PLANE 1. As usual, pairs of registers with names ending in PTH and PTL contain 19-bit addresses. 68000 programmers may treat these as one 32-bit address and write to them as one long word. You write to the high-order word, which is the register whose name ends in "PTH".

Note that the memory requirements given here are for the playfield only. You may need to allocate additional memory for other parts of the display — sprites, audio, animation — and for your application programs. Memory allocation for other parts of the display is discussed in the chapters describing those topics.

3.2.5. Coding the Bit-Planes for Correct Coloring

After you have specified the number of bit-planes and set the bit-plane pointers, you can actually write the color register codes into the bit-planes.

A One- or Two-Color Playfield

For a one-color playfield, all you need do is write 0's in all the bits of the single bit-plane. For a two-color playfield, you define a bit-plane that has 0's where you want the background color and 1's where you want the color in register 1.

A Playfield of Three or More Colors

For 3 or more colors, you need more than one bit-plane. The task here is to define each bit-plane in such a way that, when they are combined for display, each pixel contains the correct combination of bits. This is a little more complicated than a one-bit-plane playfield. The following examples show a 4-color playfield, but the basic idea and procedures are the same for playfields containing up to 32 colors.

Figure 3-8 shows two bit-planes forming a four-color playfield:

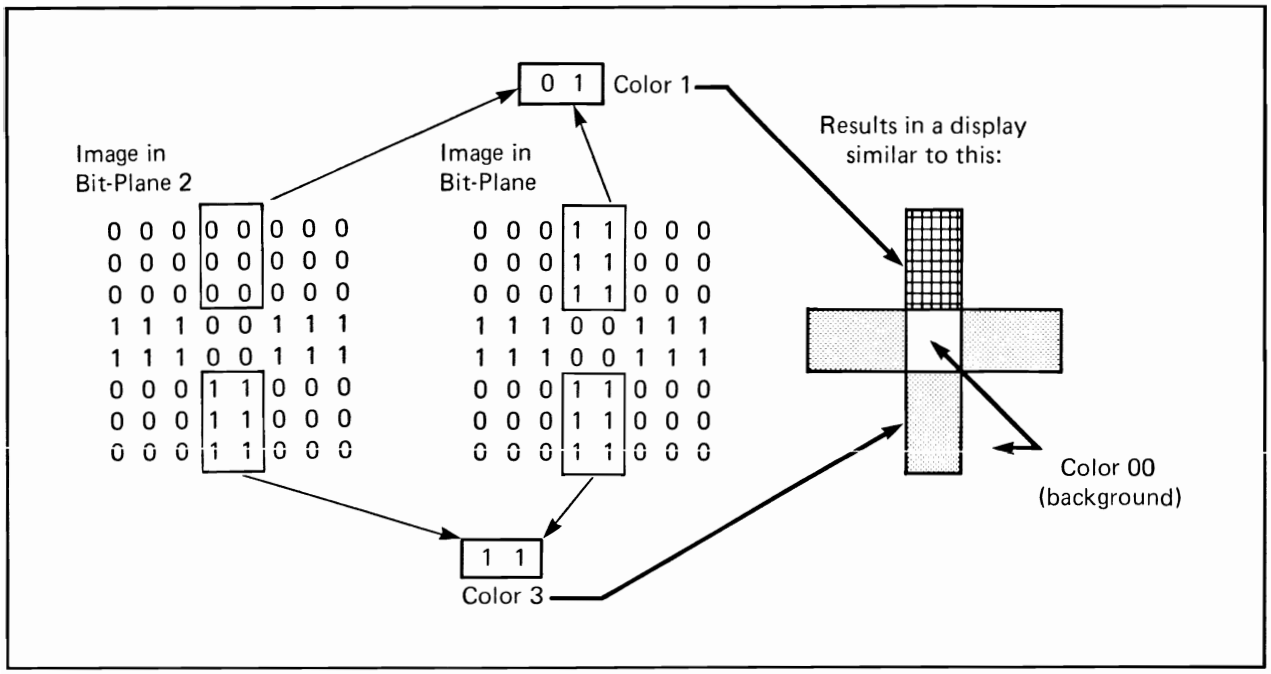


Figure 3-8: Combining Bit-Planes

You place the correct 1's and 0's in both bit-planes to give each pixel in the picture above the correct color.

In a single playfield you can combine up to 5 bit-planes in this way. Using 5 bit-planes allows a choice of 32 different colors for any single pixel. The playfield color selection charts at the end of this chapter summarize the bit combinations for playfields made from 4 and 5 bit-planes.

3.2.6. Defining the Size of the Display Window

After you have completely defined the playfield, you need to define the size of the display window. The display window is the actual size of the on-screen display. Adjustment of display window size affects the entire display area, including the border and the sprites, not just the playfield. You cannot display objects outside of the defined display window. Also, the size of the border around the playfield depends on the size of the display window.

The basic playfield described in this section is the same size as the screen display area and also the same size as the display window. This is not always the case; often the display window is smaller than the actual "big picture" of the playfield as defined in memory (the raster). A display window that is smaller than the playfield allows you to display some segment of a large playfield or scroll the playfield through the window. You can also define display windows larger than the basic playfield. These larger playfields and different-sized display windows are described in the section below called "Bit-Planes and Display Windows of All Sizes".

You define the size of the display window by specifying the vertical and horizontal positions where the window starts and stops, and writing these positions to the display window registers. The resolution of vertical start and stop is 1 scan line. The resolution of horizontal start and stop is 1 low-resolution pixel. Each position on the screen defines the horizontal and vertical position of some pixel, and this position is specified by the x and y coordinates of the pixel. This document shows the x and y coordinates in this form: (x,y). Although the coordinates begin at (0,0) in the upper left-hand corner of the screen, the first horizontal position normally used is $81 and the first vertical position is $2C. The hardware allows you to specify a starting position before ($81,$2C), but part of the display may not be visible. The difference between the absolute starting position of (0,0) and the normal starting position of ($81,$2C) is due to the way many video display monitors are designed. To overcome the distortion that can occur at the extreme edges of the screen, the scanning beam sweeps over a larger area than the front face of the screen can display. A starting position of ($81,$2C) centers a normal size display, leaving a border of 8 low-resolution pixels around the display window. Figure 3-9 shows the relationship between the normal display window, the visible screen area, and the area actually covered by the scanning beam.

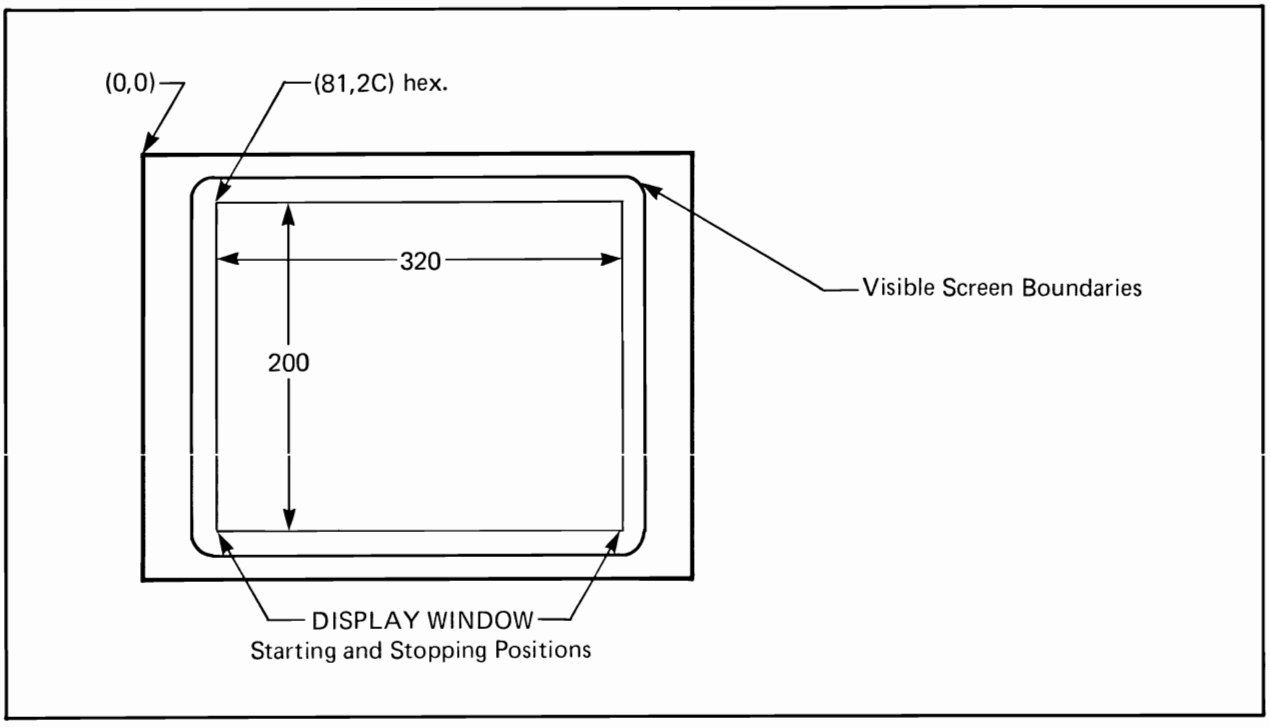


Figure 3-9: Positioning the On-Screen Display

Setting the Display Window Starting Position

A horizontal starting position of approximately $81 and a vertical starting position of approximately $2C centers the display on most standard television screens. If you select high-resolution mode (640 pixels horizontally) or interlace mode (400 lines) the starting position does not change. The starting position is always interpreted in low-resolution, non-interlace mode. In other words, you select a starting position that represents the correct coordinates in low-resolution, non-interlace mode.

The register DIWSTRT (for Display Window Start) controls the display window starting position. This register contains both the horizontal and vertical components of the display window starting positions, known respectively as HSTART and VSTART.

Setting the Display Window Stopping Position

You also need to set the display window stopping position, which is the lower right hand corner of the display window. If you select high-resolution or interlace mode, the stopping position does not change. Like the starting position, it is interpreted in low-resolution, non-interlace mode.

The register DIWSTOP (for Display Window Stop) controls the display window stopping position. This register contains both the horizontal and vertical components of the display window stopping positions, known respectively as HSTOP and VSTOP. The instructions below show how to set HSTOP and VSTOP for the basic playfield, assuming a starting position of ($81,$2C). Note that the HSTOP value you write is the actual value less 256 ($100). The HSTOP position is restricted to the right hand side of the screen. The normal HSTOP value is ($1C1) but is written as ($C1).

The VSTOP position is restricted to the lower half of the screen. This is accomplished in the hardware by forcing the MSB of the stop position to be the complement of the next MSB. This allows for a VSTOP position greater than 256 ($100) with only 8 bits. Normally, the VSTOP is set to ($F4).

The normal DIWSTART is ($2C81). The normal DIWSTOP is ($F4C1).

3.2.7. Telling the System How to Fetch and Display Data

After defining the size and position of the display window, you need to give the system the on-screen location for data fetched from memory. To do this, you describe the horizontal positions where each line starts and stops and write these positions to the data fetch registers. The data fetch registers have a 4-pixel resolution (unlike the display window registers, which have a one-pixel resolution). Each position specified is 4 pixels from the last one. Pixel 0 is position 0; pixel 4 is position 1, and so on.

Data fetch start and horizontal scrolling position interact with each other. It is recommended that data fetch start values be restricted to a programming resolution of 16 pixels (8 clocks in low resolution mode, 4 clocks in high resolution mode). The hardware requires some time after the first data fetch before it can actually display the data. As a result, there is a difference between the value of window start and data fetch start. In low-resolution mode the difference is 8.5 clocks; in high-resolution mode the difference is 4.5 clocks.

The normal low-resolution DDFSTART is ($0038). The normal high-resolution DDFSTART is ($003C). Recall that the hardware resolution of display window start and stop is twice the hardware resolution of data fetch:

($81/2 - 8.5) = ($38)
($81/2 - 4.5) = ($3C)

The relationship between data fetch start and stop is:

DDFSTART = DDFSTOP - (8*(word count - 1) for low-resolution
DDFSTART = DDFSTOP - (4*(word count - 2) for high-resolution

The normal low-resolution DDFSTOP is ($00D0). The normal high-resolution DDFSTOP is ($00D4).

You also need to tell the system exactly which bytes in memory belong on each horizontal line of the display. To do this, you specify the modulo value. Modulo refers to the number of bytes in memory between the last word on one horizontal line and the beginning of the first word on the next line. Thus, the modulo enables the system to convert bit-plane data stored in linear form (each data byte at a sequentially increasing memory address) into rectangular form (one "line" of sequential data followed by another line). For the basic playfield, where the playfield in memory is the same size as the display window, the modulo is zero because the memory area contains exactly the same number of bytes as you want to display on the screen. Figures 3-10 and 3-11 show the basic bit-plane layout in memory and how to make sure the correct data is retrieved.

The bit-plane address pointers (BPLxPTH and BPLxPTL) are used by the system to fetch the data to the screen. These pointers are dynamic; once the data fetch begins, the pointers are continuously incremented to point to the next word to be fetched (data is fetched two bytes at a time). When the end-of-line condition is reached (defined by the Data Fetch register, DDFSTOP) the modulo is added to the bit-plane pointers, adjusting the pointer to the first word to be fetched for the next horizontal line.

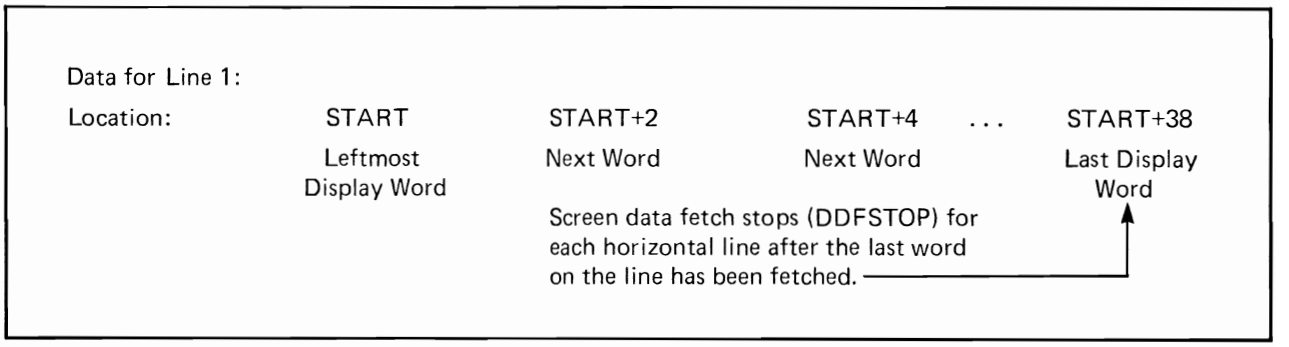


Figure 3-10: Data Fetched for the First Line When Modulo = 0

After the first line is fetched, the bit-plane pointers BPLxPTH and BPLxPTL contain the value START+40. The modulo (in this case, 0) is added to the current value of the pointer so when the pointer begins the data fetch for the next line, it fetches the data you want on that line. The data for the next line begins at memory location START+40.

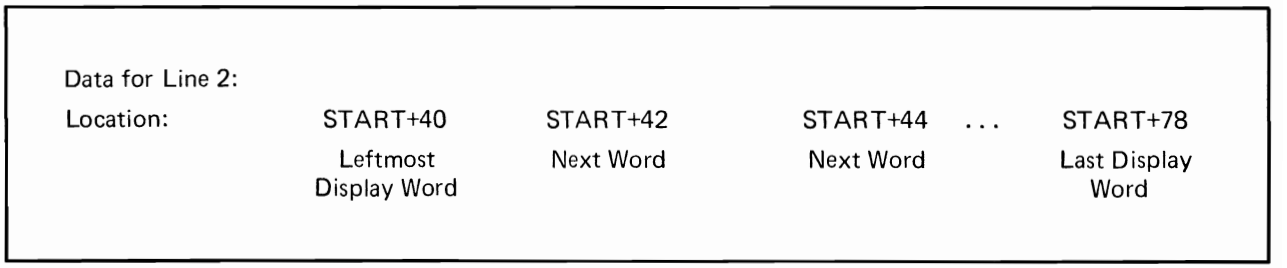


Figure 3-11: Data Fetched for the Second Line When Modulo = 0

Note that the pointers always contain an even number, because data is fetched from display a word at a time.

There are two modulo registers—BPL1MOD for the odd-numbered bit-planes and BPL2MOD for the even-numbered bit-planes.

Data Fetch in High Resolution Mode

When you are using high-resolution mode to display the basic playfield, you need to fetch 80 bytes for each line, instead of 40.

Modulo in Interlace Mode

For interlace mode, you must redefine the modulo. This is because there are two separate scannings of the video screen for a single display of the playfield. During the first scanning the odd-numbered lines are fetched to the screen, and during the second scanning the even-numbered lines are fetched.

The bit-planes for a full-screen-size, interlaced display are 400 lines long, instead of 200 lines. Assuming that the playfield in memory is the normal 320 pixels wide, then data for the interlaced picture begins at the following locations. These are all byte addresses.

Line 1	START
Line 2	START+40
Line 3	START+80
Line 4	START+120

and so on. Therefore, you use a modulo of 40 to skip the lines in the other field. For odd fields, the bit-plane pointers begin at START. For even fields, the bit-plane pointers begin at START+40.

You can use the Copper to handle resetting of the bit-plane pointers for interlaced displays.

3.2.8. Displaying and Redisplaying the Playfield

You start playfield display by making certain that the bit-plane pointers are set and bit-plane DMA is turned on. You turn on bit-plane DMA by writing a 1 to bit BPLEN in the DMACON (for DMA control) register. See Chapter 7, "System Control Hardware", for instructions on setting this register.

Each time the playfield is redisplayed, you have to reset the bit-plane pointers. Resetting is necessary because the pointers have been incremented to point to each successive word in memory and must be repointed to the first word for the next display. You write Copper instructions to handle the redisplay or perform this operation as part of a vertical blanking task.

3.2.9. Enabling the Color Display

To enable color, rather than black and white display, you need to set Bit 9 in BPLCON0. This enables the color burst signal on composite video. This does not affect RGB video.

3.2.10. Summary

The steps for defining a basic playfield are summarized here.

Define Playfield Characteristics

Specify height in lines

- 200 maximum for non-interlaced mode
- 400 maximum for interlaced mode

Specify width in pixels

- 320 maximum for low-resolution mode
- 640 maximum for high-resolution mode

Specify color for each pixel

- Load desired colors in color table registers.
- Define color of each pixel in terms of the binary value that points at the desired color register.
- Build bit-planes
- Set bit-plane registers:
    - Bits 12-14 in BPLCON0 - number of bit-planes (BPU2 - BPU0)

    * BPLxPTH - pointer bit-plane starting position in memory (written as a long word)

Specify resolution

- Low resolution:
    * 320 pixels in each horizontal line
    * Clear bit 15 in register BPLCON0 (HIRES)
- High resolution:
    * 640 pixels in each horizontal line
    * Set bit 15 in register BPLCON0 (HIRES)

Specify interlace or non-interlace

- Interlace mode
    * 400 vertical lines
    * Set bit 2 in register BPLCON0 (LACE)
- Non-interlace mode
    * 200 vertical lines
    * Clear bit 2 in BPLCON0 (LACE)

Allocate Memory

To calculate data-bytes in the total bit-planes, use the formula:

bytes per line * lines in playfield * number of bit-planes

Define Size of Display Window

Write start position of display window in DIWSTRT

- Horizontal position in bits 0 through 7 (low-order bits)
- Vertical position in bits 8 through 15 (high-order bits)

Write stop position of display window in DIWSTOP

- Horizontal position in bits 0 through 7
- Vertical position in bits 8 through 15

Define Data Fetch

Set registers DDFSTRT and DDFSTOP

- For DDFSTRT, use the horizontal position as shown in the DDFSTRT section.
- For DDFSTOP, use the horizontal position as shown in the DDFSTOP section.

Define Modulo

Set registers BPL1MOD and BPL2MOD

- Set modulo to 0 for non-interlace, 40 for interlace.

Write Copper Instructions

- To handle redisplay

Enable Color Display

- Set bit 9 in BPLCON0 to enable the color display on a composite video monitor. RGB video is not affected.

3.3. FORMING A DUAL PLAYFIELD DISPLAY

For more flexibility in designing your background display, you can specify two playfields instead of one. In dual playfield mode, one playfield is displayed directly in front of the other. For example, a computer game display might have some action going on in one playfield in the background, while the other playfield is showing a control panel in the foreground. You could then change either the foreground or the background without having to redesign the entire display. You can also move the two playfields independently.

A dual playfield display is similar to a single playfield display, differing only in these aspects:

- Each playfield in a dual display is formed from 1, 2 or 3 bit-planes.
- The colors in each playfield (up to 7 plus transparent) are taken from different sets of color registers.
- You must set a bit to activate dual playfield mode.

Figure 3-12 shows a dual playfield display.

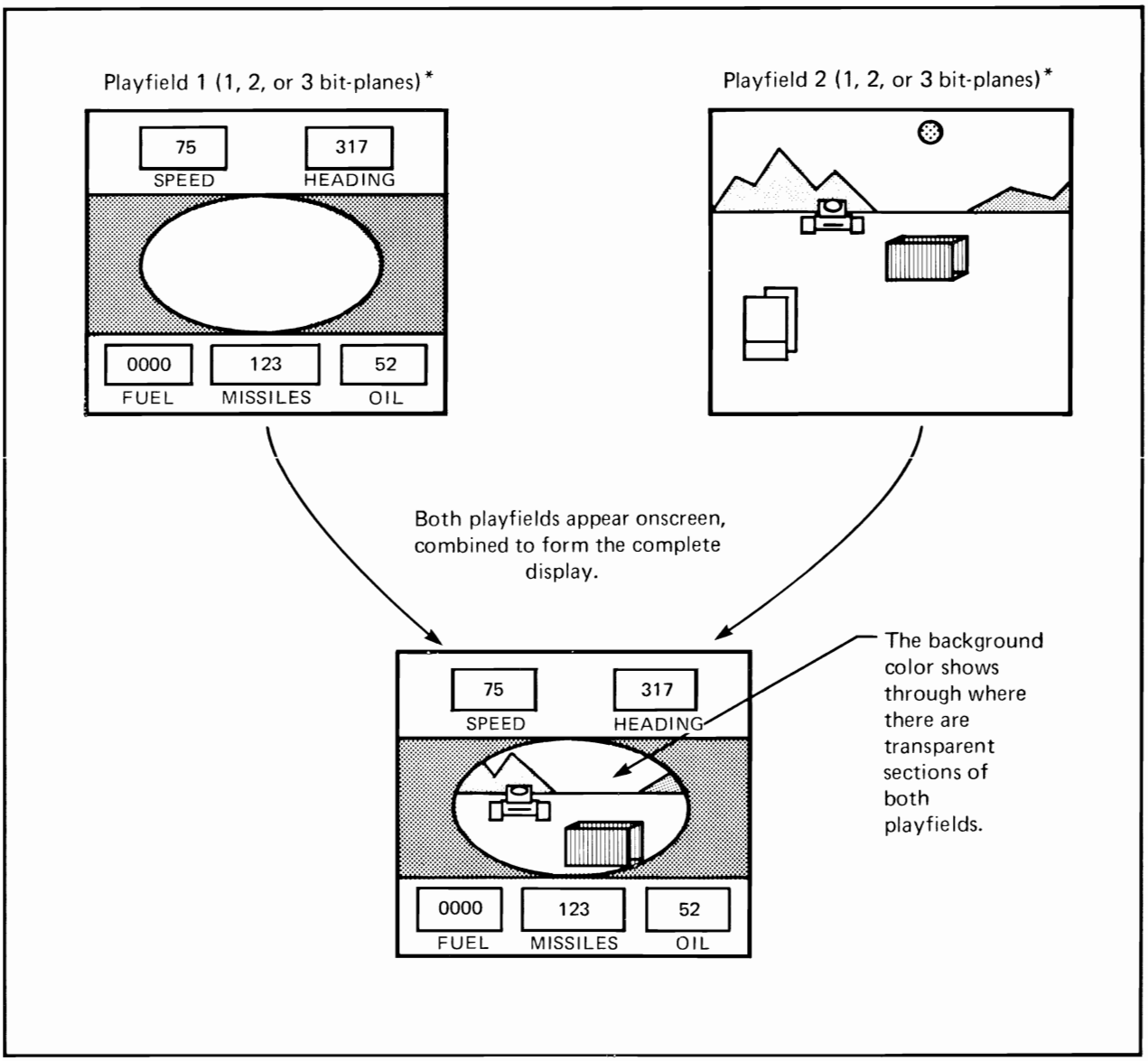


Figure 3-12: A Dual Playfield Display

The sketch also shows that one of the colors in the playfield is "transparent" (color 0 in playfield 1 and color 8 in playfield 2). You can use transparency to allow selected features of the background playfield to show through.

In dual playfield mode, each playfield is formed from up to 3 bit-planes. Color registers 0 through 7 are assigned to Playfield 1, depending upon how many bit-planes you use. Color registers 8 through 15 are assigned to Playfield 2.

3.3.1. How Bit-Planes are Assigned in Dual Playfield Mode

The 3 odd-numbered bit-planes (1, 3 and 5) are grouped together by the hardware and may be used in Playfield 1. Likewise, the 3 even-numbered bit-planes (2, 4, and 6) are grouped together and may be used in Playfield 2. The bit-planes are assigned alternately to each playfield as shown in the figure below. Note that in high-resolution mode, you can have up to 2 bit-planes in each playfield — bit-planes 1, 3 in Playfield 1 and bit-planes 2, 4 in Playfield 2.

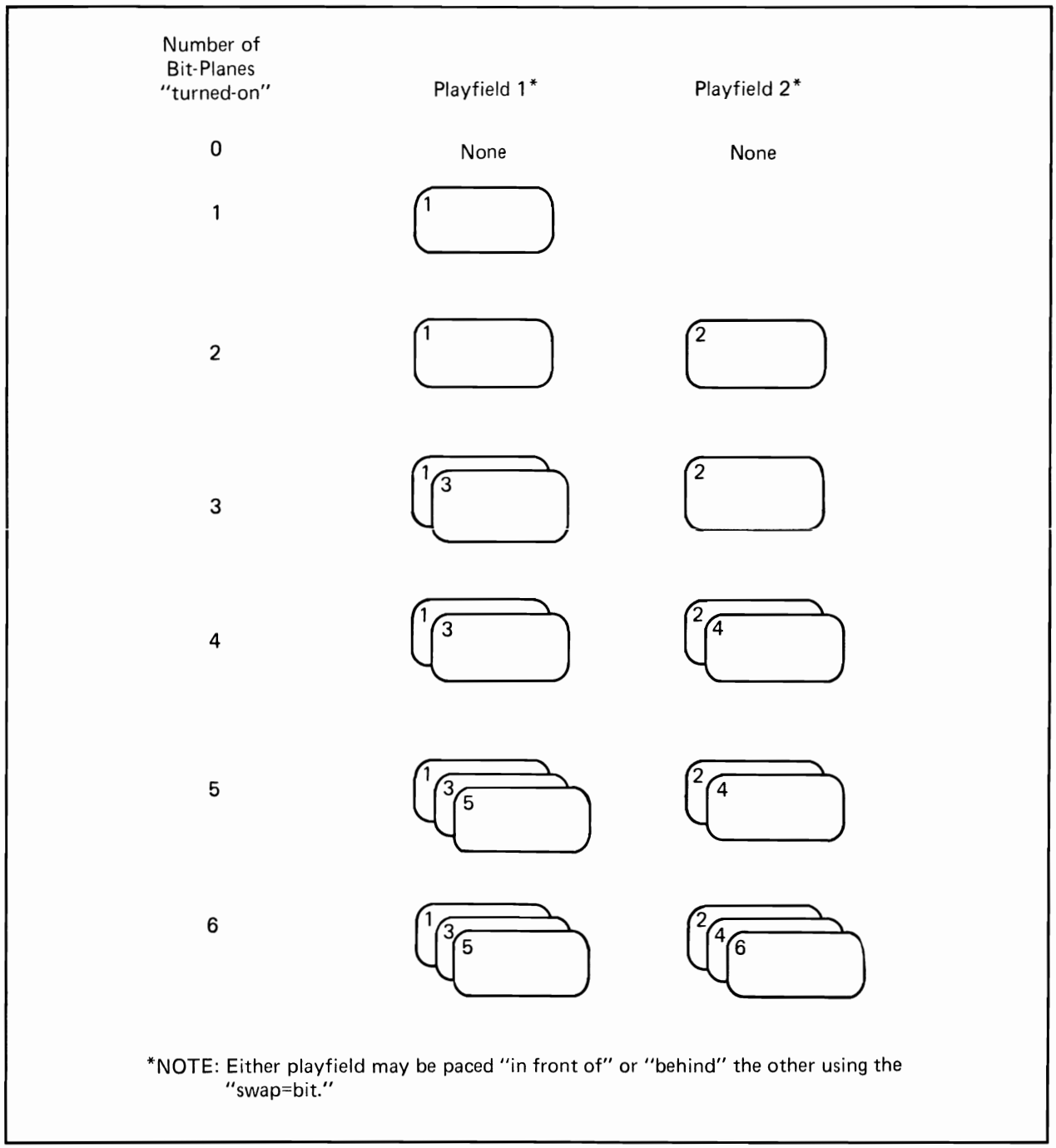


Figure 3-13: How Bit-Planes are Assigned to Dual Playfields

3.3.2. How Color Registers are Assigned in Dual Playfield Mode

When you are using dual playfields, the hardware interprets color numbers for Playfield 1 from the bit combinations of bit-planes 1, 3, and 5. Bits from PLANE 5 have the highest significance, and form the most significant digit of the color register number. Bits from PLANE 0 have the lowest significance. These bit combinations select the first 8 color registers from the color palette as shown in Table 3-7.

Table 3-7: Playfield 1 Color Registers — Low Resolution Mode

PLAYFIELD 1	
BIT COMBINATION	SELECTS
000	Transparent Mode
001	COLOR1
010	COLOR2
011	COLOR3
100	COLOR4
101	COLOR5
110	COLOR6
111	COLOR7

The hardware interprets color numbers for Playfield 2 from the bit combinations of bit-planes 2, 4, and 6. Bits from PLANE 6 have the highest significance. Bits from PLANE 2 have the lowest significance. These bit combinations select the color registers from the second 8 colors in the color table as shown in Table 3-8.

Table 3-8: Playfield 2 Color Registers — Low Resolution Mode

PLAYFIELD 2	
BIT COMBINATION	SELECTS
000	Transparent Mode
001	COLOR9
010	COLOR10
011	COLOR11
100	COLOR12
101	COLOR13
110	COLOR14
111	COLOR15

Combination 000 selects transparent mode, to show the color of whatever object (the other playfield, a sprite, or the background color) may be "behind" the playfield.

The following tables show the color registers for high resolution, dual playfield mode.

Table 3-9: Playfield 1 and 2 Color Registers — High Resolution Mode

PLAYFIELD 1

BIT COMBINATION	SELECTS
00	Transparent Mode
01	COLOR1
10	COLOR2
11	COLOR3

PLAYFIELD 2

BIT COMBINATION	SELECTS
00	Transparent Mode
01	COLOR9
10	COLOR10
11	COLOR11

3.3.3. Dual Playfield Priority and Control

Either Playfield 1 or 2 may have priority; that is, may be displayed in front of the other. Normally Playfield 1 has priority. You use the bit known as PF2PRI (Bit 6) in register BPLCON2 to control priority. When PF2PRI = 1, Playfield 2 has priority over Playfield 1. When PF2PRI = 0, Playfield 1 has priority.

You can also control the relative priority of playfields and sprites. Chapter 7, "System Control Hardware", shows you how to control the priority of these objects.

You can separately control the two playfields as follows:

- They can have different sized representations in memory and different portions of each one can be selected for display.

- They can be scrolled separately.

  NOTE: You must take special care when scrolling one playfield and holding the other stationary. When you are scrolling low resolution playfields, you must fetch one word more than the width of the playfield you are trying to scroll (two words more in high-resolution mode) in order to provide some data to display when the actual scrolling takes place. There is only one data fetch start and one data fetch stop register, and this register is shared by both playfields. If you want to scroll one playfield and hold the other, you must still adjust the data fetch start and data fetch stop to handle the one being scrolled. Therefore, you have to adjust the modulo and the bit-plane pointers of the playfield that is not being scrolled to maintain its position on the display. In low resolution mode, you adjust the pointers by

-2 and the modulo by -2. In high resolution mode, you adjust the pointers by -4 and the modulo by -4.

3.3.4. Activating Dual Playfield Mode

Writing a 1 to bit 10 (called DBLPF) of the Bit Plane Control Register BPLCON0 selects dual playfield mode. Selecting dual playfield mode changes:

- the way hardware groups the bit-planes for color interpretation; that is, all odd-numbered bit-planes are grouped together and all even-numbered bit-planes are grouped together.
- the way hardware can move the bit-planes on the screen.

3.3.5. Summary

The steps for defining dual playfields are almost the same as for the basic playfield. These steps are somewhat different:

- **Loading colors into the registers**. Keep in mind that color registers 0-7 are used by Playfield 1 and registers 8 through 15 are used by Playfield 2 (if there are 3 bit-planes in each playfield).
- **Building bit-planes**. Recall that Playfield 1 is formed from PLANES 1, 3 and 5 and Playfield 2 from PLANES 2, 4, and 6.
- **Setting the modulo registers**. Write the modulo to both BPL1MOD and BPL2MOD as you will be using both odd- and even-numbered bit-planes.

These steps are added:

- **Defining Priority**. If you want Playfield 2 to have priority, set Bit 6 (PF2PRI) in BPLCON2 to 1.
- **Activating Dual Playfield Mode**. Set bit 10 (DBLPF) in BPLCON0 to 1.

3.4. BIT-PLANES AND DISPLAY WINDOWS OF ALL SIZES

You have seen how to form single and dual playfields where the playfield in memory is the same size as the display window. This section shows you:

- How to define and use a playfield whose big picture in memory is larger than the display window.
- How to define display windows larger or smaller than the normal playfield size.
- How to move the display window in the big picture.

3.4.1. When the Big Picture is Larger than the Display Window

If you design a memory picture larger than the display window, then you must chose which part of it to display. Displaying a portion of a larger playfield differs in the following ways from the basic playfields described up to now:

- If the big picture in memory is larger than the display window, you respecify the modulo. The modulo must be some value other than 0.
- You must allocate more memory for the larger memory picture.

Specifying the Modulo

For a memory picture wider than the display window, you need to respecify the modulo so the correct data words are fetched for each line of the display. As an example, assume the display window is the standard 320 pixels wide, so 40 bytes are to be displayed on each line. The big picture in memory, however, is exactly twice as wide as the display window, or 80 bytes wide. Also, assume that you wish to display the left half of the big picture. Figure 3-14 shows the relationship between the big picture and the picture to be displayed.

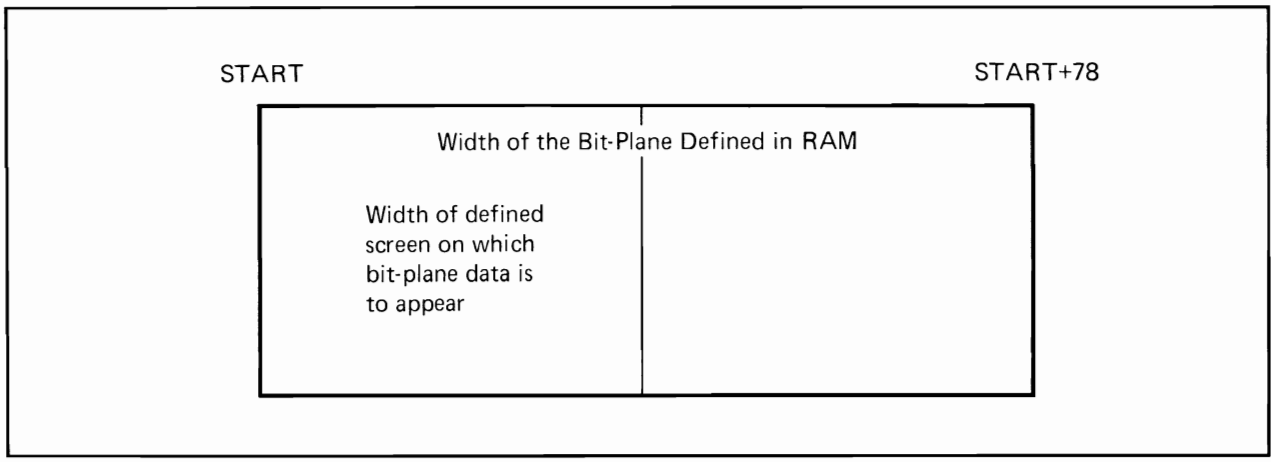


Figure 3-14: Memory Picture Larger than the Display

Because 40 bytes are to be fetched for each line, the data fetch for line 1 is as shown in Figure 3-15.

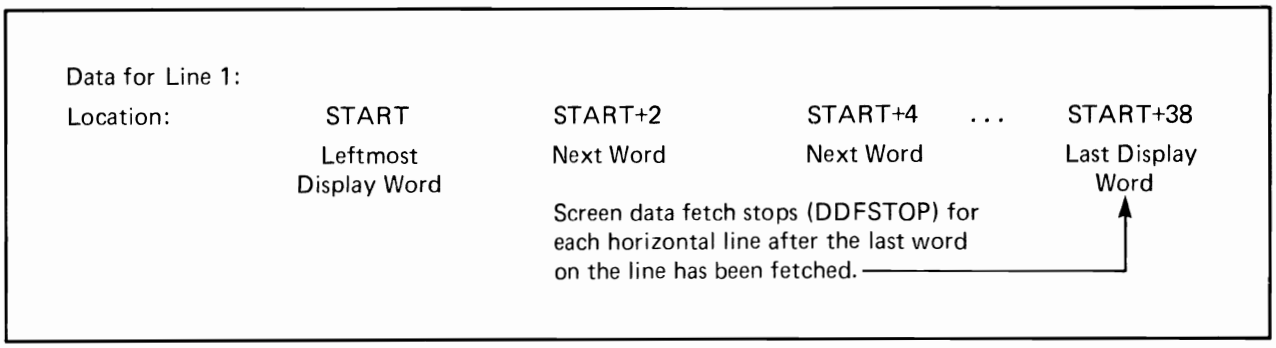


Figure 3-15: Data Fetch for the First Line when Modulo = 40

At this point, BPLxPTH and BPLxPTL contain the value START+40. The modulo, which is 40, is added to the current value of the pointer so that when it begins the data fetch for the next line, it fetches the data that you intend for that line. The data fetch for line 2 is shown in Figure 3-16.

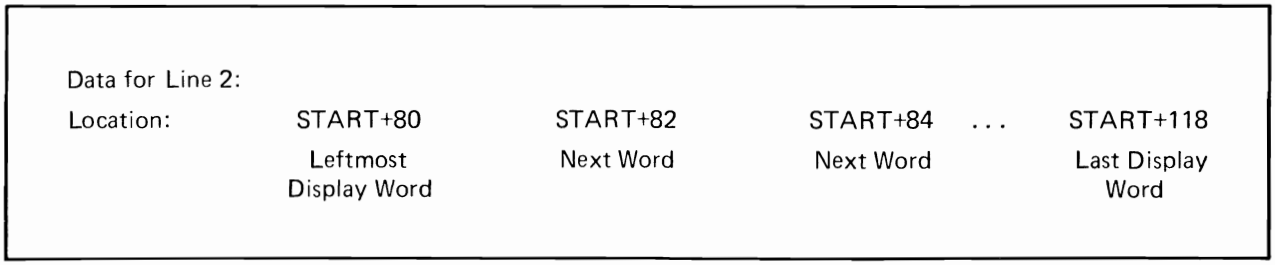


Figure 3-16: Data Fetch for the Second Line when Modulo = 40

To display the right half of the big picture, you set up a vertical blanking routine to start the bit-plane pointers at location START+40 rather than START; with the modulo remaining at 40. The data layout is shown in Figures 3-17 and 3-18.

Data for Line 2:				
Location:	START+40	START+42	START+44 ...	START+78
	Leftmost Display Word	Next Word	Next Word	Last Display Word

Figure 3-17: Data Layout for First Line - Right Half of Big Picture

Now, the bit-plane pointers contain the value START+80. The modulo (40) is added to the pointers so when they begin the data fetch for the second line, the correct data is fetched.

Data for Line 2:				
Location:	START+120	START+122	START+124 ...	START+158
	Leftmost Display Word	Next Word	Next Word	Last Display Word

Figure 3-18: Data Layout for Second Line - Right Half of Big Picture

Remember, in high resolution mode, you need to fetch twice as many bytes as in low resolution mode. For a normal size display, you fetch 80 bytes for each horizontal line instead of 40.

Specifying the Data Fetch

The data fetch registers specify the beginning and end positions for data placement on each horizontal line of the display. You specify data fetch in the same way as shown in the section called "Forming a Basic Playfield".

Memory Allocation

For larger memory pictures, you need to allocate more memory. Here is a formula for calculating memory requirements in general:

bytes per line * lines in playfield * # of bit-planes

Thus if the wide playfield described in this section is formed from two bit-planes, it requires:

80 * 200 * 2 = 32,000 bytes of memory

Recall that this is the memory requirement for the playfield alone. You need more memory for any sprites, animation, audio, or application programs you are using.

Selecting the Display Window Starting Position

The display window starting position is the horizontal and vertical coordinates of the upper left-hand corner of the display window. One register, DIWSTRT, holds both the horizontal and vertical coordinates, known as HSTART and VSTART, respectively. The 8 bits allocated to HSTART are assigned to the first 256 positions counting from the leftmost possible position. Thus, you can start the display window at any pixel position within this range.

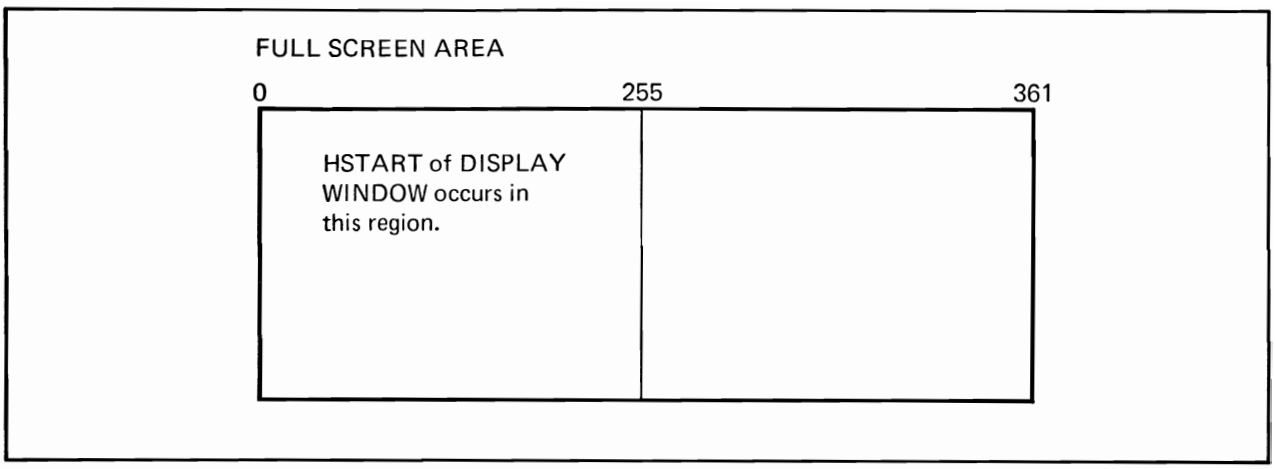


Figure 3-19: Display Window Horizontal Starting Position

The 8 bits allocated to VSTART are assigned to the first 256 positions counting down from the top of the display.

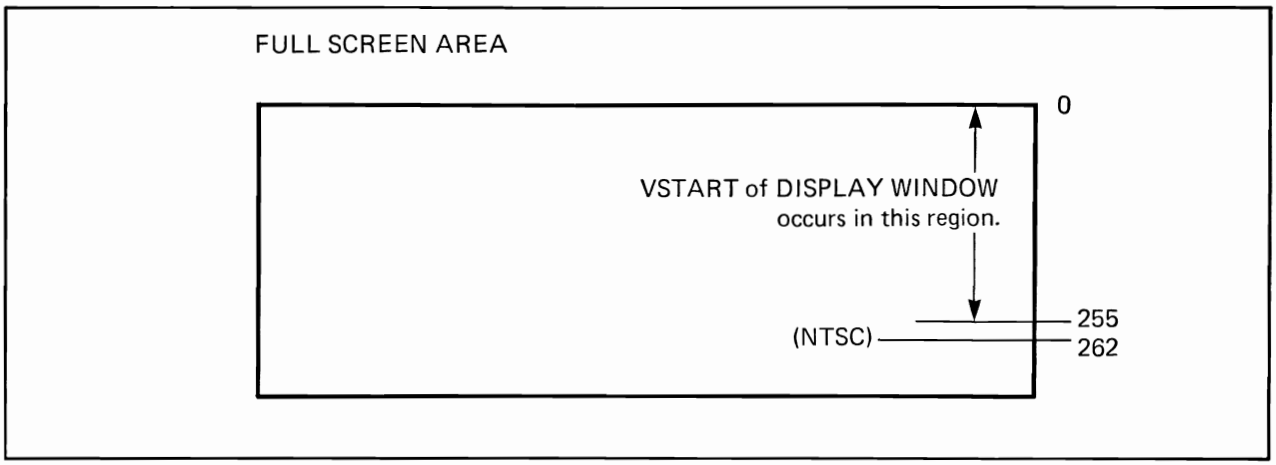


Figure 3-20: Display Window Vertical Starting Position

Recall that you select the values for the starting position as if the display were in low-resolution, non-interlace mode. Keep in mind, though, that for interlace mode the display window should be an even number of lines in height to allow for equal-size odd and even fields.

To set the display window starting position, write the value for HSTART into bits 0 through 7 and the value for VSTART into bits 8 through 15 of DIWSTRT.

Selecting the Stopping Position

Stopping position for the display window is the horizontal and vertical coordinates of the lower right-hand corner of the display window. One register, DIWSTOP, contains both coordinates, known as HSTOP and VSTOP, respectively.

See the notes in the "Forming a Basic Playfield" section for how to set these registers.

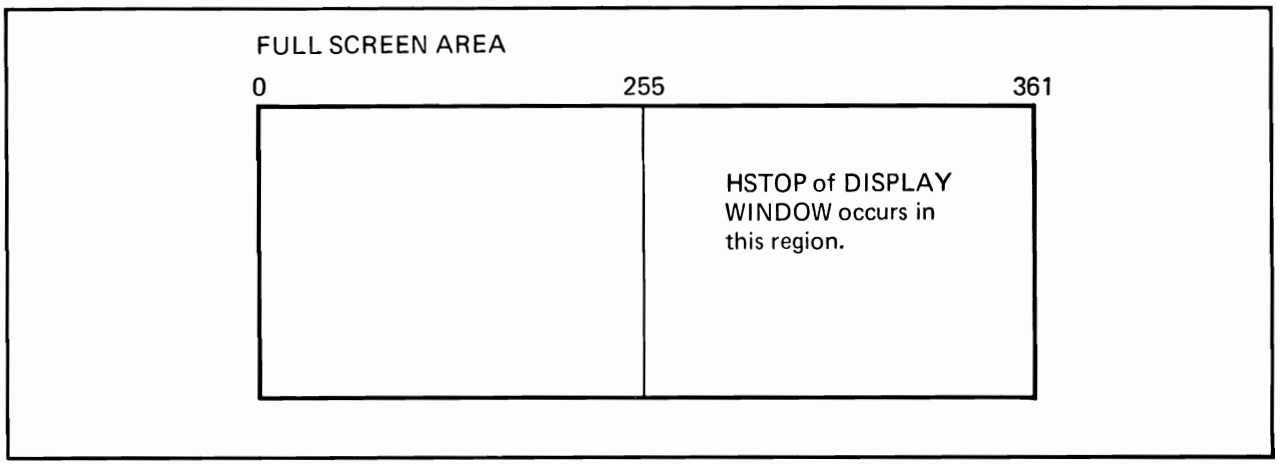


Figure 3-21: Display Window Horizontal Stopping Position

Select a value that represents the correct position in low resolution, non-interlaced mode.

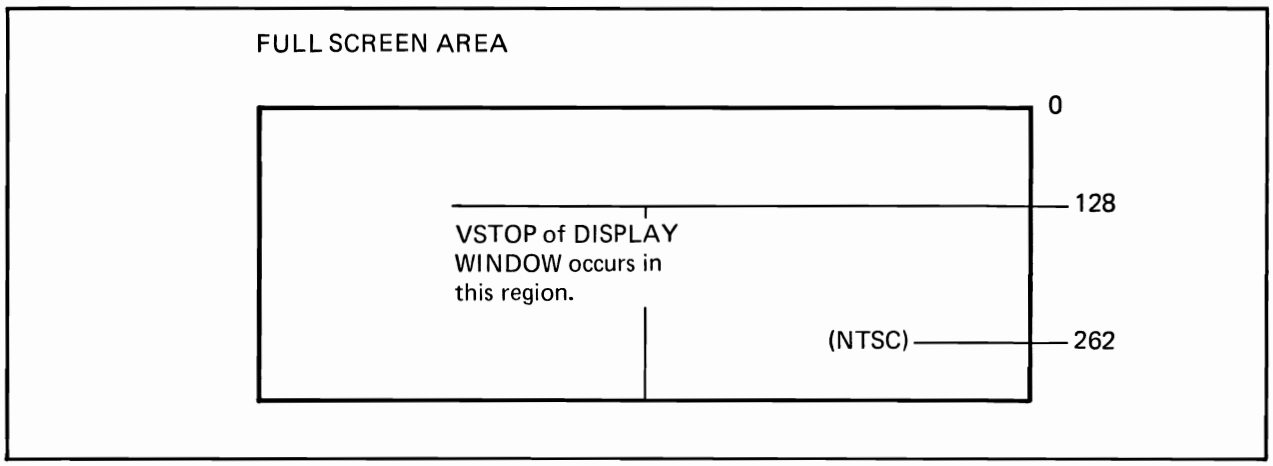


Figure 3-22: Display Window Vertical Stopping Position

To set the display window stopping position, write HSTOP into bits 0 through 7 and VSTOP into bits 8 through 15 of DIWSTOP.

3.4.2. Maximum Display Window Size

The maximum size of a playfield display is determined by the maximum number of lines and the maximum number of columns. Vertically, the restrictions are simple. There cannot be any displayed data in vertical blanking which has a range of line 0 through line 19 (20 lines total). This leaves 242 lines of displayable screen video (interlace doubles this to 484).

Horizontally, the situation is similar. Strictly speaking, the hardware sets a rightmost limit to DDFSTOP of ($D8) and a leftmost limit to DDFSTRT of ($18). This gives a maximum of 25 words fetched in low-resolution. In high resolution, the maximum here is 49 words because the rightmost limit remains ($D8) and only one word is fetched at this limit. However, horizontal blanking actually limits the displayable video to 376 low-resolution pixels (23.5 words). In addition, it should be noted that using a data fetch start earlier than ($38) will disable some sprites.

3.5. MOVING (SCROLLING) PLAYFIELDS

If you want a moving background display, you can design a playfield larger than the display window and scroll it. If you are using dual playfields, you can scroll them separately.

In vertical scrolling, the playfield appears to move smoothly up or down on the screen. All you need do for vertical scrolling is progressively increase or decrease the starting address for the bit-plane pointers by the size of a horizontal line in the playfield. This has the effect of showing a lower or higher part of the picture each field time.

In horizontal scrolling the playfield appears to move from right to left or left to right on the screen. Horizontal scrolling works differently from vertical scrolling — you must arrange to fetch one more word of data for each display line and delay the display of this data.

For either type of scrolling, resetting of pointers or data fetch registers can be handled during the vertical blanking interval by the Copper.

3.5.1. Vertical Scrolling

You can scroll a playfield upward or downward in the window. Each time you display the playfield, the bit-plane pointers start at a progressively higher or lower place in the big picture in memory. As the value of the pointer increases, more of the lower part of the picture is shown and the picture appears to scroll upward. As the value of the pointer decreases, more of the upper part is shown and the picture scrolls downward. If your picture has 200 vertical lines, each step can be as little as 1/200th of the screen. In interlace mode each step could be 1/400th of the screen if clever manipulation of the pointers is used, but it is recommended that scrolling be done 2 lines at a time to maintain the odd/even field relationship.

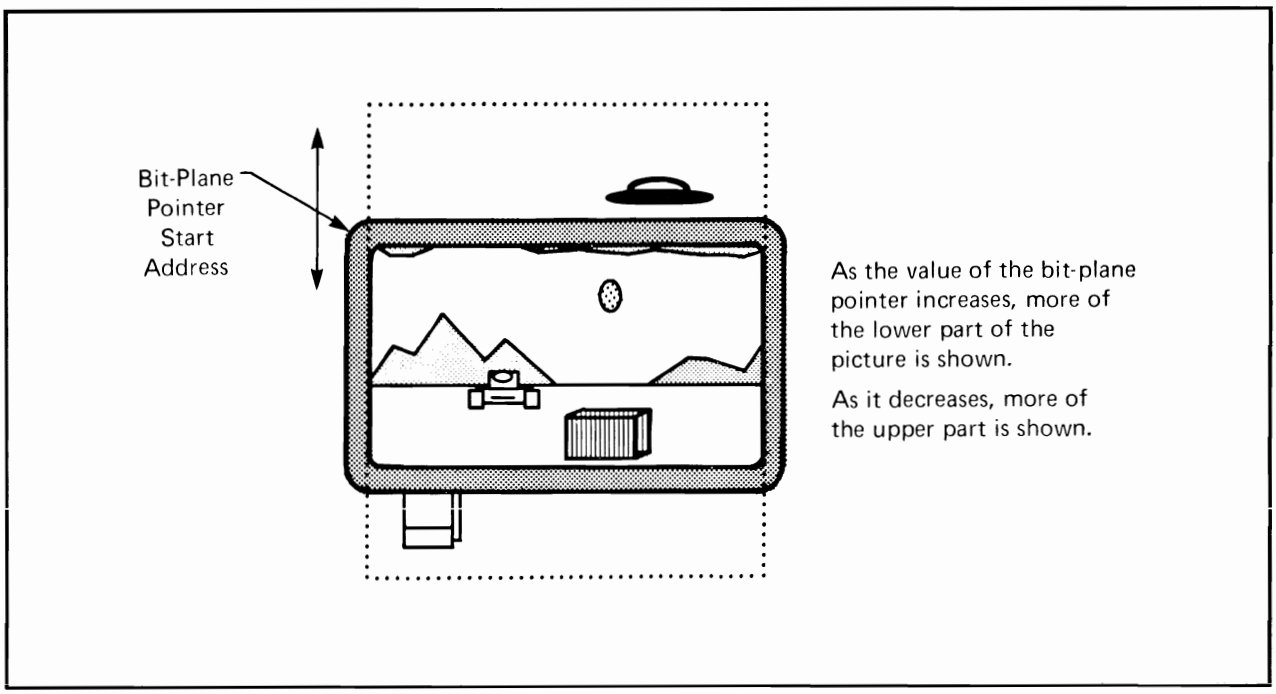


Figure 3-23: Vertical Scrolling

To set up a playfield for vertical scrolling, you need to

- Form bit-planes tall enough to allow for the amount of scrolling you want.
- Write software to calculate the bit-plane pointers for the scrolling you want, and allow for the Copper to use the resultant pointers.

Assume you wish to scroll a playfield upward one line at a time. To accomplish this, before each field is displayed, the bit-plane pointers have to increase by enough to ensure that the pointers begin one line lower each time. For a normal size low resolution display where the modulo is 0, the pointers would be incremented by 40 bytes each time.

3.5.2. Horizontal Scrolling

You can scroll playfields horizontally from left to right or right to left on the screen. You control the speed of scrolling by specifying the amount of delay in pixels. Delay means that a extra word of data is fetched but not immediately displayed. The extra word is placed just to the left of the window's leftmost edge and before normal data fetch. As the display shifts to the right, the bits in this extra word appear on-screen at the left-hand side of the window as bits on the right-hand side disappear off-screen. For each pixel of delay, the on-screen data shifts one pixel to the right each display field. The greater the delay, the greater the speed of scrolling. You can have up to 15 pixels of delay. In high-resolution mode, scrolling is in increments of 2 pixels.

Note that fetching an extra word for scrolling will disable some sprites.

Figure 3-24 shows how the delay and extra data fetch combine to cause the scrolling effect.

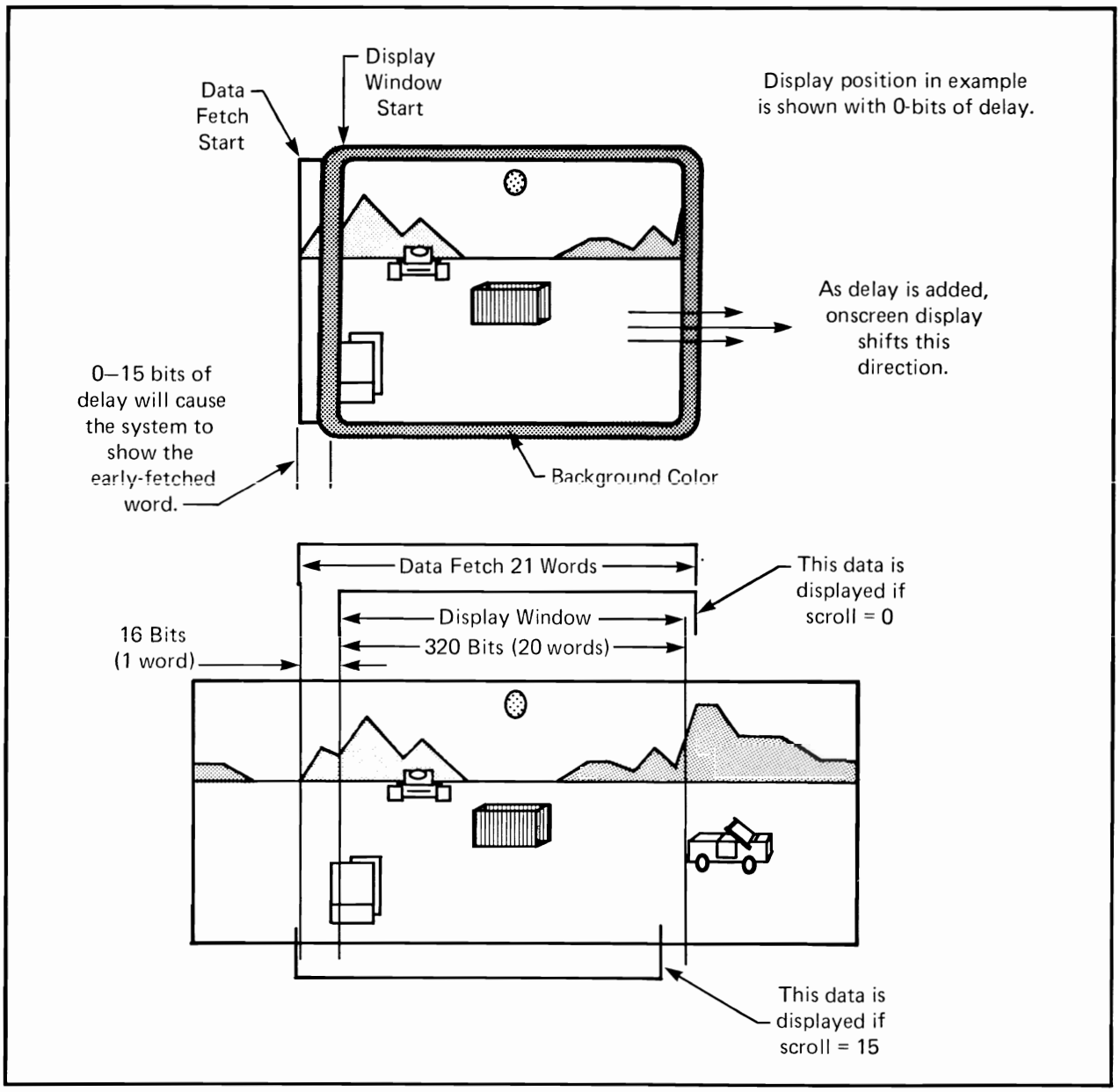


Figure 3-24: Horizontal Scrolling

To set up a playfield for horizontal scrolling, you need to:

- define bit-planes wide enough to allow for the scrolling you need
- set the data fetch registers to correctly place each horizontal line, including the extra word, on the screen
- set the delay bits
- set the modulo so the bit-plane pointers begin at the correct word for each line
- write Copper instructions to handle the changes during the vertical blanking interval

Specifying Data Fetch in Horizontal Scrolling

The normal data fetch start for non-scrolled displays is ($38). If horizontal scrolling is desired, then the data fetch must start one word sooner (DDFSTRT = $0030). Incidentally, this will disable sprite 7. DDFSTOP remains unchanged. Remember that the settings of the data fetch registers affect both playfields.

Specifying the Modulo in Horizontal Scrolling

As always, the modulo is two counts less than the difference between the address of the next word you want to fetch and the address of the last word that was fetched. As an example for horizontal scrolling, let us assume a 40-byte display in an 80-byte "big picture". Because horizontal scrolling requires a data fetch of 2 extra bytes, the data for each line will be 42 bytes long.

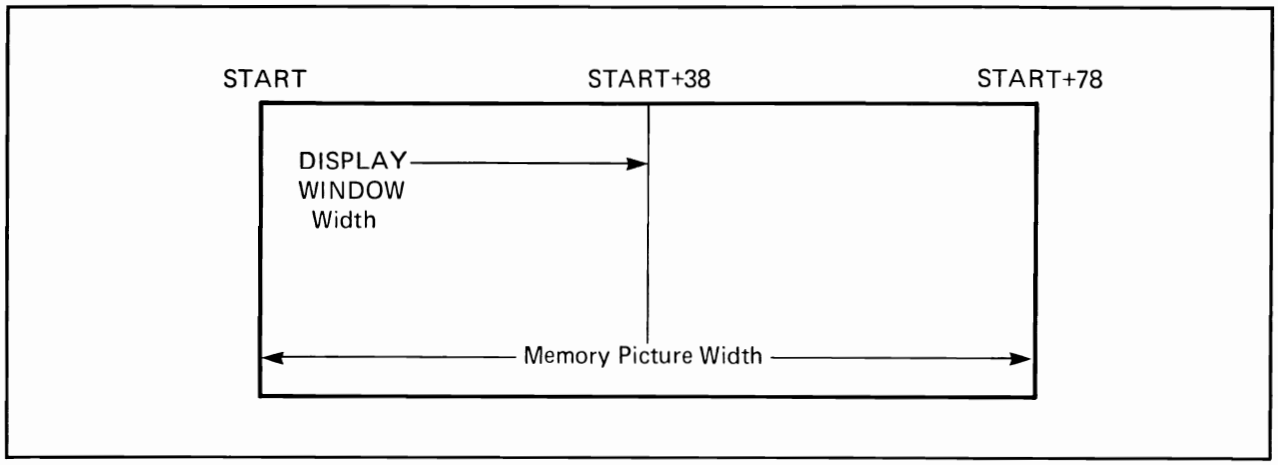


Figure 3-25: Memory Picture Larger than the Display Window

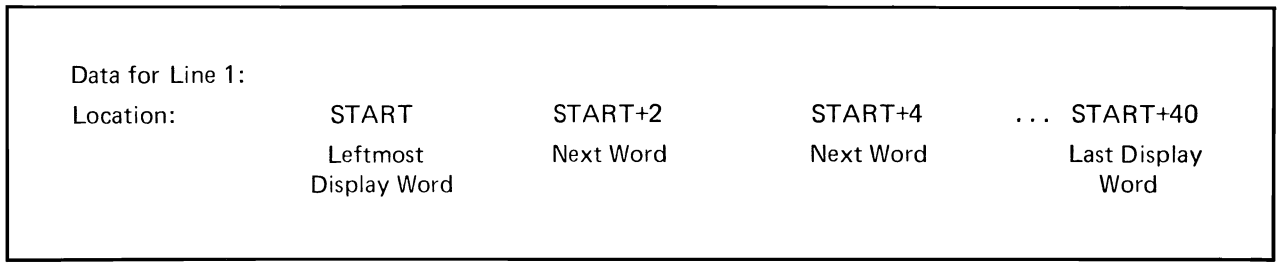


Figure 3-26: Data for Line 1 - Horizontal Scrolling

At this point the bit-plane pointers contain the value START+42. Adding the modulo of 38 gives the correct starting point for the next line.

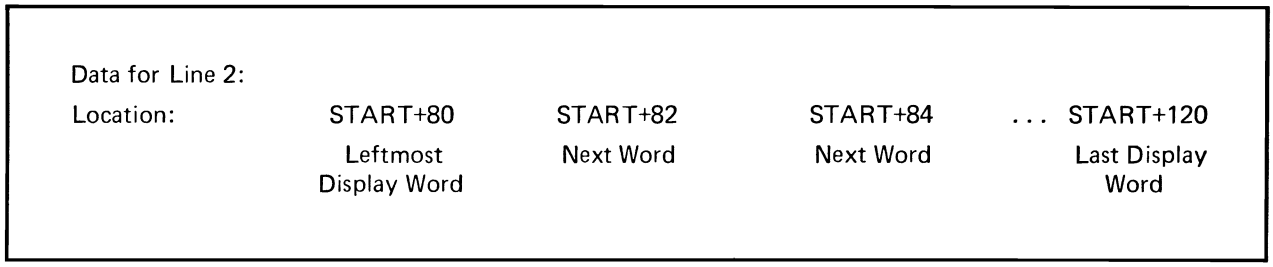


Figure 3-27: Data for Line 2 - Horizontal Scrolling

In the BPLxMOD registers you set the modulo for each bit-plane used.

Specifying Amount of Delay

The amount of delay in horizontal scrolling is controlled by bits 7-0 in BPLCON1. You set the delay separately for each playfield; bits 3-0 for Playfield 1 (bit-planes 1, 3, and 5) and bits 7-4 for Playfield 2 (bit-planes 2, 4, and 6).

NOTE: You always set all six bits, even if you have only one playfield. Set 3-0 and 7-4 to the same value if you are using only one playfield.

3.5.3. Summary

The steps for defining a scrolled playfield are almost the same as for the basic playfield. These steps are different:

- **Defining the data fetch**. Fetch one extra word per horizontal line and start it 16 pixels before the normal (unscrolled) data fetch start.

- **Defining the modulo**. The modulo is 2 counts greater than when there is no scrolling.

These steps are added:

- **For vertical scrolling, reset the bit-plane pointers for the amount of the scrolling increment**. Reset BPLxPTH and BPLxPTL during the vertical blanking interval.

- **For horizontal scrolling, specify the delay**. Set bits 7-0 in BPLCON1 for 0 to 15 bits of delay.

3.6. ADVANCED TOPICS

This section describes features that are used less often, or are optional.

3.6.1. Interactions Between Playfields and Other Objects

Playfields share the display with sprites. Chapter 7, "System Control Hardware" shows how:

- playfields can be given different video display priorities relative to the sprites.
- playfields can collide with (overlap) the sprites or each other.

3.6.2. Hold and Modify Mode

This is a special mode that allows you to produce more than 32 colors on-screen simultaneously. In this mode you can display up to 4,096 colors on the screen at the same time. Normally, as each value formed by the combination of bit-planes is selected, the data contained in the selected color register is loaded into the color output circuit for the pixel being written on the screen. Therefore, each pixel is colored by the contents of the selected color register.

In hold and modify mode, however, the value in the color output circuitry is held and one of the three components of the color (red, green, or blue) is modified by bits coming from certain pre-selected bit-planes. After modification, the pixel is written to the screen.

Hold and modify allows very fine gradients of color or shading to be produced on the screen. For example, you might draw a set of 16 vases, each a different color, using all 16 colors in the color palette. Then, for each vase, you use hold and modify to very finely shade or highlight or add a completely different color to each of the vases. Note that a particular hold and modify pixel can only change one of the three color values at a time. Thus, the effect has a limited control.

In hold and modify mode, you use all six bit-planes. planes 5 and 6 are used to modify the way bits from planes 1 - 4 are treated, as follows:

- If the 6-5 bit combination from planes 6 and 5 for any given pixel is 00, normal color selection procedure is followed. Thus, the bit combinations from planes 4-1, in that order of significance, are used to choose one of 16 color registers (registers 0 - 15).

  If only 5 bit-planes are used, the data from the 6th plane is automatically supplied with the value as 0.

- If the 6-5 bit combination is 01, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are

used to replace the 4 "Blue" bits in the corresponding color register.

- If the 6-5 bit combination is 10, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the 4 "Red" bits.

- If the 6-5 bit combination is 11, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the 4 "Green" bits.

Using hold and modify mode, it is possible to get by with defining only one color register, which is COLOR0, the color of the background. You treat the entire screen as a modification of that original color, according to the scheme above.

The bit that selects hold and modify mode is bit 11 of register BPLCON0. The following bits in BPLCON0 must be set for hold and modify mode to be active:

- Bit HOMOD, bit 11, is 1.

- Bit DBLPF, bit 10, is 0 (single playfield specified).

- Bit HIRES, bit 15, is 0 (normal resolution mode specified).

- Bits BPU2, BPU1, and BPU0 - bits 14, 13, and 12, are 101 or 110 (5 or 6 bit-planes active).

3.6.3. Forming a Display with Several Different Playfields

The graphics library provides the ability to split the screen into several viewports, each with its own colors and resolutions. See the "Amiga ROM Kernel Manual" for more information.

3.6.4. Using an External Video Source

There is an optional board available for the Amiga that provides genlock. Genlock allows you to bring in your graphics display from an external video source (such as a VCR, camera, or laser disk player). When you use genlock, the background color is replaced by the display from this external video source. For more information, see the instructions furnished with the optional board.

3.7. SUMMARY OF PLAYFIELD REGISTERS

This section summarizes the registers used in this chapter and the meaning of their bit settings. The color registers are summarized in the next section.

BPLCON0 - Bit Plane Control

NOTE: Bits in this register are not independently settable.

Bit 0 - unused

Bit 1 - ERSY (genlock video enable
1 = Genlock video enabled
0 = Genlock video disabled

Bit 2 - LACE (interlace enable)
1 = interlace mode enabled
0 = non-interlace mode enabled

Bit 3 - LPEN (light pen enable)

Bits 4-7 not used (make 0)

Bit 8 - GAUD (genlock audio enable)
1 = Genlock audio enabled
0 = Genlock audio disabled

Bit 9 - COLOR_ON (color enable)
1 = composite video color-burst enabled
0 = composite video color-burst disabled

Bit 10 - DBLPF (double-playfield enable)
1 = dual playfields enabled
0 = single playfield enabled

Bit 11 - HOMOD (hold-and-modify enable)
1 = hold and modify enabled
0 = hold and modify disabled

Bits 14, 13, 12 - BPU2, BPU1, BPU0
Number of bit-planes used.

000 = only a background color
001 = 1 bit-plane, PLANE 1
010 = 2 bit-planes, PLANES 1 and 2
011 = 3 bit-planes, PLANES 1 - 3
100 = 4 bit-planes, PLANES 1 - 4
101 = 5 bit-planes, PLANES 1 - 5
110 = 6 bit-planes, PLANES 1 - 6
111 not used

Bit 15 - HIRES (high-resolution enable)
1 = high-resolution mode
0 = low-resolution mode

BPLCON1 - Bit Plane Control

Bits 3-0 - PF1H(3-0)
Playfield 1 delay
Bits 7-4 - PF2H(3-0)
Playfield 2 delay
Bits 15-8 not used

BPLCON2 - Bit Plane Control

Bit 6 - PF2PRI

1 = Playfield 2 has priority
0 = Playfield 1 has priority

Bits 0-5 Playfield sprite priority
Bits 7-15 not used

DDFSTRT - Data Fetch Start
Beginning position for data fetch.

Bits 15-8 - not used
Bits 7-3 - pixel position H8-H4
Bits 2-0 - not used

DDFSTOP - Data Fetch Stop
Ending position for data fetch.

Bits 15-8 - not used
Bits 7-3 - pixel position H8-H4
Bits 1-0 - not used

BPLxPTH - Bit Plane Pointer
Bit plane pointer high word where *x* is the bit-plane number.

BPLxPTL - Bit Plane Pointer
Bit plane pointer low word where *x* is the bit-plane number.

DIWSTRT - Display Window Start
Starting vertical and horizontal coordinates.

Bits 15-8 - VSTART (V7-V0)
Bits 7-0 - HSTART (H7-H0)

DIWSTOP - Display Window Stop
Ending vertical and horizontal coordinates.

Bits 15-8 - VSTOP (V7-V0)
Bits 7-0 - HSTOP (H7-H0)

BPL1MOD - Bit-Plane Modulo
Odd-numbered bit-planes, Playfield 1.

BPL2MOD - Bit-Plane Modulo
Even-numbered bit-planes, Playfield 2.

3.8. SUMMARY OF COLOR SELECTION

This section contains summaries of playfield color selection including color register contents, example colors, and the differences in color selection in high resolution and low resolution modes.

3.8.1. Color Register Contents

The table below shows the contents of each color register. All color registers are write-only.

Table 3-10: Color Register Contents

Bits	15	-	12	(Unused)
Bits	11	-	8	Red
Bits	7	-	4	Green
Bits	3	-	0	Blue

3.8.2. Some Sample Color Register Contents

The table below shows a variety of colors and the hexadecimal values to load into the color registers for these colors.

Table 3-11: Some Register Values and Resulting Colors

VALUE	COLOR
$FFF	white
$D00	brick red
$F00	red
$F80	red-orange
$F90	orange
$FB0	golden orange
$FD0	cadmium yellow
$FF0	lemon yellow
$BF0	lime green
$8E0	light green
$0F0	green
$2C0	dark green
$0B1	forest green
$0BB	blue green
$0DB	aqua
$1FB	light aqua
$6FE	sky blue
$6CE	light blue
$00F	blue
$61F	bright blue
$06D	dark blue
$91F	purple
$C1F	violet
$F1F	magenta
$FAC	pink
$DB9	tan
$C80	brown
$A87	dark brown
$CCC	light grey
$999	medium grey
$000	black

3.8.3. Color Selection in Low Resolution Mode

The following table shows playfield color selection in low-resolution mode. If the bit-combinations from the playfields are as shown, then the color is taken from the color register number indicated.

Table 3-12: Low Resolution Color Selection

SINGLE PLAYFIELD		DUAL PLAYFIELD	COLOR REGISTER NUMBER
Normal Mode	Hold & Modify Mode		
Bit-planes 5,4,3,2,1	*Bit-planes 4,3,2,1*		
		Playfield 1 *Bit-Planes 5,3,1*	
00000	0000	000	0 *
00001	0001	001	1
00010	0010	010	2
00011	0011	011	3
00100	0100	100	4
00101	0101	101	5
00110	0100	110	6
00111	0111	111	7
		Playfield 2 *Bit-Planes 6,4,2*	
01000	1000	000 **	8
01001	1001	001	9
01010	1010	010	10
01011	1011	011	11
01100	1100	100	12
01101	1101	101	13
01110	1110	110	14
01111	1111	111	15
10000	|	|	16
10001	|	|	17
10010	|	|	18
10011	|	|	19
10100	NOT	NOT	20
10101	USED	USED	21
10110	IN	IN	22
10111	THIS	THIS	23
11000	MODE	MODE	24
11001	|	|	25
11010	|	|	26
11011	|	|	27
11100	|	|	28
11101	|	|	29
11110	|	|	30
11111	|	|	31

* Color Register 0 always defines the background color

** Selects "transparent" mode instead of selecting Register 8.

In hold and modify mode, the color register contents are changed as shown below. This

mode is in effect only if Bit 10 of BPLCON0 = 1.

Table 3-13: Color Selection in Hold and Modify Mode

HOLD AND MODIFY
6th and 5th Bit-Plane Special Mode

Bit-Plane 6	Bit-Plane 5		Result
0	0	Normal Operation	(use Color Register itself)
0	1	Hold green and red	B = Bit-Plane 4-1 contents
1	0	Hold green and blue	R = Bit-Plane 4-1 contents
1	1	Hold blue and red	G = Bit-Plane 4-1 contents

3.8.4. Color Selection in High Resolution Mode

The following table shows playfield color selection in high-resolution mode. If the bit-combinations from the playfields are as shown, then the color is taken from the color register number indicated.

Table 3-14: High Resolution Color Selection

SINGLE PLAYFIELD *Bit-planes 4,3,2,1*	DUAL PLAYFIELD	COLOR REGISTER NUMBER
	Playfield 1 *Bit-planes 3,1*	
0000	00 **	0 *
0001	01	1
0010	10	2
0011	11	3
0100	|	4
0101	NOT USED	5
0110	IN THIS MODE	6
0111	|	7
	Playfield 2 *Bit-planes 4,2*	
1000	00 **	8
1001	01	9
1010	10	10
1011	11	11
1100	|	12
1101	NOT USED	13
1110	IN THIS MODE	14
1111	|	15

* Color Register 0 always defines the background color.

** Selects "transparent" mode.

Chapter 4

SPRITE HARDWARE

4.1. INTRODUCTION

This chapter is about sprites, hardware objects that are created and moved independently of the playfield display and independently of each other. Together with playfields, sprites form the graphics display of the Amiga. You can create more complex animation effects by using the blitter described in the chapter called "Blitter Hardware". Sprites are easily movable graphics objects produced on-screen by 8 special-purpose sprite DMA channels. Basic sprites are 16 pixels wide and any number of lines high. You can choose from 3 colors for a sprite's pixels and a pixel may also be transparent, showing any object behind the sprite. For larger or more complex objects, or more color choices, you can combine sprites.

Sprite DMA channels can be reused several times within the same display field. Thus, you are not limited to having only 8 sprites on the screen at the same time.

About this Chapter

This chapter shows how to:

- Define the size, shape, color and screen position of sprites.
- Display and move sprites
- Combine sprites for more complex images, additional width, or additional colors
- Reuse a sprite DMA channel multiple times within a display field to create more than 8 sprites on the screen at one time.

4.2. FORMING A SPRITE

To form a sprite, you first need to define it and then create a formal data structure in memory. You define a sprite by specifying its characteristics:

- On-screen width of up to 16 pixels
- Unlimited height
- Any shape
- A combination of 3 colors, plus transparent
- Any position on the screen

4.2.1. Screen Position

A sprite's screen position is defined as a set of X,Y coordinates. Position (0,0), where X = 0 and Y = 0, is the upper left-hand corner of the display. You define a sprite's location by specifying the coordinates of its upper left-hand pixel. Sprite position is always defined as though the display mode were low-resolution and non-interlace. The X,Y coordinate system and definition of a sprite's position are graphically represented in Figure 4-1. Notice that because of display overscan, the coordinates (0,0) (that is, X = 0, Y = 0) are not normally in a viewable region of the screen. For more information about overscan and the visible area of the screen, see the "Playfield Hardware" chapter.

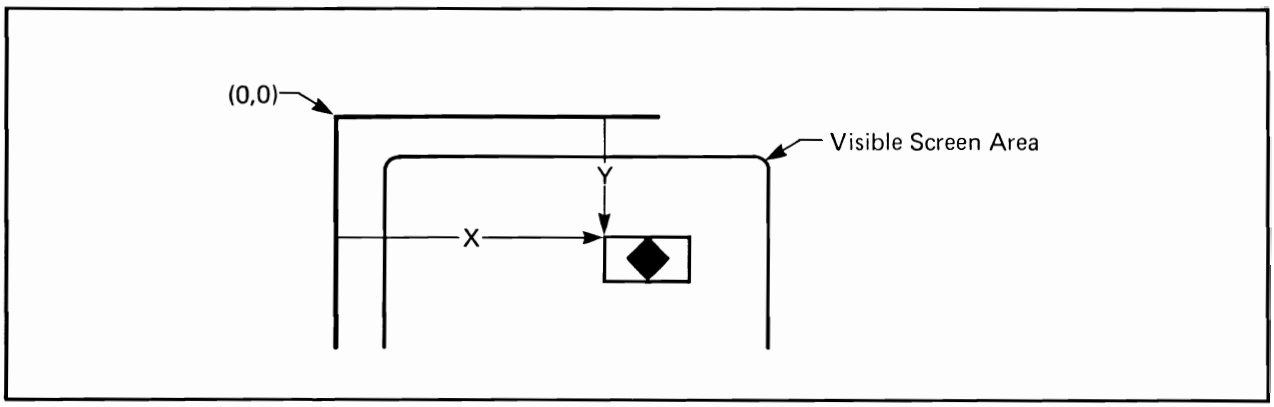


Figure 4-1: Defining Sprite On-screen Position

The amount of viewable area is also affected by the size of the playfield display window. See the "Playfield Hardware" chapter for more information about overscan and display windows.

Horizontal Position

A sprite's horizontal position (X value) can be at any pixel on the screen from 0 to 447. To be visible, however, any object must be within the boundaries of the playfield display window. In addition, the normally usable range of the video screen is from pixel 64 to pixel 383 (that is, 320 pixels of usable width). A larger area is actually scanned by the video beam but is not usually visible on the screen.

If you specify an X value for the sprite of less than 64 or an X value outside the display window, part or all of the sprite may not appear on the screen. This is sometimes desirable, and such a sprite is said to be "clipped".

To have a sprite appear, unclipped, in its correct on-screen horizontal position, add 64 to the X value. For example, to make the upper leftmost pixel of a sprite appear at a position 94 pixels from the left edge of the screen, you perform this calculation:

Desired X position	94
32 off-screen lines	+64
	158

Thus, 158 becomes the X value which will be written into the data structure.

Note that the X position represents the location of the very first (leftmost) pixel in the full 16-bit-wide sprite. This is always the case even if the leftmost pixels are specified as transparent and do not appear on the screen. If the sprite shown in Figure 4-2 were located at an X value of 158, the actual image would begin on-screen 4 pixels later at 162. The first 4 pixels in this sprite are transparent and allow the the background to show through.

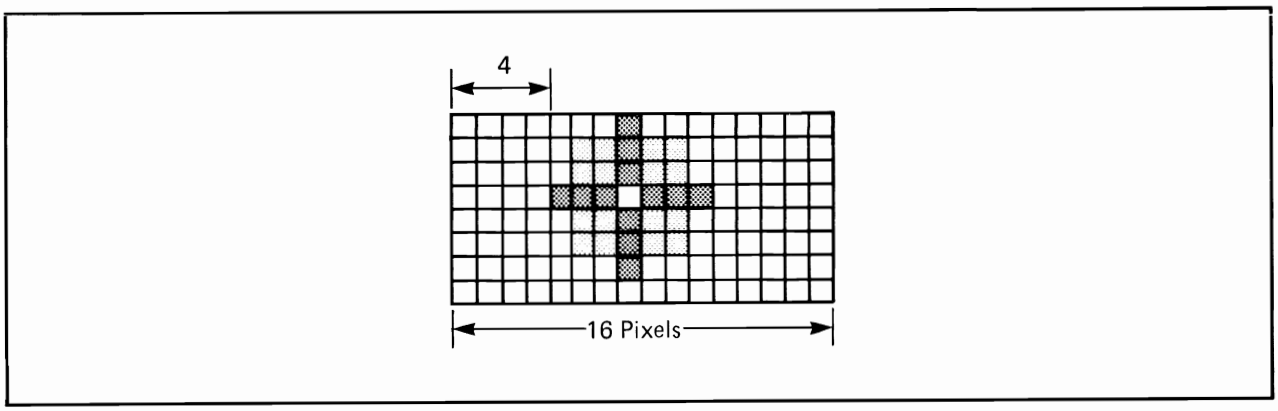


Figure 4-2: Position of Sprites

Vertical Position

You can select any position from line 0 to line 262 for the topmost edge of the sprite. The normal usable range of the video screen, however, is from line 44 through line 243. This allows the normal display height of 200 lines in non-interlace mode. If you specify a vertical

position (Y value) of less than 44, the top edge of the sprite may not appear on screen.

To have a sprite appear in its correct on-screen vertical position, add 44 to the desired position. For example, to make the upper leftmost pixel appear 25 lines below the top edge of the screen, perform this calculation:

Desired Y position	25
44 above-screen lines	+44
	69

Thus, 69 is the Y-value you will write into the data structure.

Clipped Sprites

As noted above, sprites will be partially or totally clipped if they pass across or beyond the boundaries of the display window. The values of 64 (horizontal) and 44 (vertical) are "normal" for a centered display on a standard video monitor. If you choose other values to establish your display window, your sprites will be clipped accordingly.

Size of Sprites

Sprites are 16 pixels wide and can be almost any height you wish — as little as 1 line tall or taller than the screen. You would probably move a very tall sprite vertically to display a portion of it at a time.

Sprite size is based on a pixel that is 1/320th of a normal screen width and 1/200th of a normal screen height. This pixel size corresponds to the normal resolution and non-interlace modes of the normal full-size playfield. Sprites, however, are independent of playfield modes of display, so changing the resolution or interlace mode of the playfield has no effect on the size or resolution of a sprite.

4.2.2. Shape of Sprites

A sprite can have any shape that will fit within the 16-pixel width. You define a sprite's shape by specifying which pixels actually appear in each of the sprite's locations. For example, Figures 4-3 and 4-4 show a spaceship whose shape is marked by X's. The first figure shows only the spaceship as you might sketch it out on graph paper. The second figure

shows the spaceship within the 16-pixel width. The 0's around the spaceship mark the part of the sprite not covered by the spaceship and transparent when displayed.

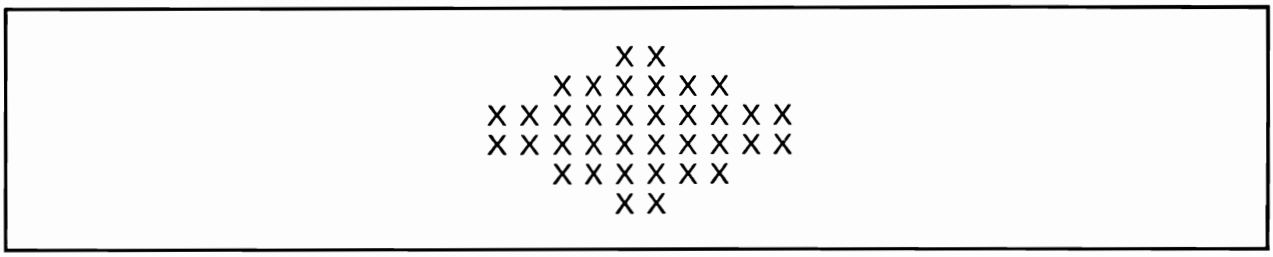


Figure 4-3: Shape of Spaceship

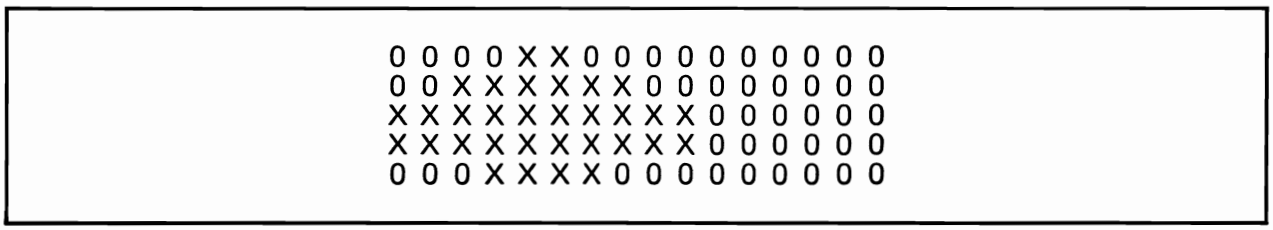


Figure 4-4: Sprite with Spaceship Shape Defined

In this example, the widest part of the shape is 10 pixels and the shape is shifted to the left of the sprite. Whenever the shape is narrower than the sprite, you can control which part of the sprite is used to define the shape. This particular shape could also start at any of the pixels from 2-7 instead of pixel 1.

4.2.3. Sprite Color

When sprites are used individually (that is, not "attached" as described under "Attached Sprites" later), each pixel can be one of 3 colors, or transparent. Colors are selected in much the same manner as playfield colors. Figure 4-5 shows how the color of each pixel in a sprite is determined. The 0's and 1's in the two data words that define each line of a sprite in the data structure form a binary number.

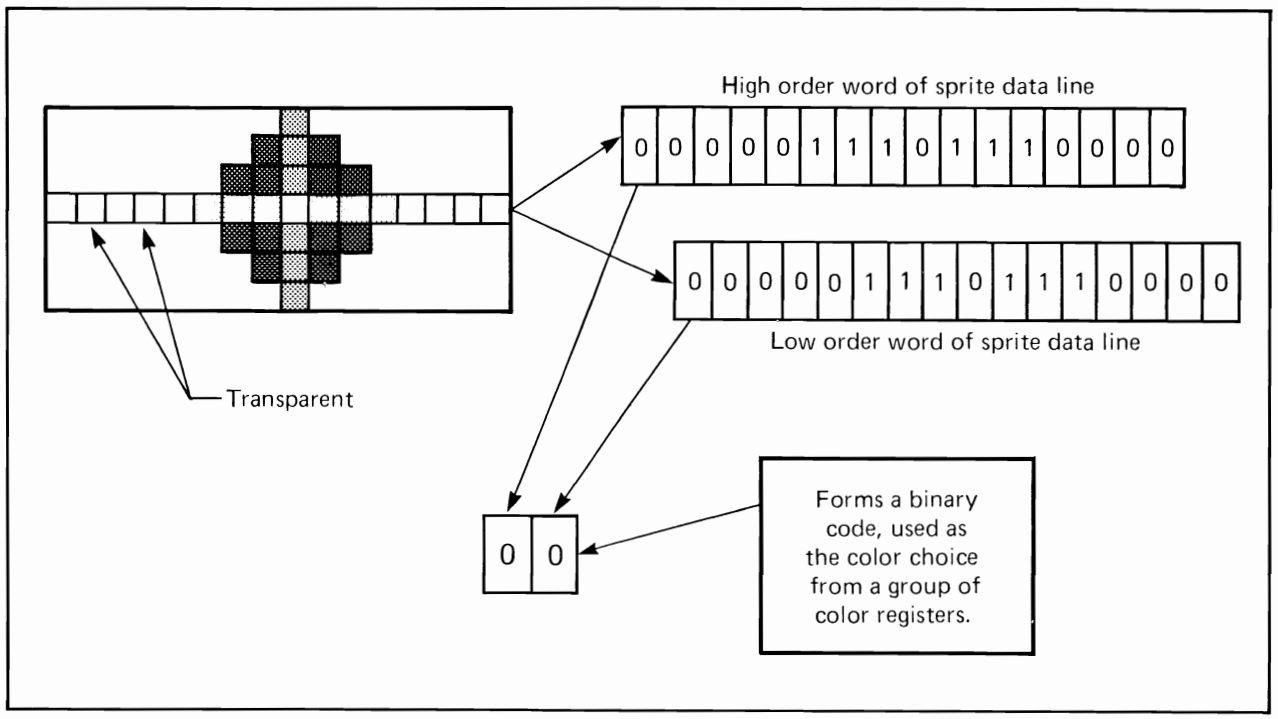


Figure 4-5: Sprite Color Definition

This binary number points to one of the 4 color registers assigned to that particular sprite DMA channel. There are 32 color registers in the system, each containing a user-defined color. The 8 sprites use color registers 16 - 31. For purposes of color selection, the 8 sprites are organized into pairs and each pair uses 4 of the color registers as shown in the Figure 4-6. Note that the color value of the first register in each group of 4 registers is ignored by sprites. When the sprite bits select this register, "transparent" is used.

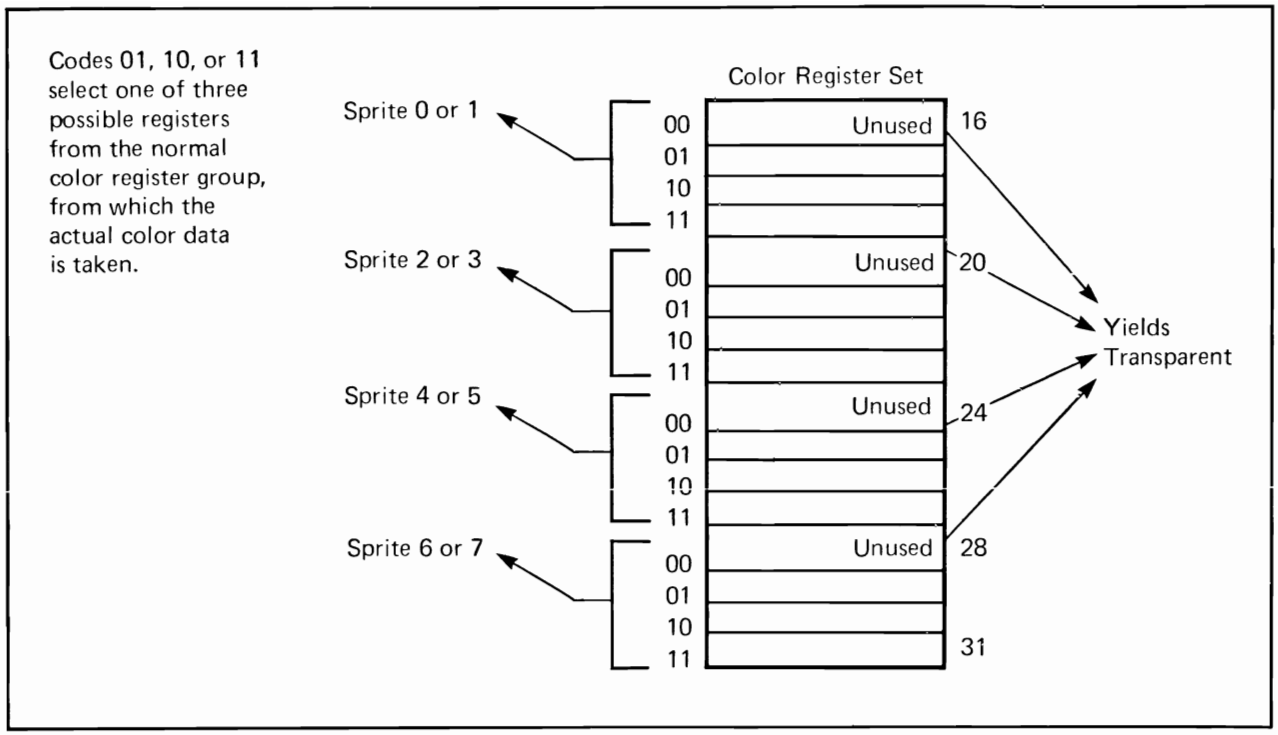


Figure 4-6: Color Register Assignments

If you require certain colors in a sprite, you will want to load the sprite's color registers with those colors. The "Playfield Hardware" chapter contains instructions on loading color registers.

The binary number 00 is special in this color scheme. A pixel whose value is 00 becomes transparent and shows the color of any other sprite or playfield that has lower video priority. An object with low priority appears "behind" an object with higher priority. Each sprite has a fixed video priority with respect to all the other sprites. You can vary the priority between sprites and playfields. (See Chapter 7, "System Control Hardware", for more information about sprite priority.)

4.2.4. Designing a Sprite

To design a sprite, it is convenient to lay out the sprite on paper first. You can show the desired colors as numbers from 0 to 3. For example, the spaceship shown above might look like this:

```
0000122332210000
0001223333221000
0012223333222100
0001223333221000
0000122332210000
```

The next step is to convert the numbers 0-3 into binary numbers, which will be used to build the color descriptor words of the sprite data structure. The section below shows how to do this.

4.2.5. Building the Data Structure

After defining the sprite, you need to build its data structure, which is a series of 16-bit words in a contiguous memory area. Some of the words contain position and control and information and some contain color descriptions. To create a sprite's data structure, you need to:

- Write the horizontal and vertical position of the sprite into the first control word.
- Write the vertical stopping position into the second control word.
- Translate the decimal color numbers 0 - 3 in your sprite grid picture into binary color numbers. Use the binary values to build color descriptor (data) words and write these words into the data structure.
- Write the control words that indicate the end of the sprite data structure.

The following shows a sprite data structure with the memory location and function of each word:

Memory Location	16-bit Word	Function
N	Sprite Control Word 1	Vertical and horizontal start position
N+1	Sprite Control Word 2	Vertical stop position
N+2	Color descriptor low word	Color bits for Line 1
N+3	Color descriptor high word	Color bits for Line 1
N+4	Color descriptor low word	Color bits for Line 2
N+5	Color descriptor high word	Color bits for Line 2
	.	
	.	
	.	
	End-of-data words	Two words indicating the next usage of this sprite

All memory addresses for sprites are word addresses. You will need enough contiguous memory to provide room for:

2 words for the control information
2 words for each horizontal line in the sprite
2 end-of-data words

Because this data structure must be accessible by the special-purpose chips, you must ensure that this data is located within the lowest 512K bytes of the system memory.

Figure 4-7 shows how the data structure relates to the sprite.

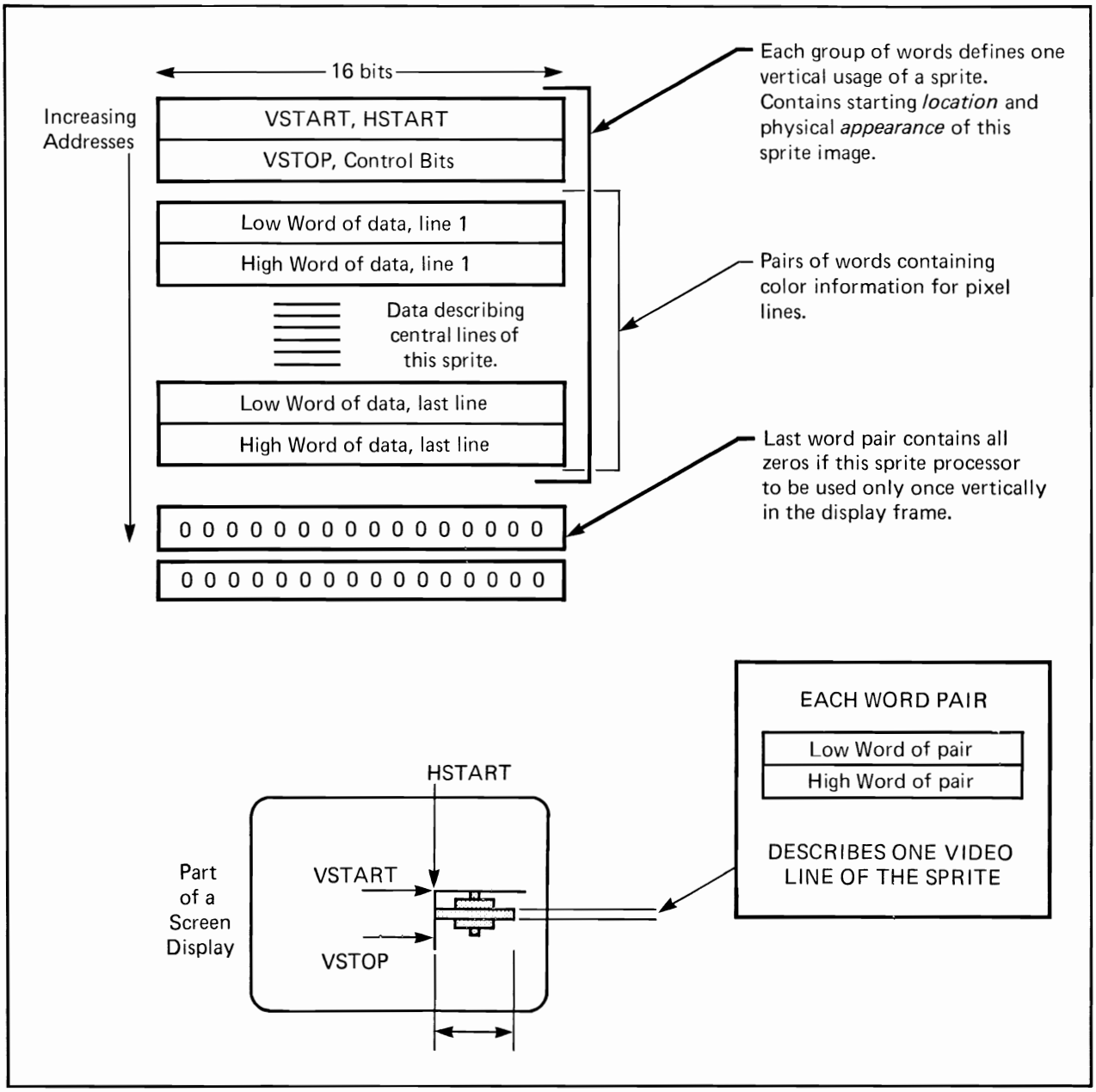


Figure 4-7: Data Structure Layout

Sprite Control Word 1 : SPRxPOS

This word contains the vertical (VSTART) and horizontal (HSTART) starting position for the sprite. This is where the topmost line of the sprite will be positioned.

Bits 15-8 contain the low 8 bits of VSTART
Bits 7-0 contain the high 8 bits of HSTART

Sprite Control Word 2 : SPRxCTL

This word contains the vertical stopping position of the sprite on the screen. This word also contains some data having to do with sprite attachment, which is described later on.

SPRxCTL

Bits 15-8	the low 8 bits of VSTOP
Bit 7	(used in attachment)
Bits 6-3	unused (make zero)
Bit 2	the VSTART high bit
Bit 1	the VSTOP high bit
Bit 0	the HSTART low bit

The value (VSTOP - VSTART + 1) defines how many lines high the sprite will be.

Sprite Color Descriptor Words

It takes two color descriptor words to describe each horizontal line of a sprite, the high order word and the lower order word. To calculate how many color descriptor words you need, multiply the height of the sprite in lines by 2. The bits in the high order color descriptor word contribute the leftmost digit of the binary color selector number for each pixel; the low order word contributes the rightmost digit.

To form the color descriptor words, you first need to form a picture of the sprite, showing the color of each pixel as a number from 0 - 3. Each number represents one of the colors in the sprite's color registers. For example, here is the spaceship sprite again:

0000122332210000
0001223333221000
0012223333222100
0001223333221000
0000122332210000

Next, you translate each of the numbers in this picture into a binary number. The first line in binary is shown below. The binary numbers are represented vertically with the low digit in the top line and the high digit right below it. This is how the two color descriptor words for each sprite line are written in memory.

0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0

The first line above becomes the color descriptor high word for line 1 of the sprite. The second line becomes the color descriptor low word. In this fashion, you translate each line in the sprite into binary 0's and 1's.

Each of the binary numbers formed by the combination of the two data words for each line refers to a specific color register in that particular sprite channel's segment of the color table. Sprite channel 0, for example, takes its colors from registers 17 - 19. The binary numbers corresponding to the color registers for sprite DMA channel 0 are shown below:

Binary Number	Color Register Number
00	transparent
01	17
10	18
11	19

Recall that binary 00 always means transparent and never refers to a color.

End-of-data Words

When the vertical position of the beam counter is equal to the VSTOP value in the sprite control words, the next two words fetched from the sprite data structure are written into the sprite control registers instead of being sent to the color registers. These two words are interpreted by the hardware in the same manner as the original words first loaded into the

control registers. If the VSTART value contained in these words is lower than the current beam position, then this sprite will not be reused in this field. For consistency, the value 0 should be used for both words when ending the usage of a sprite. Sprite reuse is discussed later on.

4.3. DISPLAYING A SPRITE

After building the data structure, which contains the size and color of the sprite and its initial position, you need to tell the system to display it. This section describes the display of sprites in "automatic" mode. In automatic mode, once the sprite DMA channel begins to retrieve and display the data, the display continues until the VSTOP position is reached. Manual mode is described later on in this chapter.

The steps in displaying the sprite are:

1. Decide which of the 8 sprite DMA channels to use.
2. Set the sprite pointers to tell the system where to find the sprite data.
3. Turn on sprite direct memory access if it is not already on.
4. For each subsequent display field, during the vertical blanking interval, rewrite the sprite pointers.

4.3.1. Selecting the Sprite DMA Channel and Setting the Pointers

In deciding which DMA channel to use, you should take into consideration the colors assigned to the sprite and the sprite's video priority.

The sprite DMA channel uses 2 pointers to read in sprite data and control words. During the vertical blanking interval before the first display of the sprite, you need to write the sprite's memory address into these pointers.

There are two pointers for each sprite, called SPRxPTH and SPRxPTL, where "x" is the number of the sprite DMA channel. SPRxPTH points to the high 3 bits of the memory

address of the first word in the sprite and SPRxPTL points to the low 15 bits. As usual, you can write a long word into SPRxPTH.

These pointers are dynamic; they are incremented by the sprite DMA channel to point first to the control words, then to the data words, and finally to the end-of-data words. After reading in the sprite control information and storing it in other registers, they proceed to read in the color descriptor words. The color descriptor words are stored in sprite data registers, which are used by the sprite DMA channel to display the data on screen. For more information about how the sprite DMA channels handle the display, see the "Hardware Details" section below.

4.3.2. Resetting the Address Pointers

For one single display field, the system will automatically read the data structure and produce the sprite on-screen in the colors in the sprite's color registers. If you want the sprite to be displayed in subsequent display fields, you have to rewrite the contents of the sprite pointers during each vertical blanking interval. The pointers must be rewritten because during the display field, they are incremented to point to the data which is being fetched as the screen display progresses.

The rewrite becomes part of the vertical blanking routine that can be handled by instructions in the Copper lists.

4.4. MOVING A SPRITE

This section shows how to move a sprite generated in automatic mode. Moving a sprite is a simple matter of specifying a different position in the data structure. The data will be reread and the sprite redrawn automatically for each display field. Therefore, if you change the position data before the sprite is redrawn, it appears elsewhere and seems to be moving.

You must take care that you are not moving the sprite (that is, changing control word data) at the same time the system is using that data to find out where to display the object. As data is retrieved, the system might find the start position for one field and the stop position for the following field. This would cause the system to misinterpret the data, causing a "glitch" and messing up the screen. Therefore, you should only change the content of the control words during a time when the system is not trying to read them. Usually, the vertical blanking period is a safe time, so moving the sprites becomes part of the vertical blanking tasks and is handled by the Copper as shown in the example below.

As sprites move about on the screen, they can collide with each other or with either of the two playfields. You can use the hardware to detect these collisions and exploit this capability for special effects. You can use collision detection to keep a moving object within specified on-screen boundaries. Collision is described in Chapter 7, "System Control Hardware".

4.5. CREATING ADDITIONAL SPRITES

To use additional sprites, you create a data structure for each one and arrange the display as shown in the previous section, naming the pointers SPR1PTH and SPR1PTL for sprite DMA channel 1, SPR2PTH and SPR2PTL for sprite DMA channel 2, and so on.

Note that when you enable sprite DMA for one sprite, you enable DMA for all the sprites and place them all in automatic mode. Thus, you do not need to repeat this step when using additional sprite DMA channels. Once the sprite DMA channels are enabled, all 8 sprite pointers must be initialized to either a real sprite or a safe null sprite. An uninitialized sprite could cause spurious sprite video to appear.

Also, recall that each pair of sprites takes its color from different color registers, as shown in the table below:

Sprite Number	Color Registers
0 and 1	17 - 19
2 and 3	21 - 23
4 and 5	25 - 27
6 and 7	29 - 31

When you have more than one sprite on the screen, you may need to take into consideration their relative video priority, that is, which sprite appears in front of or behind another. Each sprite has a fixed video priority with respect to all the others. The lowest numbered sprite has the highest priority and appears in front of all other sprites. The highest numbered sprite has the lowest priority. This is illustrated in Figure 4-8.

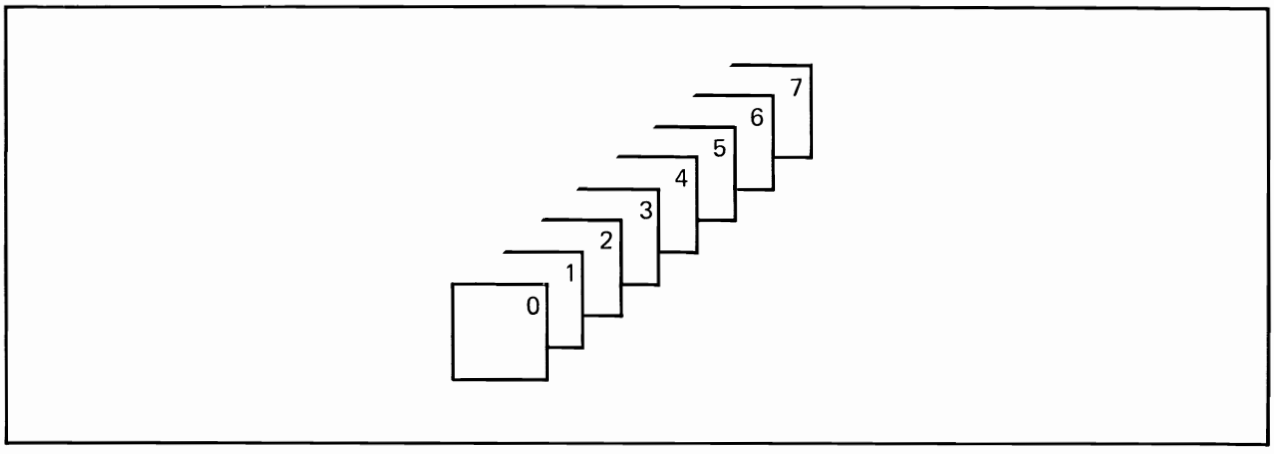


Figure 4-8: Sprite Priority

4.6. REUSING SPRITE DMA CHANNELS

Each of the 8 sprite DMA channels can produce more than 1 independently controllable image. There may be times when you want more than 8 objects. Or, you may have attached some of the sprites to produce more colors or larger objects or overlapped some to produce more complex images and are left with fewer than 8 objects. You can reuse each sprite DMA channel several times within the same display field, as shown in Figure 4-9.

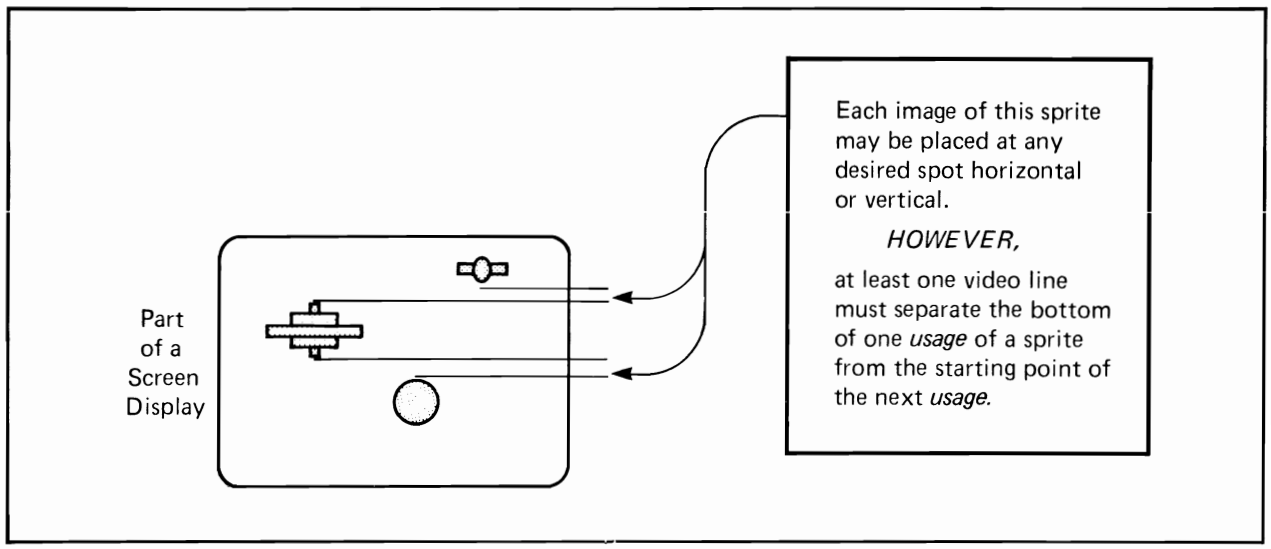


Figure 4-9: Typical Example of Sprite Re-use

In single-sprite usage, two all-zero words are placed at the end of the data structure to stop the DMA channel from retrieving any more data for that particular sprite during that display field. To reuse a DMA channel, you replace this pair of zero words with another complete sprite data structure, which describes the reuse of the DMA channel a position lower on the screen than the first use. You place the two all-zero words at the end of the data structure which contains the information for all usages of the DMA channel. For example, Figure 4-10 shows the data structure which describes the picture above.

The only restrictions on reuse of sprites during a single display field is that the bottom line of one usage of a sprite must be separated from the top line of the next usage by at least one horizontal scan line. This restriction is necessary because only two DMA cycles per

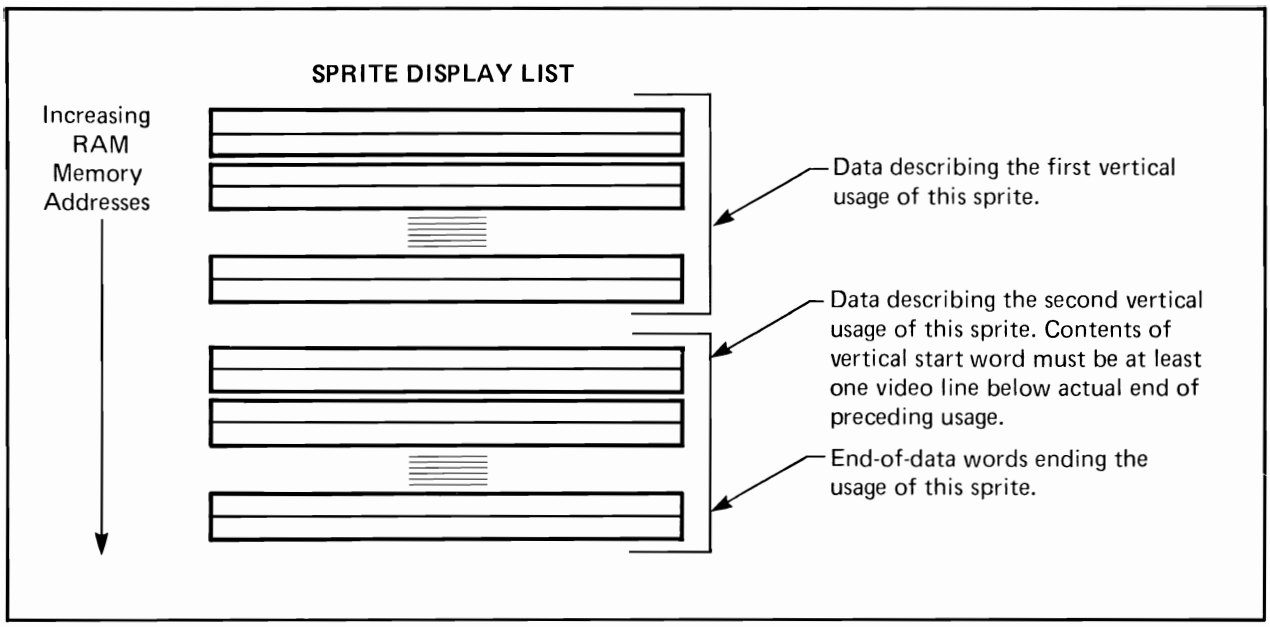


Figure 4-10: Typical Data Structure for Sprite Re-use

horizontal scan line are allotted to each of the 8 channels. The sprite channel needs the time during the blank line to fetch the control word describing the next usage of the sprite.

4.7. OVERLAPPED SPRITES

For more complex or larger moving objects, you can overlap sprites. Overlapping simply means that the sprites have the same or relatively close screen positions. A relatively close screen position can result in an object that is wider than 16 pixels.

The built-in sprite video priority ensures that one sprite appears to be behind the the other when sprites are overlapped. The priority circuitry gives the lowest-numbered sprite the highest priority and the highest numbered sprite the lowest priority. Therefore, when designing displays with overlapped sprites, make sure the "foreground" sprite has a lower number than the "background" sprite. In Figure 4-11, for example, the cage should be generated by a lower-numbered sprite DMA channel than the monkey.

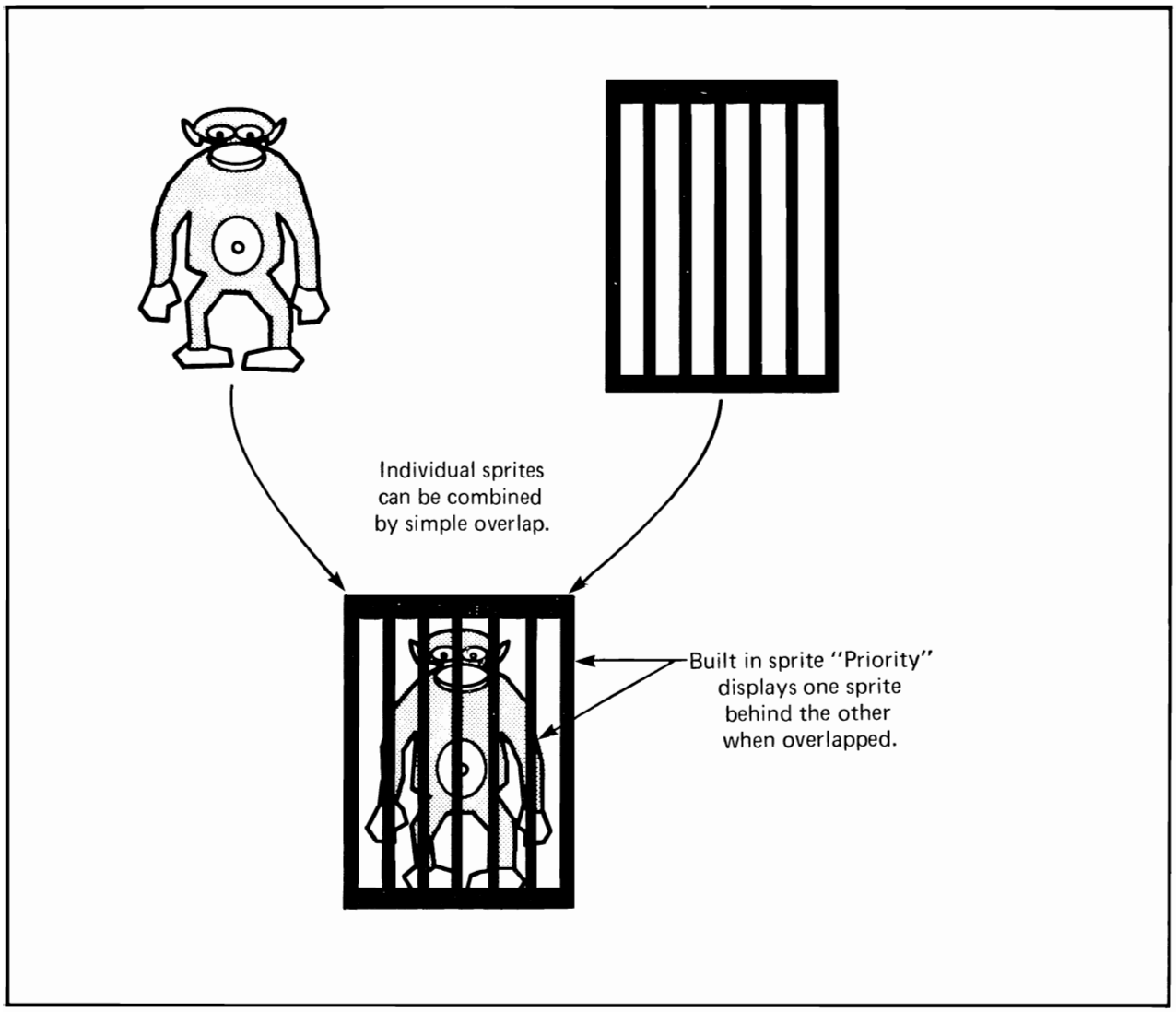


Figure 4-11: Overlapping Sprites (not attached)

You can create a wider sprite display by placing two sprites next to each other. For instance, Figure 4-12 shows the spaceship sprite and how it can be made twice as large by using two sprites placed next to each other.

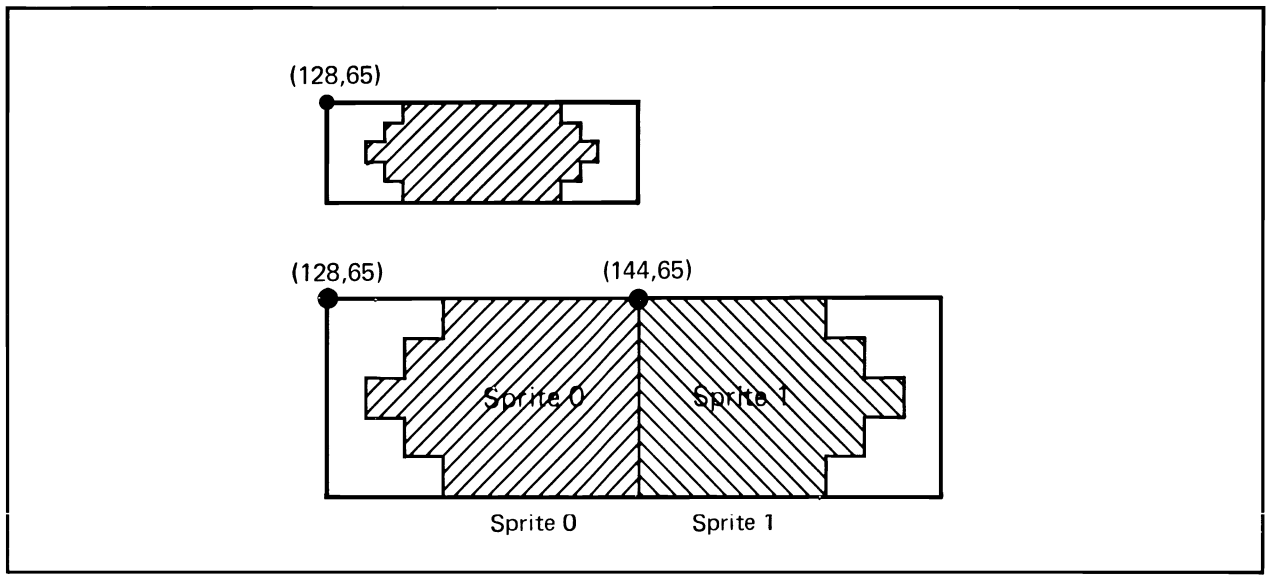


Figure 4-12: Placing Sprites Next to Each Other

4.8. ATTACHED SPRITES

You can create sprites that have 15 possible color choices (plus transparent) instead of 3 (plus transparent), by "attaching" two sprites. To create attached sprites, you need to:

- Use 2 channels per sprite, creating two sprites of the same size and located at the same position.
- Set a bit called ATTACH in the second sprite control word.

The 15 colors are selected from the full range of color registers available to sprites — registers 17 through 31. The extra color choices are possible as each pixel contains 4 bits instead of only 2 as in the normal, unattached sprite. Each sprite in the attached pair contributes 2 bits to the binary color selector number. For example, if you are using sprite DMA channels 0 and 1, the high and low order color descriptor words for line 1 in both data structures are combined into line 1 of the attached object.

Sprites can be attached in the combinations shown in the following table:

sprite 1 to sprite 0
sprite 3 to sprite 2
sprite 5 to sprite 4
sprite 7 to sprite 6

Any or all of these attachments can be active during the same display field. As an example, assume that you wish to have more colors in the spaceship sprite and you are using sprite DMA channels 0 and 1. There are 5 colors plus transparent in this sprite.

0000154444510000
0001564444651000
0015676446765100
0001564444651000
0000154444510000

The first line in this sprite requires the following 4 data words to form the correct binary color selector numbers:

	Pixel Number															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Line 1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line 2:	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
Line 3:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line 4:	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0

The binary numbers 0 through 15 select registers 17 through 31 as shown in the table below:

Table 4-1: Color Registers in Attached Sprites

Decimal Number	Binary Number	Color Register Number
0	0000	16 *
1	0001	17
2	0010	18
3	0011	19
4	0100	20
5	0101	21
6	0110	22
7	0111	23
8	1000	24
9	1001	25
10	1010	26
11	1011	27
12	1100	28
13	1101	29
14	1110	30
15	1111	31

* Unused, yields transparent pixel.

The highest numbered sprite (number 1, in this example), contributes the highest order bits (leftmost) in the binary number. The high order data word in each sprite contributes the leftmost digit. Therefore, the lines above are written to the sprite data structures like this:

Line 1	Sprite 1 high order word for sprite line 1
Line 2	Sprite 1 low order word for sprite line 1
Line 3	Sprite 0 high order word for sprite line 1
Line 4	Sprite 0 low order word for sprite line 1

Attachment is in effect only when the ATTACH bit, bit 7 in Sprite Control Word 2, is set to 1 in the data structure for the odd-numbered sprite. So, in this example, you set bit 7 in Sprite Control Word 2 in the data structure for Sprite 1.

When the sprites are moved, the Copper list must keep them both at exactly the same position. If they are not kept together on the screen, then their pixels will change color. Each sprite will revert to 3 colors plus transparent, but the colors may be different than if they were ordinary, unattached sprites. The color selection for the lower numbered sprite will be from color registers 17-19. The color selection for the higher numbered sprite will be from color registers 20, 24, and 28.

4.9. MANUAL MODE

It is almost always best to load sprites by using the automatic DMA channels. Sometimes, however, it is useful to load these registers directly from one of the microprocessors. Sprites may be activated "manually" whenever they are not being used by a DMA channel. The same sprite that is showing a DMA-controlled icon near the top of the screen can also be reloaded manually to show a vertical colored bar near the bottom of the screen. Sprites can be activated manually even when the sprite DMA is turned off.

You display sprites manually by writing to the sprite data registers SPRxDATB and SPRxDATA, in that order. You write to SPRxDATA last because that address "arms" the sprite to be output at the next horizontal comparison. The data written will then be displayed on every line, at the horizontal position given in the "H" portion of the position registers SPRxPOS and SPRxCTL. If the data is unchanged, the result will be a vertical bar. If the data is reloaded very line, a complex sprite can be produced.

The sprite can be terminated ("disarmed") by writing to the SPRxCTL register. If you write to the SPRxPOS register, you can manually move the sprite horizontally at any time, even during normal sprite usage.

4.10. SPRITE HARDWARE DETAILS

Sprites are produced by the circuitry shown in Figure 4-13. This figure shows in block form how a pair of data words becomes a set of pixels displayed on the screen.

The circuitry elements for sprite display are explained below.

Sprite data registers

The registers SPRxDATA and SPRxDATB hold the bit patterns that describe one horizontal line of a sprite for each of the 8 sprites. A line is 16 pixels wide and each line is defined by two words to provide selection of 3 colors and transparent.

Parallel-to-serial converters

Each of the 16 bits of the sprite data bit pattern is individually sent to the color select circuitry at the time that the pixel associated with that bit is being displayed on-screen.

Immediately after the data is transferred from the sprite data registers, each parallel-to-serial converter begins shifting the bits out of the converter, most significant bit (left-most bit) first. The shift occurs once each low-resolution pixel time, and continues until all 16 bits have been transferred to the display circuitry. The shifting and data output does not begin again until the next time this converter is loaded from the data registers.

Because the video image is being produced by an electron beam being swept from left to right on the screen, the bit-image of the data corresponds exactly to the image that actually appears on the screen (most significant data on the left).

Sprite serial video data

Sprite data goes to the priority circuit to establish the priority between sprites and

playfields.

Sprite position register
: The registers, called SPRxPOS, contain the horizontal position value (X value) and vertical position value (Y value) for each of the 8 sprites.

Sprite control register
: These registers, called SPRxCTL, contain the stopping position for each of the 8 sprites and whether or not a sprite is attached.

Beam counter
: The beam counter tells the system the current location of the video beam that is producing the picture.

Comparator
: This device compares the value of the beam counter to the Y value in the position register SPRxPOS. If the beam has reached the position at which the leftmost upper pixel of the sprite is to appear, the comparator issues a load signal to the serial-to-parallel converter and the sprite display begins.

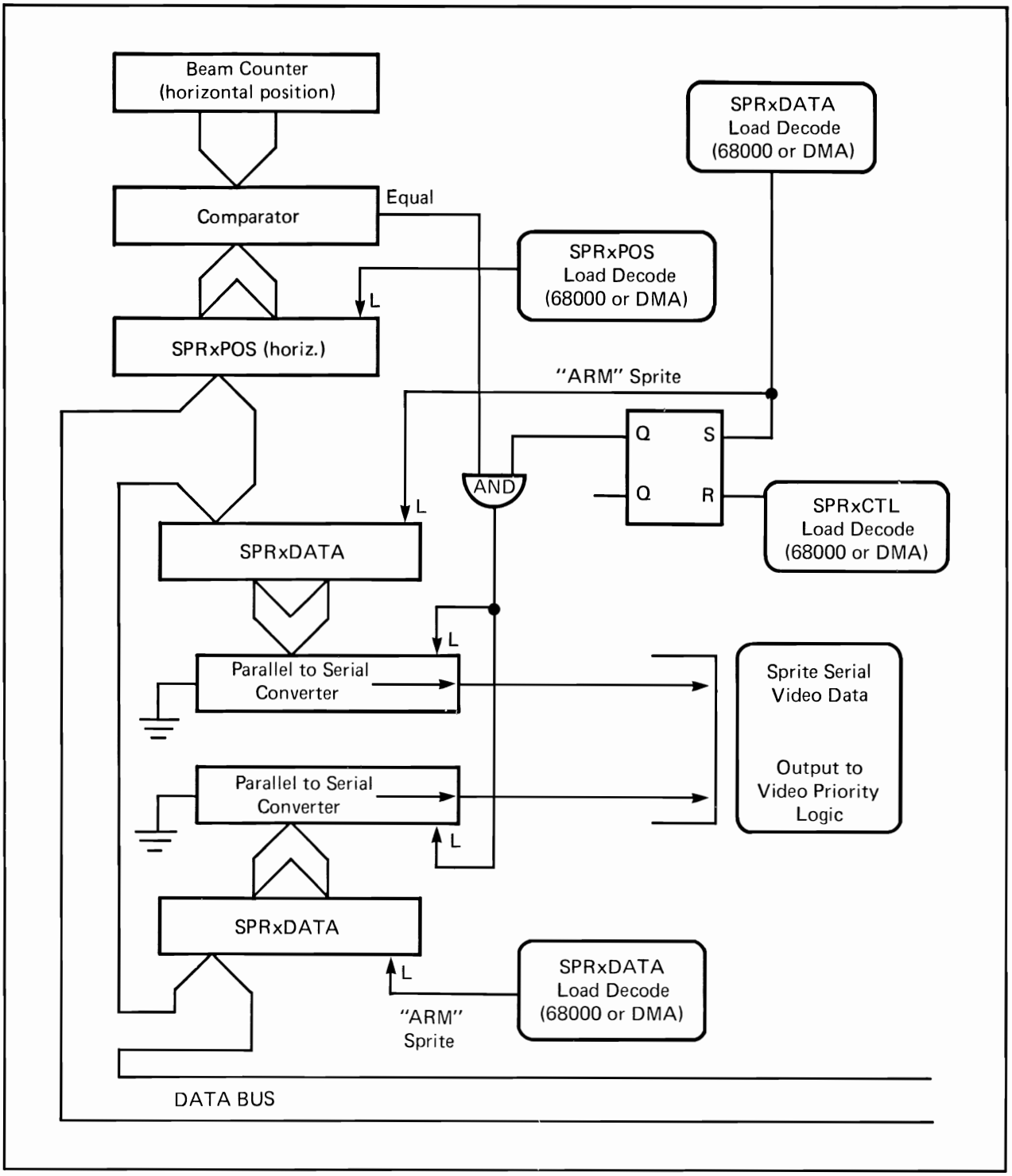


Figure 4-13: Sprite Control Circuitry

Figure 4-13 shows the following:

- writing to the sprite control registers disables the horizontal comparator circuitry. This prevents the system from sending any output from the data registers to the serial converter or to the screen.

- writing to the sprite A data register enables the horizontal comparator. This enables output to the screen when the horizontal position of the video beam equals the horizontal value in the position register.

- If the comparator is enabled, the sprite data will be sent to the display, with the leftmost pixel of the sprite data placed at the position defined in the horizontal part of SPRxPOS.

- As long as the comparator remains enabled, the current contents of the sprite data register will be output at the selected horizontal position on a video line.

- The data in the sprite data registers does not change. It is either rewritten by the user, or modified under DMA control.

This is how the components described above produce the automatic DMA display. When the sprites are in DMA mode, the 18-bit sprite pointer register (composed of SPRxPTH and SPRxPTL) is used to read the first two words from the sprite data structure. These words contain the starting and stopping position of the sprite. Next, the pointers write these words into SPRxPOS and SPRxCTL. After this write, the value in the pointers points to the address of the first data word (low word of data for line 1 of the sprite.)

Writing into the SPRxCTL register disabled the sprite. Now the sprite DMA channel will wait until the vertical beam counter value is the same as the data in the VSTART (Y value) part of SPRxPOS. When these values match, the system enables the sprite data access.

The sprite DMA channel examines the contents of VSTOP (from SPRxCTL) (location of the line after the last line of the sprite) and VSTART (from SPRxPOS) to see how many lines of sprite data are to be fetched. Two words are fetched per line of sprite height and written into the sprite data registers. The first word is stored in SPRxDATA and the second word in SPRxDATB.

The fetch and store for each horizontal scan line occurs during a horizontal blanking interval, far to the left of the start of the screen display. This arms the sprite horizontal comparators and allows them to start the output of the sprite data to the screen when the horizontal beam count values matches the value stored in the HSTART (X value) part of SPRxPOS.

If the count of VSTOP - VSTART equals zero, no sprite output occurs. The next data word pair will be fetched, but it will not be stored into the sprite data registers. It will instead become the next pair of data words for SPRxPOS and SPRxCTL.

When a sprite is used only once within a single display field, the final pair of data words, which follow the sprite color descriptor words, are loaded automatically as the next contents of the SPRxPOS and SPRxCTL registers. To stop the sprite after that first data set, the pair of words should be all zeros.

Thus, if you have formed a sprite pattern in memory, this same pattern will be produced as pixels line-by-line automatically under DMA control.

4.11. SUMMARY OF SPRITE REGISTERS

There are 8 complete sets of registers used to describe the sprites. Each set consists of 5 registers. Only the registers for sprite 0 are described here. All of the others are the same except for the name of the register, which includes the appropriate number.

4.11.1. Pointers

Pointers are registers that are used by the system to point to the current data being used. During a screen display, the registers are incremented to point to the data being used as the screen display progresses. Therefore, pointer registers must be freshly written during the start of the vertical blanking period.

SPR0PTH and SPR0PTL
: This pair of registers contains the 18-bit word address of Sprite 0 DMA data. These registers contain the high 3 bits and low 15 bits of the address, respectively. Because these two register addresses are contiguous, 68000 programmers can write a long word into SPR0PTH, as usual.

Pointer register names for the other sprites are:

SPR1PTH	SPR1PTL
SPR2PTH	SPR2PTL
SPR3PTH	SPR3PTL
SPR4PTH	SPR4PTL
SPR5PTH	SPR5PTL
SPR6PTH	SPR6PTL
SPR7PTH	SPR7PTL

4.11.2. Control Registers

SPR0POS
: Sprite 0 position register.

  The word written into this register controls the position on the screen at which the upper left-hand corner of the sprite is to be placed. The most significant bit of the first data word will be placed in this position on the screen. Note that the sprites have a placement resolution on a full screen of 320 by 200. The sprite resolution is independent of the bit-plane resolution.

  Bit positions:

15-8 specify the vertical start position, bits V7 - V0.

7-0 specify the horizontal start position, bits H8 - H1.

NOTE: This register is normally only written by the sprite DMA channel itself. See the details above regarding the organization of the sprite data. This register is usually updated directly by DMA.

SPR0CTL

This register is normally only used by the sprite DMA channel. It contains control information which is used to control the sprite data fetch process.

Bit positions:

Bits 15-8 specify vertical stop position for a sprite image, bits V7 - V0.

Bit 7 Attach bit.

This bit is only valid for odd-numbered sprites. It indicates that sprites 0, 1 (or 2,3 or 4,5 or 6,7) will, for color interpretation, be considered as paired, and as such be called 4 bits deep. The odd-numbered (higher number) sprite contains bits with the higher binary significance.

During attach mode, the attached sprites are normally moved horizontally and vertically together under processor control.

This allows a greater selection of colors within the boundaries of the sprite itself. The sprites, although attached, remain capable of independent motion; however, and they will assume this larger color set only when their edges overlay one another.

Bits 6-3 Reserved for future use (make zero).

Bit 2 Bit V8 of vertical start.

Bit 1 Bit V8 of vertical stop.

Bit 0 Bit H0 of horizontal start.

Other sprite position and control registers:

SPR1POS	SPR1CTL
SPR2POS	SPR2CTL
SPR3POS	SPR3CTL
SPR4POS	SPR4CTL
SPR5POS	SPR5CTL
SPR6POS	SPR6CTL
SPR7POS	SPR7CTL

4.11.3. Data registers

The following registers, although defined in the address space of the main processor, are normally used only by the display processor. They are the holding registers for the data obtained by DMA cycles.

SPR0DATA, SPR0DATB	data registers for Sprite 0
SPR1DATA, SPR1DATB	data registers for Sprite 1
SPR2DATA, SPR2DATB	data registers for Sprite 2
SPR3DATA, SPR3DATB	data registers for Sprite 3
SPR4DATA, SPR4DATB	data registers for Sprite 4
SPR5DATA, SPR5DATB	data registers for Sprite 5
SPR6DATA, SPR6DATB	data registers for Sprite 6
SPR7DATA, SPR7DATB	data registers for Sprite 7

4.12. SUMMARY OF SPRITE COLOR REGISTERS

Sprite data words are used to select the color of the sprite pixels from the system color register set as indicated in the following tables.

If the bit-combinations from single sprites are the following, then the color will be taken from the registers shown.

SINGLE SPRITES		COLOR
Sprite	Value	REGISTER
0 or 1	00	Not used *
	01	17
	10	18
	11	19
2 or 3	00	Not used *
	01	21
	10	22
	11	23
4 or 5	00	Not used *
	01	25
	10	26
	11	27
6 or 7	00	Not used *
	01	29
	10	30
	11	31

* Selects transparent mode.

If the bit-combinations from attached sprites are the following then the color will be taken from the register shown.

ATTACHED SPRITES 0,1 or 2,3 or 4,5 or 6,7	COLOR REGISTER
0000	Not used *
0001	17
0010	18
0011	19
0100	20
0101	21
0110	22
0111	23
1000	24
1001	25
1010	26
1011	27
1100	28
1101	29
1110	30
1111	31

* Selects transparent mode.

Chapter 5

AUDIO HARDWARE

5.1. INTRODUCTION

This chapter shows you how to directly access the audio hardware to produce sounds. The major topics in this chapter are:

- A brief overview in this section of how a computer produces sound
- How to produce simple steady and changing sounds and more complex ones
- How to use the audio channels for special effects, wiring them for stereo sound if desired, or using one channel to modulate another
- How to produce quality sound within the system limitations

At the end of the chapter, you will find an appendix that gives you values to use for creating musical notes on the equal-tempered musical scale.

This chapter is not a tutorial on computer sound synthesis; to thoroughly describe how to create sound on a computer would require a far longer document. We can only point the way and show you how to use the Amiga's features. Computer sound production is fun, but complex, and usually requires a great deal of trial and error on the part of the user — you use the instructions to create some sound and play it back, readjust the parameters and play it again, and so on.

The following works are recommended for more information on creating music with computers.

- *Introduction to Computer Music* by Wayne A. Bateman (John Wiley and Sons, New York: 1980).
- *Musical Applications of Microprocessors* by Hal Chamberlain (Hayden: 1980).

Introducing Sound Generation

Sound travels through air to your ear drums as a repeated cycle of air pressure variations, or sound waves. Sounds can be represented as graphs that model how the air pressure varies

over time. The attributes of a sound, as you hear it, is related to the shape of the graph. If the waveform is regular and repetitive, it will sound like a tone with steady pitch (highness or lowness), such as a single musical note. Each repetition of a waveform is called a cycle of the sound. If the waveform is irregular, the sound will have little or no pitch, like a loud clash or rushing water. How often the waveform repeats (its frequency), has an effect upon its pitch; sounds with higher frequencies are higher in pitch. Humans can hear sounds that have a frequency of between 20 and 20,000 cycles per second. The amplitude of the waveform (highest point on the graph), is related to the perceived loudness of the sound. Finally, the general shape of the waveform determines its tone quality, or timbre. Figure 5-1 shows a particular kind of waveform, called a sine wave, that represents one cycle of a simple tone.

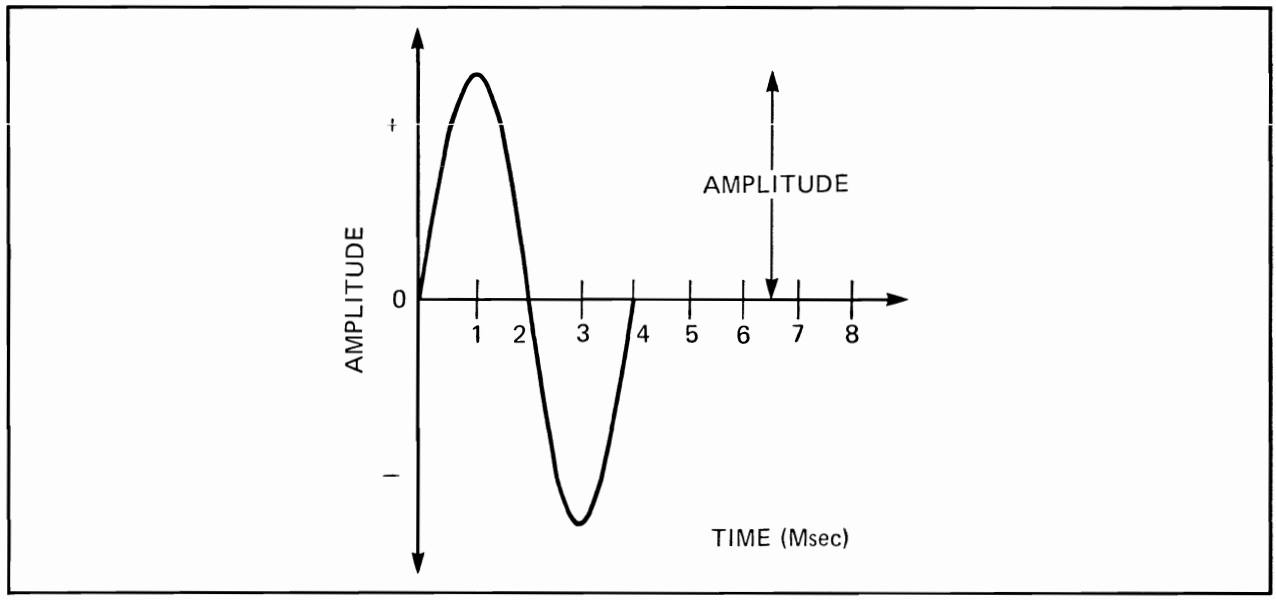


Figure 5-1: Sine Waveform

In electronic sound recording and output devices, the attributes of sounds are represented by the parameters of amplitude and frequency. Frequency is the number of cycles per second, and the most common unit of frequency is the Hertz (Hz), which is 1 cycle per second. Large values, or high frequencies, are measured in kilohertz (kHz) or megahertz (mHz).

Frequency is strongly related to the perceived pitch of a sound. When frequency increases, pitch increases and the sound gets higher. This relationship is exponential. An increase from 100 Hz to 200 Hz results in a large increase in pitch, but an increase from 1,000 Hz to 1,100 Hz is hardly noticeable. Musical pitch is represented in octaves. If a tone is one octave

higher than another, then the higher tone has a frequency twice as high and the perceived pitch is twice as high.

The second parameter that defines a waveform is its amplitude. In an electronic circuit, amplitude relates to the voltage or current in the circuit. When a signal is going to a speaker, the amplitude is expressed in watts. Perceived sound intensity is measured in decibels (db). Human hearing has a range of about 120 db; 1 db is the faintest audible sound. Roughly every 10 db corresponds to a doubling of sound, and 1 db is the smallest change in amplitude that is noticeable in a moderately loud sound. Volume, which is the amplitude of the sound signal which is output, corresponds logarithmically to decibel level.

The frequency and amplitude parameters of a sine wave are completely independent. When sound is heard, however, there is interaction between loudness and pitch. Lower-frequency sounds decrease in loudness much faster than high-frequency sounds.

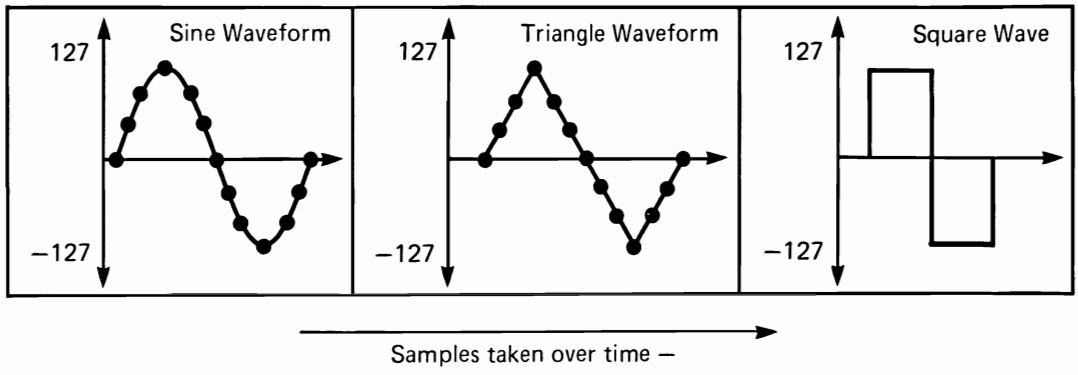
The third attribute of a sound, timbre, depends on the presence or absence of overtones, or harmonics. Any complex waveform is actually a mixture of sine waves of different amplitudes, frequencies, and phases (the starting point of the waveform on the time axis). These component sine waves are called harmonics. A square waveform, for example, has an infinite number of harmonics.

In summary, all steady sounds can be described by their frequency, overall amplitude, and relative harmonic amplitudes. The audible equivalents of these parameters are pitch, loudness, and timbre, respectively. Changing sound is a steady sound whose parameters change over time.

In electronic production of sound, an analog device, such as a tape recorder, records sound waveforms and their cycle frequencies as a continuously variable representation of air pressure. The tape recorder then plays back the sound by sending the waveforms to an amplifier where they are changed into analog voltage waveforms. The amplifier sends the voltage waveforms to a loudspeaker, which translates them into air pressure vibrations that the listener perceives as sound.

A computer cannot store analog waveform information. In computer production of sound, a waveform has to be represented as a finite string of numbers. To do this, the time axis of the graph of a single waveform is divided into equal segments, each of which represents a short enough time so the waveform does not change a great deal. Each of the resulting points is called a sample. These samples are stored in memory and you can play them back at a frequency you determine. The computer feeds the samples to a digital-to-analog converter (DAC), which changes them into an analog voltage waveform. To produce the sound, the analog waveforms are sent first to an amplifier, then to a loudspeaker.

Figure 5-2 shows an example of a sine wave, a square wave, and a triangle wave, along with a table of samples for each. Note that the illustrations are not to scale and there are fewer dots in the wave forms than there are samples in table. The amplitude axis values 127 and -128 represent the high and low limits on relative amplitude.



TIME	SINE	SQUARE	TRIANGLE
0	0	100	0
1	39	100	20
2	75	100	40
3	103	100	60
4	121	100	80
5	127	100	100
6	121	100	80
7	103	100	60
8	75	100	40
9	39	100	20
10	0	−100	0
11	−39	−100	−20
12	−75	−100	−40
13	−103	−100	−60
14	−121	−100	−80
15	−127	−100	−100
16	−121	−100	−80
17	−103	−100	−60
18	−75	−100	−40
19	−39	−100	−20

Figure 5-2: Digitized Amplitude Values

The Amiga Sound Hardware

The Amiga has 4 hardware sound channels. You can independently program each of the channels to produce complex sound effects. You can also attach channels so that one channel modulates the sound of another, or combine two channels for stereo effects.

Each audio channel includes an 8-bit digital-to-analog converter driven by a direct memory access (DMA) channel. The audio DMA can retrieve two data samples during each horizontal video scan line. For simple, steady tones, the DMA can automatically play a waveform repeatedly, or you can program all kinds of complex sound effects.

There are two methods of basic sound production on the Amiga — automatic (DMA) sound generation and direct (non-DMA) sound generation. When you use automatic sound generation, the system retrieves data automatically by direct memory access.

5.2. FORMING AND PLAYING A SOUND

This section shows you how to create a simple, steady sound and play it. Many basic concepts that apply to all sound generation in the Amiga are introduced in this section.

Following are the basic steps in producing a steady tone:

1. Decide which channel to use.
2. Define the waveform and create the sample table in memory.
3. Set registers telling the system where to find the data and the length of the data.
4. Select the volume at which the tone is to be played.
5. Select the sampling period, or output rate of the data.
6. Select an audio channel and start up the DMA.

5.2.1. Deciding Which Channel to Use

The Amiga has 4 audio channels. Channels 0 and 3 are connected to the left side stereo output jack. Channels 1 and 2 are connected to the right side output jack. Select a channel on the side from which the output is to appear.

5.2.2. Creating the Waveform Data

The waveform used as an example in this section is a simple sine wave, which produces a pure tone. To conserve memory, you normally define only one full cycle of a waveform in memory. For a steady, unchanging sound, the values at the waveform's beginning and ending points, and the trend or slope of the data at the beginning and end should be closely related. This ensures that a continuous repetition of the waveform sounds like a continuous stream of sound.

Sound data is organized as a set of 8-bit data items; each item is a sample from the waveform. Each data word retrieved for the audio channel consists of two samples. Sample values can range from -128 to +127.

As an example, the data set shown below produces a close approximation to a sine wave. Note that the data is stored in byte address order with the first digitized amplitude value at the lowest byte address, and second at the next byte address, and so on. Also, note that the first byte of data must start at a word-address boundary. This is because the audio DMA retrieves one word (16 bits) at a time, and uses the sample it reads as two bytes of data.

To use audio channel 0, write the address of "audiodata" into AUD0LC, where the audio data is organized as shown below. For simplicity, "AUDxLC" in the table below stands for the combination of the two actual location registers (AUDxLCH and AUDxLCL). For the audio DMA channels to be able to retrieve the data, the data address to which AUD0LC points must be located in the low 512 Kbytes of RAM.

Table 5-1: Sample Audio Data Set for Channel 0

audiodata --->	AUD0LC	100	98
	AUD0LC + 2	92	83
	AUD0LC + 4	71	56
	AUD0LC + 6	38	20
	AUD0LC + 8	0	-20
	AUD0LC + 10	-38	-56
	AUD0LC + 12	-71	-83
	AUD0LC + 14	-92	-83
	AUD0LC + 16	-100	-98
	AUD0LC + 18	-92	-83
	AUD0LC + 20	-71	-56
	AUD0LC + 22	-38	-20
	AUD0LC + 24	0	20
	AUD0LC + 26	38	56
	AUD0LC + 28	71	83
	AUD0LC + 30	92	98

NOTES:

1. Audio data is located on a word-address boundary.
2. AUD0LC stands for AUD0LCL and AUD0LCH.

5.2.3. Telling the System About the Data

In order to retrieve the sound data for the audio channel, the system needs the following information:

- Where is the data located?
- How long (in words) is the data?

The location registers AUDxLCH and AUDxLCL contain the high 3 bits and low 15 bits respectively of the starting address of the audio data. Since these two register addresses are contiguous, writing a long word into AUDxLCH moves the audio data address into both locations. The "x" in the register names stands for the number of the audio channel where the output will occur. The channels are numbered 0, 1, 2, and 3.

These registers are location registers, as distinguished from pointer registers. You only need to specify the contents of these registers once; no resetting is necessary when you wish the audio channel to keep on repeating the same waveform. Each time the system retrieves the last audio word from the data area, it uses the contents of these location registers to again find the start of the data. Assuming the first word of data starts at location "audiodata" and you are using channel 0, here is how to set the location registers:

```
                AUD0LC equ AUD0LCH      ;make AUD0LC stand for AUD0LCL
where0data:     lea audiodata, a0
                move.l a0, AUD0LC       ;put address (32 bits)
                                        ;into location register.
```

The length of the data is the number of samples in your waveform divided by 2, or the number of words in the data set. Using the sample data set above, the length of the data is 16 words. You write this length into the audio data length register for this channel. The length register is called AUDxLEN, where "x" refers to the channel number. You set the length register AUD0LEN to 16 as shown below.

```
setaud0length:    move.w #16, AUD0LEN
```

5.2.4. Selecting the Volume

The volume you set here is the overall volume of all the sound coming from the audio channel. The relative loudness of sounds, which will concern you when you combine notes, is determined by the amplitude of the wave form. There is a 6-bit volume register for each audio channel. To control the volume of sound that will be output through the selected audio channel, you write the desired value into the register AUDxVOL, where "x" is replaced by the channel number. You can specify values from 64 to 0. These volume values correspond to decibel levels. The appendix to this chapter contains a table showing the decibel value for each of the 65 volume levels. For a typical output at volume 64, with maximum data values of -128 to 127, the voltage output is between +.4 volts and -.4 volts. Some volume levels and the corresponding decibel values are shown below.

Table 5-2: Volume Values

Volume	Decibel Value	
64	0	(maximum volume)
48	-2.5	
32	-6.0	
16	-12.0	(12 db down from the volume at maximum level)

For any volume setting from 64 to 0, you write the value into bits 5-0 of AUD0VOL. For example:

```
set aud0volume:    move.w #48, AUD0VOL
```

The decibels are shown as negative values from a maximum of 0 because this is the way a recording device, such as a tape recorder, shows the recording level. Usually, the recorder

has a dial showing 0 as the optimum recording level. Anything less than the optimum value is shown as a minus quantity.

5.2.5. Selecting the Data Output Rate

The pitch of the sound produced by the waveform depends upon its frequency. To tell the system what frequency to use, you need to specify the sampling period. The sampling period specifies the number of system clock ticks, or timing intervals that should elapse between each sample (byte of audio data) fed to the digital-to-analog converter in the audio channel. There is a period register for each audio channel. The contents of the period register is used as a count-down value; each time the register counts down to 0, another sample is retrieved from the waveform data set for output. In units, the period value represents clock ticks per sample. The minimum period value you should use is 124 ticks per sample and the maximum is 65535. For quality sound, there are other constraints on the sampling period (see the section called "Producing Quality Sound"). Note that a low period value corresponds to a higher frequency sound and a high period value corresponds to a lower frequency sound.

Limitations on Selection of Sampling Period

The sampling period is limited by the number of DMA cycles allocated to an audio channel. Each audio channel is allocated one DMA slot per horizontal scan line of the screen display. An audio channel can retrieve two data samples during each horizontal scan line. The following calculation gives the maximum sampling rate in samples per second.

2 samples/line X 262.5 lines/frame X 59.94 frames/second = 31,469 samples/second

The figure of 31,469 is a theoretical maximum. In order to save buffers, the hardware is designed to handle 28,867 samples/second. The system timing interval is 279.365 nanoseconds, or .279365 microseconds. The maximum sampling rate of 28,867 samples per second is 34.642 microseconds per sample (1/28,867 = .000034642). The formula for calculating the sampling period is:

Period value = sample interval/clock interval

so the minimum period value is derived by dividing 34.642 microseconds per sample by the number of microseconds per interval:

$$\text{Minimum period} = \frac{34.642 \text{ microseconds/sample}}{0.279365 \text{ microseconds/interval}}$$

$$= 124 \text{ timing intervals/sample}$$

Therefore, a value of at least 124 must be written into the period register to assure that the audio system DMA will be able to retrieve the next data sample. If the period value is below 124, by the time the cycle count has reached 0, the audio DMA will not have had enough time to retrieve the next data sample and the previous sample will be reused.

Specifying the Period Value

If you have selected the desired interval between data samples, then you can calculate the value to place in the period register by using the period formula:

Period value = desired interval/clock interval

As an example, let's say you wanted to produce a 1 kHz sine wave, using a table of 8 data samples (4 data words):

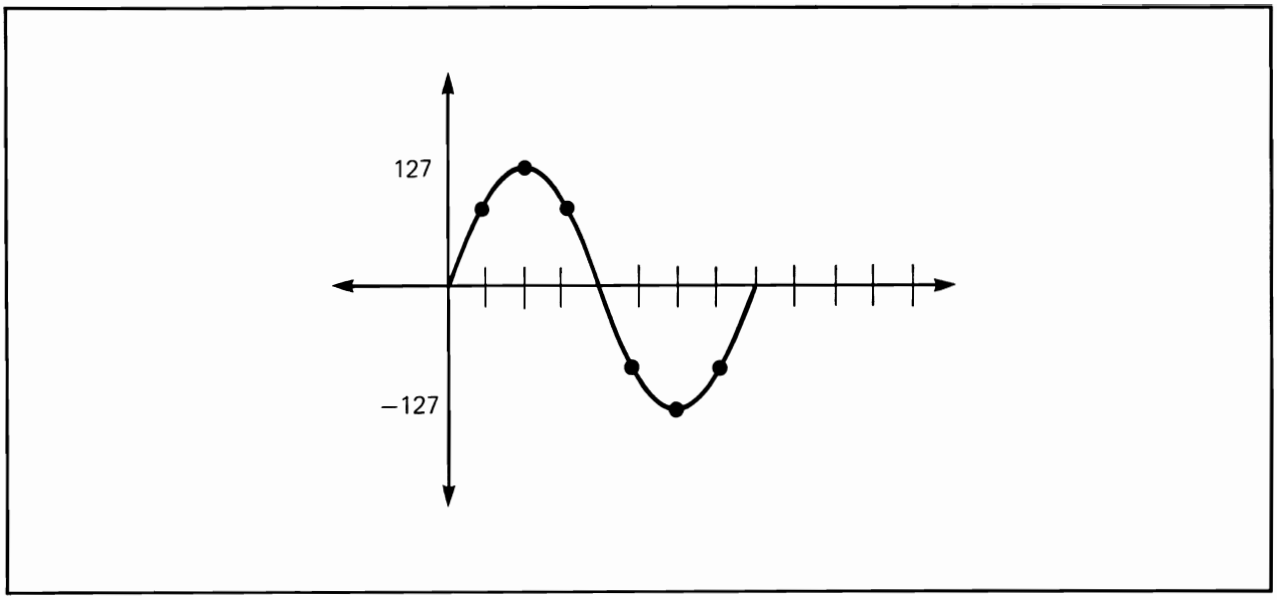


Figure 5-3: Example Sine Wave

Sampled Values:	0
	90
	127
	90
	0
	-90
	-127
	-90

To output the series of 8 samples at 1 kHz (1000 cycles per second), each full cycle is output in 1/1000th of a second. Therefore, each individual value must be retrieved in 1/8th of that time. This translates to 1,000 microseconds per waveform or 125 microseconds per sample. To correctly produce this waveform, the period value should be:

$$\text{Period value} = \frac{\text{125 microseconds/sample}}{\text{0.279365 microseconds/interval}}$$

$$= \quad \text{447 timing intervals/sample}$$

To set the period register, you write the period value into the register AUDxPER, where "x" is the number of the channel you are using. For example, the following instruction shows how to write a period value of 447 into the period register for channel 0.

```
setaud0period:    move.w #447, AUD0PER
```

To produce quality sound, you should observe some limitations on period to avoid aliasing distortion. See the section below called "Producing Quality Sound".

For the relationship between period and musical pitch, see the appendix to this chapter, which contains a listing of the equal-tempered musical scale.

5.2.6. Playing the Waveform

After you have defined the audio data location, length, volume and period, you can play the waveform by starting the DMA for that audio channel. This starts the output of sound. Once started, the DMA continues until you specifically stop it. Thus, the waveform gets played over and over again, producing the steady tone. The system uses the value in the location registers each time it replays the waveform.

To start the channel, you write a 1 into the AUDxEN bit of the DMA control register named DMACON. To start the DMA, you write a 1 into the DMAEN bit of DMACON. All these bits and their meanings are shown in the table below:

Table 5-3: DMA and Audio Channel Enable Bits

DMACON REGISTER

BIT	NAME	FUNCTION
15	SETCLR	When this bit is written as a 1, it sets any bit in DMACONW for which the corresponding bit position is also a 1, leaving all other bits alone.
9	DMAEN	Only while this bit is a 1 can *any* direct memory access occur.
3	AUD3EN	Audio channel 3 enable
2	AUD2EN	Audio channel 2 enable
1	AUD1EN	Audio channel 1 enable
0	AUD0EN	Audio channel 0 enable

For example, if you are using channel 0, then you write a 1 into bit 9 to enable DMA and a 1 into bit 0 to enable the audio channel, as shown below.

```
SET         equ    $08000
AUD0EN      equ    $01
DMAEN       equ    $0200

beginchan0:     move.w #(SET + AUD0EN + DMAEN), DMACONW
```

5.2.7. Stopping the Audio DMA

You can stop the channel by writing a 0 into the AUDxEN bit at any time. However, you cannot resume the output at the same point in the waveform by just writing a 1 in the bit again. Enabling an audio channel almost always starts the data output again from the top of the list of data pointed to by the location registers for that channel. If the channel is disabled for a very short time (less than two sampling periods) it may stay on and thus continue from where it left off.

Here is how to stop audio DMA:

```
                CLEAR  equ 0
stopaudchan0:   move.w #(CLEAR + AUD0EN), DMACONW    ;Stop only this
                                                     ;channel's DMA
```

5.2.8. Summary

These are the steps necessary to produce a steady tone:

1 Define the waveform

2 Create the data set containing the pairs of data samples (data words). Normally, a data set contains the definition of one waveform.

3 Set the location registers:

AUDxLCH (high 3 bits)
AUDxLCL (low 15 bits)

4 Set the length register, AUDxLEN, to the number of data words to be retrieved before starting at the address currently in AUDxLC.

5 Set the volume register, AUDxVOL

6 Set the period register, AUDxPER

7 Start the audio DMA by writing 1 into bit 9, DMAEN, along with a 1 in the SETCLR bit and a 1 in the position of the AUDxEN bit of the channel or channels you want to start.

5.2.9. Example

This example gathers together all of the program segments from the preceding sections.
In this example, a sine wave is played through channel 0.

```
AUD0LC    equ   AUD0LCH
SET       equ   $08000
CLEAR     equ   0
AUD0EN    equ   $01
DMAEN     equ   $0200
ds.w 0                          ;be sure word-aligned

sinedata:  dc.b 0, 90, 127, 90, 0, -90, -127, -90

main:
           lea sinedata, a0          ;address of data
                                     ;to audio location register 0
where0data:
           move.l a0, AUD0LC         ;note that the 68000 can
                                     ;write this as though it is
                                     ;a 32-bit register at the low-bits
                                     ;location (common to all locations
                                     ;and pointer registers in the system).

setaud0length:
           move.w #4, AUD0LEN        ;set length in words

setaud0volume:
           move.w #64, AUD0VOL       ;let's use maximum volume

setaud0period:
           move.w #447, AUD0PER

beginchan0:
           move.w #(SET + DMAEN + AUD0EN), DMACONW

           end
```

5.3. PRODUCING COMPLEX SOUNDS

You can create sounds that are more complex, for example, different musical notes joined into a one-voice melody, different notes played at the same time, or modulated sounds.

5.3.1. Joining Tones

You join tones by writing the location and length registers, starting the audio output, and rewriting the registers in preparation for the next audio waveform that you wish to connect to the first one. This is made easy by the timing of the audio interrupts and the existence of backup registers. The location and length registers are read by the DMA channel before audio output begins. The DMA channel then stores the values in backup registers. Once the original registers have been read by the DMA channel, you can change their values without disturbing the operation you started with the original register contents. Thus you can write the contents of these registers, start an audio output, and then rewrite the registers in preparation for the next waveform you want to connect to this one.

Interrupts occur immediately after the audio DMA channel has read the location and length registers and stored their values in the backup registers. Once the interrupt has occurred, you can rewrite the registers with the location and length for the next waveform segment. This combination of backup registers and interrupt timing lets you always be one step ahead of the audio DMA channel, allowing your sound output to be continuous and smooth.

If you do not rewrite the registers, the current waveform will be repeated. Each time the length counter reaches zero, both the location and length registers are reloaded with the same values to continue the audio output.

Example

This example details the system audio DMA action in a step-by-step fashion.

Suppose you wanted to join together a sine and a triangle waveform, end-to-end, for a special audio effect, alternating between them. Here is a sequence which shows the action of your program as well as its interaction with the audio DMA system. The example assumes that the period, volume, and length of the data set remains the same for both the sine wave and the triangle wave.

Your Interrupt Response

If (wave = triangle)
write AUD0LCL with address of sine wave data
Else if (wave = sine)
write AUD0LCL with address of triangle wave data

Main Program

1. Set up volume, period, and length.
2. Write AUD0LCL with address of sine wave data.
3. Start DMA.
4. Continue with something else.

System Response

As soon as DMA starts,

a. Copy to "backup" length register from AUD0LEN
b. Copy to "backup" location register from AUD0LCL (will be used as a pointer showing current data word to fetch)
c. Create an interrupt for the 68000 saying that it has completed retrieving working copies of length and location registers.
d. Start retrieving audio data each allocated DMA time slot.

5.3.2. Playing Multiple Tones at the Same Time

You can play multiple tones either by using several channels independently or by summing the samples in several data sets, playing the summed data sets through a single channel.

Since all 4 audio channels are independently programmable, each channel has its own data set and can be playing a different tone or musical note on each channel.

5.3.3. Modulating Sound

To provide more complex audio effects, you can use one audio channel to modulate another. This increases the range and type of effects that can be produced. You can modulate a channel's frequency or amplitude, or do both types of modulation on a channel at the same time.

Amplitude modulation affects the volume of the waveform. It is often used to produce vibrato or tremolo effects. Frequency modulation affects the period of the waveform. Although the basic waveform itself remains the same, the pitch is increased or decreased by frequency modulation.

The system uses one channel to modulate another when you attach two channels. The attach bits in the ADKCON register control how the data from an audio channel is interpreted (see the table below). Normally, each channel produces sound when it is enabled. If the "attach" bit for an audio channel is set, that channel ceases to produce sound and its data is used to modulate the sound of the next higher-numbered channel. When a channel is used as a modulator, the words in its data set are no longer treated as two individual bytes. Instead, they are are treated as "modulator" words. The data words from the modulator channel are written into the corresponding registers of the modulated channel each time the period register of the modulator channel times out.

To modulate only the amplitude of the audio output, you attach a channel as a volume modulator. You define the modulator channel's data set as a series of words, each containing volume information in the following format:

BITS	FUNCTION
15 - 7	Not used
6 - 0	Volume information, V6 - V0

To modulate only the frequency, you attach a channel as a period modulator. You define the modulator channel's data set as a series of words, each containing period information in the following format:

BITS	FUNCTION
15 - 0	Period information, P15 - P0

If you want to modulate both period and volume on the same channel, you need to attach the channel as both a period and volume modulator. For instance, if channel 0 is used to modulate both the period and frequency of channel 1, you set two attach bits — bit 0 to modulate the volume and bit 4 to modulate the period. When period and volume are both modulated, words in the modulator channel's data set are defined alternately as volume and

period information.

The sample set of data below shows the differences in interpretation of data when a channel is used directly for audio, when it is attached as volume modulator, when it is attached as a period modulator, and when it is attached as a modulator of both volume and period.

Table 5-4: Data Interpretation in Attach Mode

	Interpretation of Data Words			
Data Words	*Independent (not Modulating)*	*Modulating Both Period and Volume*	*Modulating Only Period*	*Modulating Only Volume*
Word 1	| data | data |	| volume for other channel |	| period |	| volume |
Word 2	| data | data |	| period for other channel |	| period |	| volume |
Word 3	| data | data |	| volume for other channel |	| period |	| volume |
Word 4	| data | data |	| period for other channel |	| period |	| volume |

The lengths of the data sets of the modulator and the modulated channels are completely independent.

Channels are attached by the system in a pre-determined order, as shown in the table below. To attach a channel as a modulator, you set its attach bit to 1. If you set either the volume or period attach bits for a channel, that channel's audio output will be disabled and it will be attached to the next higher channel as shown in the table above. Since an attached channel always modulates the next higher numbered channel, you cannot attach channel 3. Writing a 1 into one of its modulate bits only disables its audio output.

Table 5-5: Channel Attachment for Modulation

ADKCON REGISTER

BIT	NAME	FUNCTION
7	ATPER3	Use audio channel 3 to modulate nothing (disables audio output of channel 3)
6	ATPER2	Use audio channel 2 to modulate *period* of channel 3
5	ATPER1	Use audio channel 1 to modulate *period* of channel 2
4	ATPER0	Use audio channel 0 to modulate *period* of channel 1
3	ATVOL3	Use audio channel 3 to modulate nothing (disables audio output of channel 3)
2	ATVOL2	Use audio channel 2 to modulate *volume* of channel 3
1	ATVOL1	Use audio channel 1 to modulate *volume* of channel 2
0	ATVOL0	Use audio channel 0 to modulate *volume* of channel 1

5.4. PRODUCING QUALITY SOUND

When trying to create quality sound, you need to consider the following factors:

- o waveform transitions
- o sampling rate
- o efficiency
- o noise reduction
- o avoidance of aliasing distortion
- o limitations of the low pass filter

5.4.1. Making Waveform Transitions

To avoid unpleasant sounds when you change from one waveform to another, you need to make the transitions smooth.

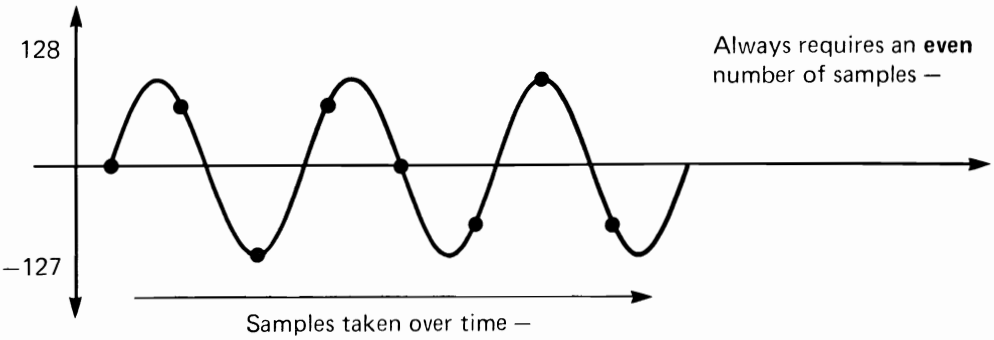
You can avoid "clicks" by making sure the waveforms start and end at approximately the same value. You can avoid "pops" by only starting a waveform at a zero-crossing point.

You can avoid "thumps" by arranging the average amplitude of each wave to about the same value. The average amplitude is the sum of the bytes in the waveform divided by the number of bytes in the waveform.

5.4.2. Sampling Rate

If you need high precision in your frequency output, you may find that the frequency you wish to produce is somewhere between two available sampling rates, but not close enough for your requirements. In those cases, you may have to adjust the length of the audio data table in addition to altering the sampling rate.

For higher frequencies, you may also need to use audio data tables that contain more than one full cycle of the audio waveform to reproduce the desired frequency more accurately, as illustrated in the figure below.



Shows a case where a high frequency waveform may need more than one full cycle to accurately reproduce the periodic waveform.

Figure 5-4: Waveform with Multiple Cycles

5.4.3. Efficiency

There is a certain amount of overhead involved in the handling of audio DMA. If you are trying to produce a smooth continuous audio synthesis, you should try to avoid as much of the system control overhead as possible. Basically, the larger the audio buffer you provide to the system, the less often it will need to interrupt to reset the pointers to the top of the next buffer and, coincidentally, the lower the amount of system interaction will be required. If there is only one waveform buffer, the hardware automatically resets the pointers and there is no software overhead for resetting the pointers.

The "Joining Tones" section demonstrated how you could join "ends" of tones together by responding to interrupts and changing the values of the location registers to splice tones together. If your system is heavily loaded, it is possible that the response to the interrupt might not happen in time to assure a smooth audio transition. Therefore, it is advisable to utilize the longest possible audio table where a smooth output is required. This takes advantage of the audio DMA capability as well as minimizing the number of interrupts to which the 68000 must respond.

5.4.4. Noise Reduction

To reduce noise levels and produce an accurate sound, try to use the full range of -128 to 127 when you represent a waveform. This reduces how much noise (quantization error) will be added to the signal by using more bits of precision. Quantization noise is caused by the introduction of round-off error. If you are trying to reproduce a signal, such as a sine wave, you can only represent the amplitude of each sample with so many digits of accuracy. The difference between the real number and your approximation is round-off error, or noise.

By doubling the amplitude, you have half as much noise because the size of the steps of the wave form stays the same and is therefore a smaller fraction of the amplitude. In other words, if you try to represent a waveform using, for example, a range of only +3 to -3, the size of the error in the output would be considerably larger than if you use a range of +127 to -128 to represent the same signal. Proportionally, the digital value used to represent the waveform amplitude will have a lower error. As you increase the number of possible sample levels, you decrease the relative size of each step and, therefore, decrease the size of the error.

To produce quiet sounds, continue to define the waveform using the full range, but adjust the volume. This maintains the same level of accuracy (signal-to-noise ratio) for quiet sounds as for loud sounds.

5.4.5. Aliasing Distortion

When you use sampling to produce a waveform, there is a side effect caused when the sampling rate "beats" or combines with the frequency you wish to produce. This produces two additional frequencies, one at the sampling rate plus the desired frequency and the other at the sampling rate minus the desired frequency. This is called aliasing distortion.

Aliasing distortion is eliminated when the sampling rate exceeds the output frequency by at least 7 kHz. This puts the beat frequency outside the range of the low pass filter, cutting off the undesirable frequencies. Figure 5-5 shows a frequency domain plot of the anti-aliasing low pass filter used in the system.

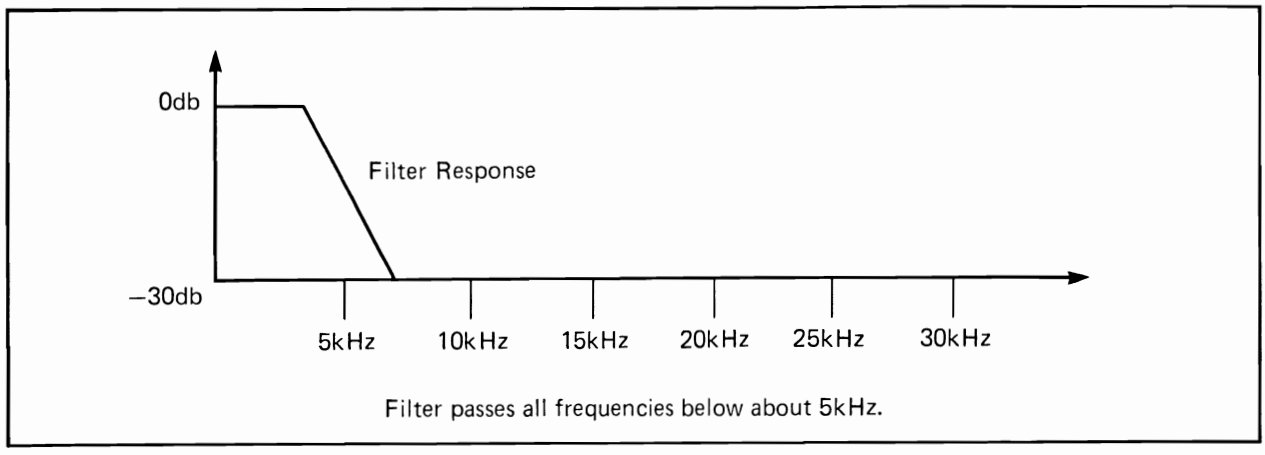


Figure 5-5: Frequency Domain Plot of Low-Pass Filter

Figure 5-6 shows that is is permissible to use a 12 kHz sampling rate to produce a 4 kHz waveform. Both of the beat frequencies are outside the range of the filter, as shown in these calculations:

$12 + 4 = 16$ kHz
$12 - 4 = 8$ kHz

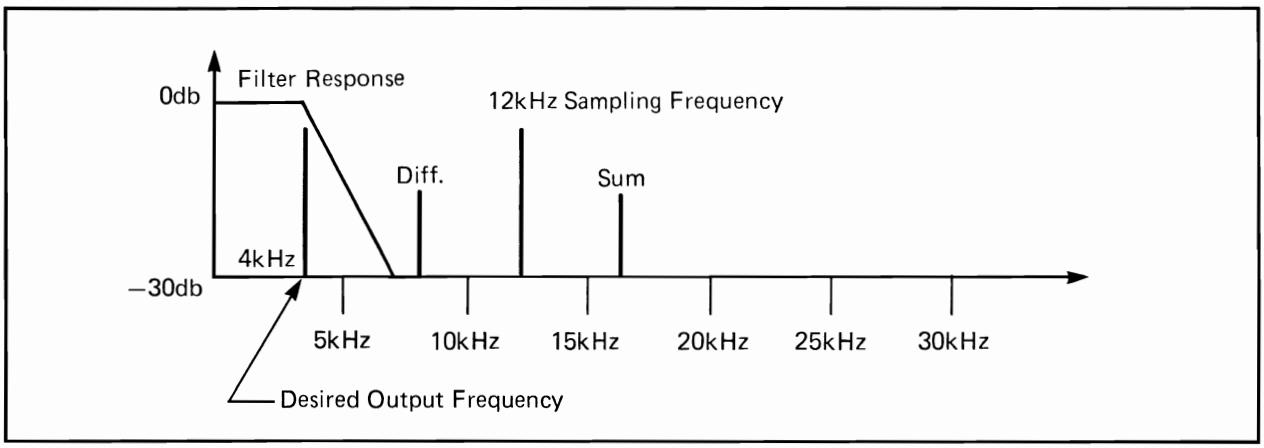


Figure 5-6: Noise-Free Output (No Aliasing Distortion)

You can see in Figure 5-7 that is unacceptable to use a 10 kHz sampling rate to produce a 4 kHz waveform. One of the beat frequencies (10 - 4) is within the range of the filter, allowing some of that undesirable frequency to show up in the audio output.

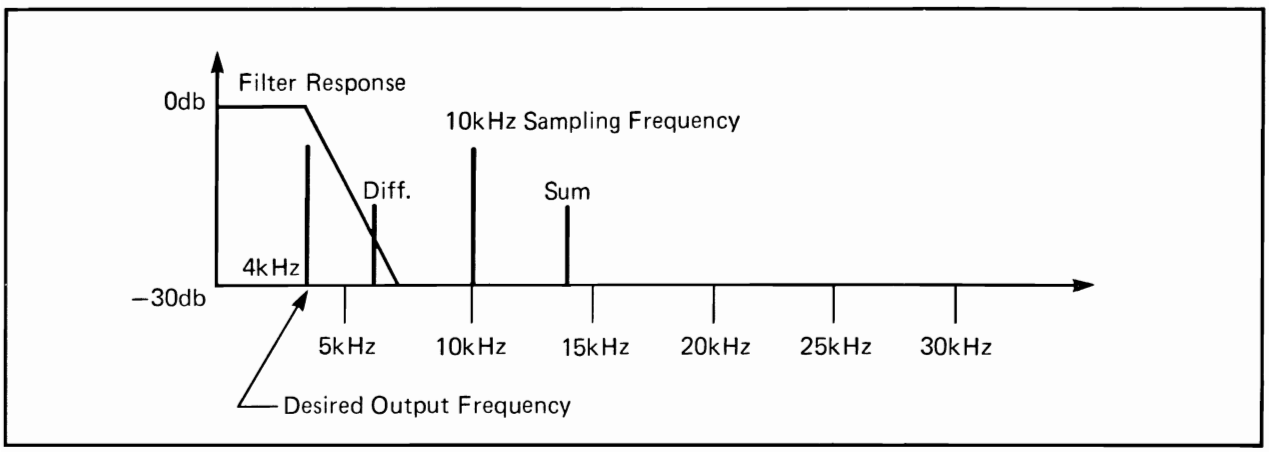


Figure 5-7: Some Aliasing Distortion

All of this gives rise to the following equation, showing that the sampling frequency must exceed the output frequency by at least 7 kHz, so that the difference beat frequency will be above the cutoff range of the anti-aliasing filter:

$$\text{Minimum sampling rate} = \text{Highest Frequency Component} + 7 \text{ kHz}$$

The frequency component of the equation is stated as "Highest Frequency Component" because you may be producing a complex waveform with multiple frequency elements, rather than a pure sine wave.

5.4.6. Low Pass Filter

The system includes a low pass filter that eliminates aliasing distortion as described above. This filter becomes active around 4 kHz and gradually begins to attenuate (cut off) the signal. Generally, you cannot clearly hear frequencies higher than 7 kHz. Therefore, you get the most complete frequency response in the frequency range of 0 - 7 kHz. If you are making frequencies from 0 to 7kHz, you should select a sampling rate no less than 14 kHz, which corresponds to a sampling period in the range 124 to 256.

At a sampling period around 320, you begin to lose the higher frequency values in the 0 to 7 kHz range, as shown in the following table.

	SAMPLING PERIOD	SAMPLING RATE (kHz)	MAXIMUM OUTPUT FREQUENCY (kHz)
Maximum sampling *rate*	124	29	7
Minimum sampling *rate* for 7 kHz output	256	14	7
Sampling rate too low for 7 kHz output	320	11	4

5.5. USING DIRECT (NON-DMA) AUDIO OUTPUT

It is possible to create sound by writing the audio data one word at a time to the audio output addresses, instead of setting up a list of audio data in memory. This method of controlling the output is more processor intensive and is therefore not recommended.

To use direct audio output, you do not enable the DMA for the audio channel you wish to use. This changes the timing of the interrupts. The normal interrupt occurs after a data address has been read. In direct audio output, the interrupt occurs after one data word has been output.

Unlike the DMA-controlled automatic data output, if you do not write a new set of data to the output addresses before two sampling intervals have elapsed, the audio output will cease changing. The last value remains as an output of the digital-to-analog converter.

You set the volume and period registers as usual.

5.6. THE EQUAL-TEMPERED MUSICAL SCALE

This section gives a close approximation of the equal tempered scale. The "Period" column gives the period count you enter into the period register.

See the explanatory notes following this table for determining AUDxLEN value.

Table 5-6: The Equal-Tempered Scale

PERIOD	NOTE	IDEAL FREQUENCY (with AUDxLEN=8)	ACTUAL FREQUENCY (with AUDxLEN=8)
508	A	440.0	440.4
480	A#	466.2	466.1
453	B	493.9	493.9
428	C	523.3	522.7
404	C#	554.4	553.8
381	D	587.3	587.2
360	D#	622.3	621.4
339	E	659.3	659.9
320	F	698.5	699.1
302	F#	740.0	740.8
285	G	784.0	785.0
269	G#	830.6	831.7
254	A	880.0	880.8
240	A#	932.3	932.2
226	B	987.8	989.9
214	C	1046.5	1045.4
202	C#	1108.7	1107.5
190	D	1174.7	1177.5
180	D#	1244.5	1242.9
170	E	1318.5	1316.0
160	F	1396.9	1398.3
151	F#	1480.0	1481.6
143	G	1568.0	1564.5
135	G#	1661.2	1657.2

NOTES:

1. In this scale, the frequency for the note A is 440.0 Hz and A# is the twelfth root of 2 (1.059463) times higher in frequency than A. The note B is the twelfth root of 2 higher than A#. This is followed by C, C#, D, D#, E, F, F#, G, G# and goes back to A at 880.0 Hz, an octave higher, and so on. Use this scale for waveforms where the fundamental is 2 to the n'th bytes long and where n is an integer. For example, for A at 440.0 Hz with a period of 508, the sample table contains 16 samples per cycle:

$$\frac{\text{3579545 clocks/second}}{\text{508 clocks/sample X 440 cycles/second}} = 16 \text{ samples/cycle}$$

$$= 2^{4}$$

$$n = 4$$

It follows that for A at 440.0 Hz with a period of 254, the sample table has to contain 32 samples per cycle (AUDxLEN = 16).

2. The general rule is that doubling the sampling frequency (halving the sampling period) changes the octave of the note being played. Thus, if you play a C at a sampling period of 256, then playing the same note with a sampling period of 128 gives a C an octave higher.
3. Before using the lower octaves in this table, be sure to read the section called "Aliasing Distortion".

5.7. DECIBEL VALUES FOR VOLUME RANGES

The following table provides the corresponding decibel values for the volume ranges of the Amiga system.

Table 5-7: Decibel Values and Volume Ranges

Volume	Decibel Value	Volume	Decibel Value
64	0.0	32	-6.0
63	-0.1	31	-6.3
62	-0.3	30	-6.6
61	-0.4	29	-6.9
60	-0.6	28	-7.2
59	-0.7	27	-7.5
58	-0.9	26	-7.8
57	-1.0	25	-8.2
56	-1.2	24	-8.5
55	-1.3	23	-8.9
54	-1.5	22	-9.3
53	-1.6	21	-9.7
52	-1.8	20	-10.1
51	-2.0	19	-10.5
50	-2.1	18	-11.0
49	-2.3	17	-11.5
48	-2.5	16	-12.0
47	-2.7	15	-12.6
46	-2.9	14	-13.2
45	-3.1	13	-13.8
44	-3.3	12	-14.5
43	-3.5	11	-15.3
42	-3.7	10	-16.1
41	-3.9	9	-17.0
40	-4.1	8	-18.1
39	-4.3	7	-19.2
38	-4.5	6	-20.6
37	-4.8	5	-22.1
36	-5.0	4	-24.1
35	-5.2	3	-26.6
34	-5.5	2	-30.1
33	-5.8	1	-36.1
		0	minus infinity

Chapter 6

BLITTER HARDWARE

6.1. INTRODUCTION

The blitter is a high-performance graphics engine that uses up to four DMA channels. The operations it performs after a setup of its registers are considerably faster than those performed by the 68000. The blitter can be used for data copying and line drawing. The initial sections of this chapter concentrate on the data-copying features of the blitter. The final section of the chapter concentrates on the line-drawing capabilities.

Here is a quick summary of blitter features and the operations they perform:

- DATA COPYING - The blitter can copy bit-plane image data anywhere.
- MULTIPLE SOURCES - Instead of simply retrieving data from a single source, the blitter can retrieve data from up to three sources as it prepares the result for a possible destination area.
- LOGIC OPERATIONS - The blitter can perform up to 256 logic operations on the 3 data sources as they are being copied.
- MULTIPLE MODULOS - The blitter is provided with a separate modulo for each of the sources and for the destination. This allows the blitter to move data to and from identical rectangular windows within different sizes of larger playfield images.
- ASCENDING AND DESCENDING ADDRESSING - The blitter can change addresses in an ascending or descending manner. It can either start at the bottom address of both the source and the destination areas and move the data while incrementing addresses or start at the top address of the source and destination and decrement addresses during the move.
- SHIFTING - The blitter can shift one or two of its data sources up to 15 bits before applying it to the logic operation, allowing movement of images in memory across word boundaries.
- MASKING - The blitter can mask the leftmost and rightmost data word from each horizontal line. Mask registers are provided for the first and the last words on every line of blitter data. This allows logic operations on bit-boundaries from both the left and the right edge of a rectangular region.
- ZERO DETECTION - The blitter can store the result of the logic operations back into memory or simply sense whether there were any 1-bits present as a result of the logic operation. This feature can be used for hardware-assisted software collision detection.
- AREA-FILLING - The blitter can perform a hardware-assisted area fill between predrawn lines.

- LINE-DRAWING - The blitter can draw ordinary lines at any angle and can also apply a pattern to the lines it draws. It can also draw special lines with one pixel dot per horizontal line (a special mode needed during the blitter fill operation).

About this Chapter

This chapter introduces each available feature of the blitter individually. Then each section builds on the information presented in the preceding section to provide examples of the functions of the features. At the end of the chapter there is information about an alternate method of minterm selection, using Venn diagrams.

6.2. DATA COPYING

The primary purpose of the blitter is to copy (transfer) data in large blocks from one memory location to another. This type of machine has been referred to, therefore, as a "blitter" (for block image transferrer). Because this machine has many more features than just transfer, it is also referred to as the "Bimer" (for bitmap image manipulator).

Images in memory are stored in a linear fashion, with each word of data on a line being located at an address that is one greater than the word on its left. This is illustrated below. Note that each line is a "plus one" continuation of the previous line.

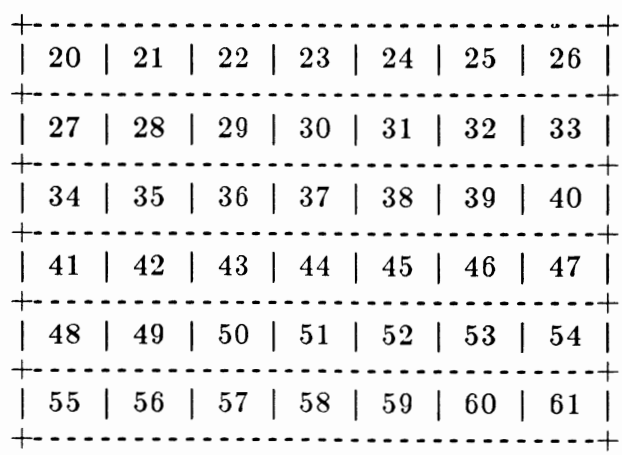


Figure 6-1: How Images are Stored in Memory

This map represents a single bit-plane (one bit of color) of an image at word addresses 20 through 61. Each of these addresses accesses one word (16 pixels) of a single bit-plane. If this image required 16 colors, four bit-planes like this would be required in memory; and four copy (move) operations would be required to completely move the image.

The blitter is very efficient at copying such blocks because it only needs to be told the starting address (20), the destination address, and the size of the block (h = 6, w = 7). It will then automatically move the data, one word at a time, whenever the data bus is available. When the transfer is complete, it will signal the processor with a flag and an interrupt.

Note that this copy (move) operation is operating on memory and may, or may not, be changing the memory presently being used for display.

6.3. MULTIPLE SOURCES

The blitter uses up to four DMA channels. Three DMA channels are dedicated to retrieving data from memory that the blitter uses. These are designated as source A, source B, and source C. The one destination DMA channel is designated as destination D. As is shown in the following sections, it is not always necessary to use all the sources nor is it always appropriate to use the destination DMA channel.

Each channel may be independently enabled by bits 11, 10, 9, and 8 of BLTCON0. These are called USEA, USEB, USEC, and USED. All three sources (if enabled) are fetched from memory in a pipelined fashion and held in registers for logic combination before being sent to the destination.

6.4. LOGIC OPERATIONS

Being able to logically combine data bits from separate image sources during a data move allows the blitter to be very efficient in performing graphics drawing and animation features. For example, you could design a rectangular object to combine on-screen with a pre-existing graphic image (perhaps a car that you want to move in front of some buildings).

Producing this effect requires predrawn images of both the car and the buildings. To animate the car (that is, to move it in front of the buildings), first save the background image where the car will be placed. Then copy the car in its first location. Then restore the old background image and save a new section of the background from the second location. Again, copy the car, this time to the second location. A continuous sequence of save, draw, restore creates the desired effect.

Assume source A is the car image outline (mask), source B is one of the car image's bit planes, and source C is building data or background. The following operation saves the background where the car is going to be placed (destination on the left, sources on the right):

$$T = AC$$

This equation states that the background (C) should be saved (copied) to a temporary destination (T) wherever the car outline mask (A) "and" the background (C) exist together.

Now the car is placed in the background with the following operation:

$$C = AB + \overline{A}C$$

This equation states that the destination is the same as the background source (C), and background (C) should be replaced with car data (B) wherever the car outline mask (A) is true, but (or) should stay background (C) wherever the mask is not true ($\overline{A}$). Now the background must be restored (to prepare for car placement in a different location) using the following operation:

$$C = AT$$

This equation states that the background (C) should be replaced with the saved background (T) wherever the car outline mask exists (A "and" T).

By shifting both the car data (B) and the car outline mask (A) to a new location, and repeating the above 3 steps over and over, the car will appear to move across the background (the buildings).

6.4.1. Blitter Logic Operations - Combining Minterms

The blitter performs various logic operations like the one shown in the last section by combining minterms. A minterm is one of eight possible logical combinations of data bits from three different data sources.

For example, the following equation uses 2 minterms, ABC and AB$\overline{C}$:

$$D = ABC + AB\overline{C}$$

This means that the logic value of D is a 1 if either ABC = 1 or AB$\overline{C}$ = 1.

Another way of reading this equation is that D is true if and only if both A and B are true. This is because the equation could be grouped as:

$$D = AB\ (\ C + \overline{C}\ )$$

However, since the term (C + $\overline{C}$) is always true, this equation reduces to D = AB. Therefore, selecting the two minterms ABC and AB$\overline{C}$ will give the logic operation D = AB. These two minterms are selected with bits 7 and 6 of BLTCON0.

The minterms that can be selected by BLTCON0 control bits are as follows:

MINTERMS:	ABC	$AB\overline{C}$	$A\overline{B}C$	$A\overline{BC}$	$\overline{A}BC$	$\overline{A}B\overline{C}$	$\overline{AB}C$	$\overline{ABC}$
ENABLE BITS (BLTCON0 LF7-LF0):	7	6	5	4	3	2	1	0

Since there are eight minterms, there are 256 possible equations that can be selected.

6.4.2. Table of Commonly Used Equations

For your convenience, the following table contains a set of commonly used equations. The last one in the table ($D = AB + \overline{A}C$) is often referred to as the "cookie-cut" minterm selector.

SELECTED EQUATION	BLTCON0 LF CODE	SELECTED EQUATION	BLTCON0 LF CODE
$D = A$	F0	$D = AB$	C0
$D = \overline{A}$	0F	$D = A\overline{B}$	30
$D = B$	CC	$D = \overline{A}B$	0C
$D = \overline{B}$	33	$D = \overline{AB}$	03
$D = C$	AA	$D = BC$	88
$D = \overline{C}$	55	$D = B\overline{C}$	44
$D = AC$	A0	$D = \overline{B}C$	22
$D = A\overline{C}$	50	$D = \overline{AC}$	11
$D = \overline{A}C$	0A	$D = A + \overline{B}$	F3
$D = \overline{AC}$	05	$D = \overline{A} + \overline{B}$	3F
$D = A + B$	FC	$D = A + \overline{C}$	F5
$D = \overline{A} + B$	CF	$D = \overline{A} + \overline{C}$	5F
$D = A + C$	FA	$D = B + \overline{C}$	DD
$D = \overline{A} + C$	AF	$D = \overline{B} + \overline{C}$	77
$D = B + C$	EE	$D = AB + \overline{A}C$	CA
$D = \overline{B} + C$	BB		

Table 6-1: Table of Common Minterm Values

6.4.3. Equation to Minterm Conversion

An example of how to convert an equation to minterm format in order to derive the select code is given below:

$$D = AB + \overline{A}C \qquad \text{(Starting equation)}$$

$$D = AB\,(\,C + \overline{C}\,) + \overline{A}C\,(\,B + \overline{B}\,) \qquad \text{(Multiplying by 1)}$$

$$D = ABC + AB\overline{C} + \overline{A}BC + \overline{A}\overline{B}C \qquad \text{(Final minterms)}$$

This final form contains only terms that contain *all* of the input sources. These are the minterms you use. These minterms are selected with the minterm enable bits LF7-LF0 as shown below:

ABC	$AB\overline{C}$	$A\overline{B}C$	$A\overline{B}\overline{C}$	$\overline{A}BC$	$\overline{A}B\overline{C}$	$\overline{A}\overline{B}C$	$\overline{A}\overline{B}\overline{C}$	(Available minterms)
1	1	0	0	1	0	1	0	(BLTCON0 LF7-0 code = CA)

6.5. MULTIPLE MODULOS

The blitter uses modulos to allow manipulation of smaller images within larger images. A modulo is the difference between the width of the larger image and the smaller image being manipulated. There are four modulos in the blitter. This allows all three sources and the destination to each have a different size larger bit-plane image.

Figure 6-2 below shows a possible bit-plane image that is larger than the source window being used by the blitter. The numbers represent the addresses (in memory) of the data words containing the image.

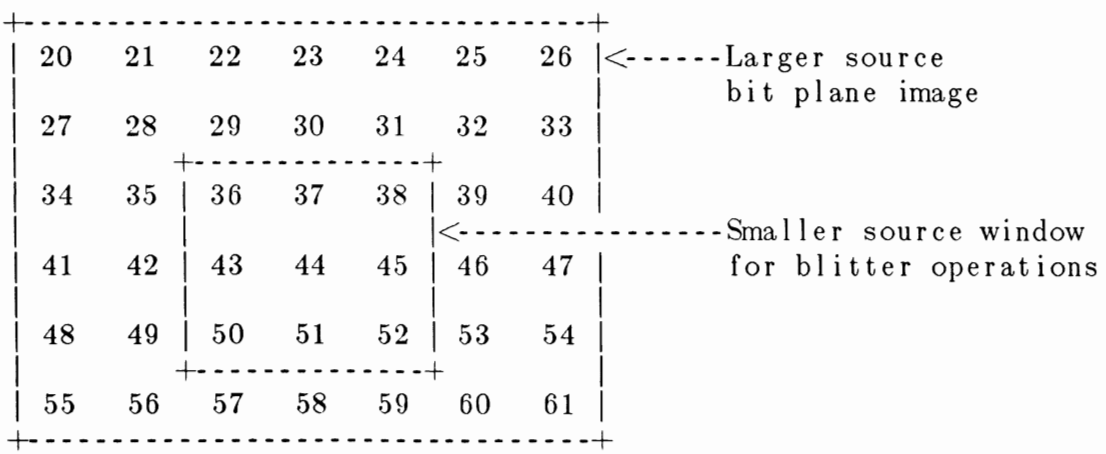


Figure 6-2: Bit-plane Image Larger than the Blitter Source Window

Note that in order to operate on the smaller window only, the address sequence must be as follows:

36, 37, 38,, 43, 44, 45,, 50, 51, 52

This requires a normal increment (+1) each time, and at the end of each window line the addition of a jump value of 4, to bring the address pointer to the start of the next window line. This jump value is called the modulo and is equal to the difference between the width of the large image and the width of the smaller window.

The blitter has a separate modulo register for each of the 3 possible source images, and for the destination image (4 in all). This allows the larger bit map image of each source and the destination to be a different size, even though the smaller window for each is identical.

Note that while the hardware deals in words for pointers and modulos, the values loaded into these hardware registers from the 68000 are treated as byte counts. For example, a jump value of 4 for a modulo would actually be an 8 when written from the 68000.

6.6. ASCENDING AND DESCENDING ADDRESSING

It is important to be able to control the direction of the address increment or decrement when the source and destination areas overlap. By specifying ascending or descending, you can prevent the blitter from destroying source data during overlapping data moves.

If you wish to move data towards a higher address in memory with an overlap between source and destination areas, you should use the descending (address decrement) mode for the data move. If you wish to move data towards a lower address in memory with an overlap between the source and destination areas, you should use the ascending (address increment) mode for the data move. The descending mode is selected with Bit 1 to BLTCON1.

6.7. SHIFTING

The Blitter contains a circuit known as a barrel shifter that can be used with both the A data source, and the B data source. This shifter allows movement of images on pixel boundaries, even though the pixels are addressed 16 at a time by each word address of the bit plane image.

As described previously under "Logic Operations", the movement of a car image (B) across a background (C) requires both the car image (B) and the car outline mask (A) to be shifted to a new position each time the background is saved ($T = AC$), the car placed ($C = AB + A\overline{C}$), and the background restored ($C = AT$).

There are 2 shift controls. Bits 15 through 12 of BLTCON0 select the shift value for source A. Bits 15 through 12 of BLTCON1 select the shift value for source B. Both values are normally set the same.

6.8. MASKING

If an object is not an even multiple of 16 bits in width, the blitter can mask off either the left or the right edge in order to work with only the actual bit-boundary rectangle enclosing the object. First and last word masking is particularly useful when you need to store the images of a text font in a packed edge-to-edge organization.

For example, assume a packed font that contains both an H and an I as shown below:

```
+---------------+
|1111      1111111|
|  11      11   11 |
|  11      11   11 |
|  111111111   11 |
|  111111111   11 |
|  11      11   11 |
|  11      11   11 |
| 111      1111111|
+---------------+
```

To isolate the I-character, the first 11-bits along the left edge of the enclosing rectangle must be masked. The blitter includes this capability, called the first-word mask, and applies it to the leftmost word on each horizontal line. Only when there is a 1-bit in the first-word mask, will that bit of source-A actually appear in the logic operation.

For example, if the first-word mask (BLTAFWM) is 0000000000001111, the data the blitter will see, using the input for source A shown above, is as follows:

```
+---------------+
|              1111|
|               11 |
|               11 |
|               11 |
|               11 |
|               11 |
|               11 |
|               11 |
|              1111|
+---------------+
```

Figure 6-3: Blitter Masking Example

In the same way that the blitter uses the first-word mask to mask the leftmost side of the

source-A data, the blitter also includes a last-word mask (BLTALWM) to mask the rightmost word of the source-A data. So, it is possible to extract rectangular data from a source whose right and left edges are between word boundaries.

If the window is only one word wide (as illustrated above), the first and last word masks will overlap, and source-A bits will be passed only where both masks are true. This example assumed the last word mask was loaded with all ones (FFFF) as should all masks when they are not needed.

6.9. ZERO DETECTION

A Blitter zero flag is provided that can be tested to determine if the logic operation selected has resulted in a null (empty = all zeros) logic operation result. The zero flag (BZERO) in bit 13 of DMACONR will stay true if the result is all zeros.

This feature is usually used to assist collision detection by "anding" two images together to test for overlap. The operation D = AB is performed (D can actually be disabled), and if images A and B do not overlap, the zero flag will stay true.

6.10. AREA FILLING

As well as copying data, the blitter can simultaneously perform a fill operation during the copy. The fill operation has only one restriction. The area to be filled must be defined by first drawing untextured lines that are only one bit wide. A special line draw mode is available for this (see the "Line Drawing" section).

6.10.1. Inclusive (Normal) Area Filling

The figure below shows a typical area fill. It demonstrates one of the bars from a bar chart.

BEFORE	AFTER
001000100	001111100
001000100	001111100
001000100	001111100
001000100	001111100

Figure 6-4: Area-Fill Example - Bar Chart

A blitter line-draw is first performed to provide the two vertical lines, each one bit wide. Then, to fill this area, you follow these steps:

NOTE: A fill operation can be performed during other blitter data copy operations; however, it is often done separately, as shown here.

1. Set the modulos equal to the width of the total image minus the width of the rectangle to be filled.
   (BLTxMOD) (x = A,B,C,D)

2. Set the source and destination pointers to the same value. A case like this requires only one source and destination. This should point to the last (lower-right) word of the enclosing rectangle (see also item 3 below).
   (BLTxPTH, BLTxPTL) (x = A,B,C,D)

3. Run the blitter in the descending direction. The fill operation operates correctly only in the descending mode (right to left).
   (BLTCON1, Bit 1 = 1)

4. Use the control bit called "FCI" (for fill-carry-in) to define how the fill operation should be performed.
   (BLTCON1, Bit 2 = 0) This defines the fill start state as a 0.

5. Define the horizontal and vertical size of a rectangle of words that will enclose the lines around the area to be filled. This value must be written to the size control (BLTSIZE) register to start the fill.

The blitter uses the FCI (fill-carry-in) bit as the starting fill state, starting at the rightmost edge of each line. For each "1" bit in the source area, the blitter "flips" the fill state either filling or not filling the space with 1's. This continues for each line until the left edge of the blit is reached. At that point, the filling stops. For another example, examine the figure below. Only the 1-bits are shown in the figure. The 0-bits are blank and the figure is not drawn to scale.

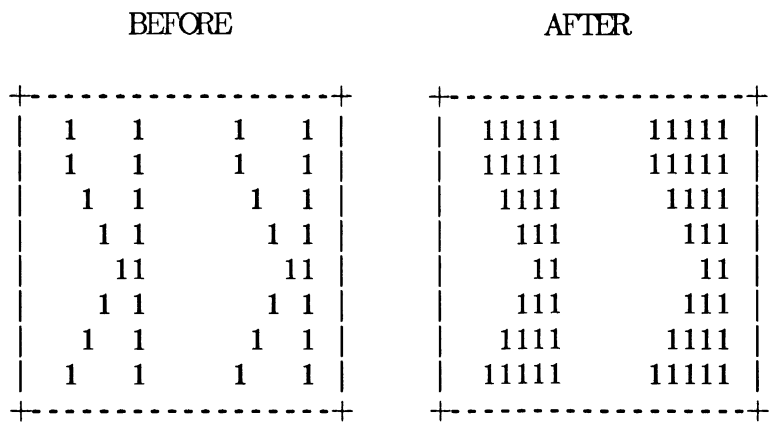


Figure 6-5: Use of the FCI Bit - Before

With the fill-carry-in (FCI) bit as a 1 instead of a zero, the area outside of the lines is filled with 1's and inside the lines is left with 0's in between.

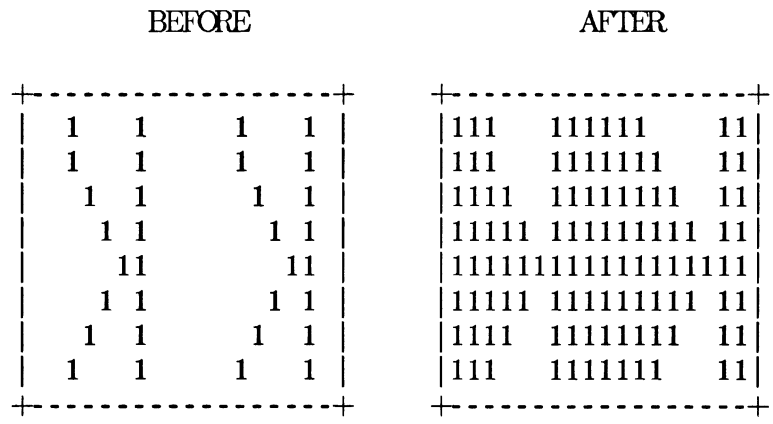


Figure 6-6: Use of the FCI Bit - After

6.10.2. Exclusive Area Filling

There are 2 fill enable bits within BLTCON1. They are called:

IFE - Inclusive Fill Enable (used in the previous examples)

EFE - Exclusive Fill Enable (used in the example below)

Exclusive fill enable means:

Exclude (remove) the outline on the trailing edge (left side) of the fill.

Since the blitter is running in descending mode during a fill, the trailing edge is formed from the leftmost of each pair of bits on a horizontal line.

If you wish to produce very sharp, single point vertices, exclusive fill enable must be used. The illustration below shows how a single point vertex is produced using exclusive fill enable.

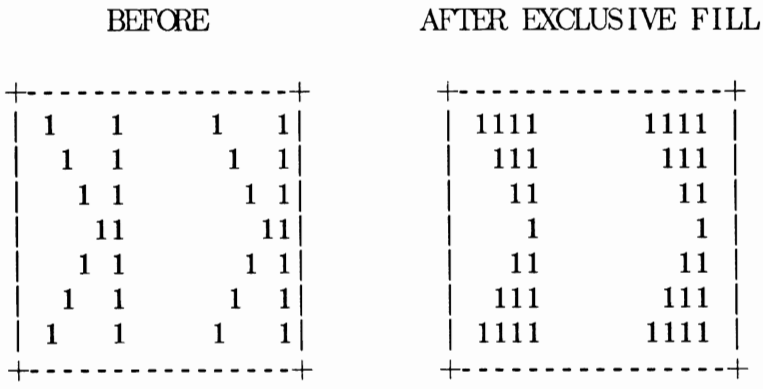


Figure 6-7: Single Point Vertex Example

6.11. LINE DRAWING

In addition to all the functions described above, the Blitter has a line drawing mode. The line draw mode is selected by placing a 1 in Bit 0 of BLTCON1, which causes redefinition of some of the other control bits in BLTCON0 and BLTCON1. (See the description of the BLTCON registers in the appendix to this manual for the meanings of the other control bits.)

When in line draw mode, the blitter has the following features:

- Draws lines up to 1,024 pixels long (twice as big as the high resolution screen).
- Draws lines with regular or inverse video.
- Draws solid lines or textured lines.

- Draws special lines with one dot on each scan line, for use with area fill.

Many of the blitter registers serve other purposes in line draw mode. These registers and their functions are itemized below for reference purposes. You should consult the appendix to this manual for more detailed descriptions of the use of these registers and control bits in the line draw mode.

REGISTER NAME	BIT NUMBER	BIT NAME	STATE	PURPOSE
BLTCON1	0	LINE	1	Enables line draw mode
BLTCON1	15,14,13,12	BSH	0	Starts texture at Bit 0
BLTCON1	1	SING	0,1	Set for single bit width line
BLTCON0	15,14,13,12	START		Code for horizontal position of first pixel
BLTCON0	11,10, 9, 8	USE	1011	Required for line draw
BLTADAT	All		8000	Index required for line draw
BLTBDAT	All		0 to FFFF	Line texture register
BLTSIZE	5-0	w	02	Required for line draw
BLTSIZE	15-6	h		Line length up to 1024
BLTAMOD	All			2(2Y - 2X) *
BLTBMOD	All			2(2Y) *
BLTCMOD	All			Width of total image
BLTDMOD	All			Width of total image
BLTAPTL	All			(2Y - X) *
BLTCPT	All			Starting address of line
BLTDPT	All			Starting address of line
BLTCON1	4,3,2			Octant select code (See the octant figure below.)

* Y and X are the height and width of the rectangle enclosing the line.

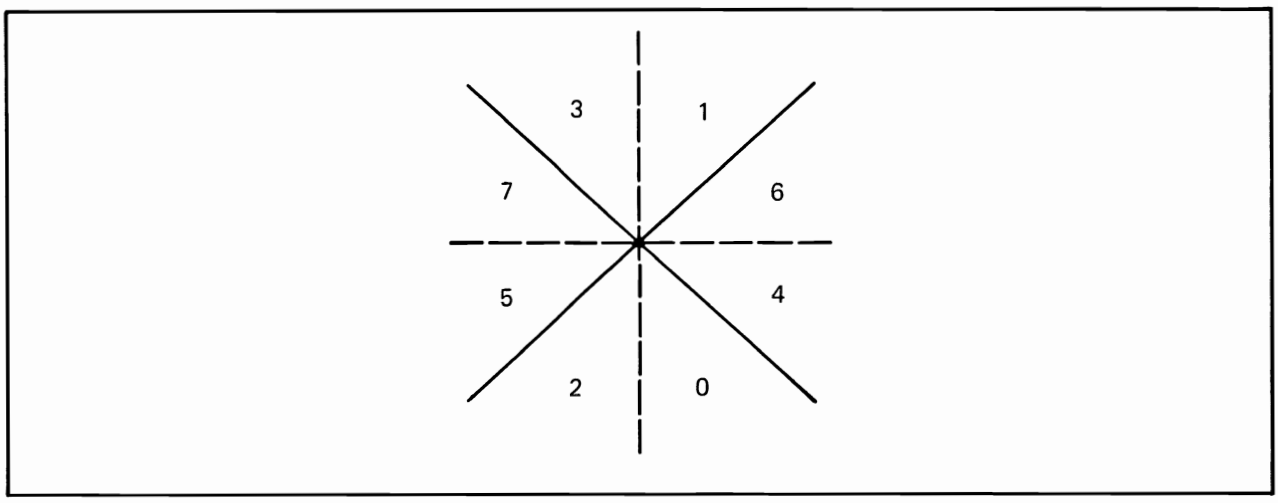


Figure 6-8: Octants for Line Drawing

6.12. VENN DIAGRAMS

The VENN diagram shows a set of three circles labeled A, B and C. In the diagram, the numbers 0 through 7 in various areas correspond to the minterm numbers shown above.

To select which minterms are necessary to produce a certain kind of equation result, you need only to examine the circles and their intersections, and copy down the numbers seen there.

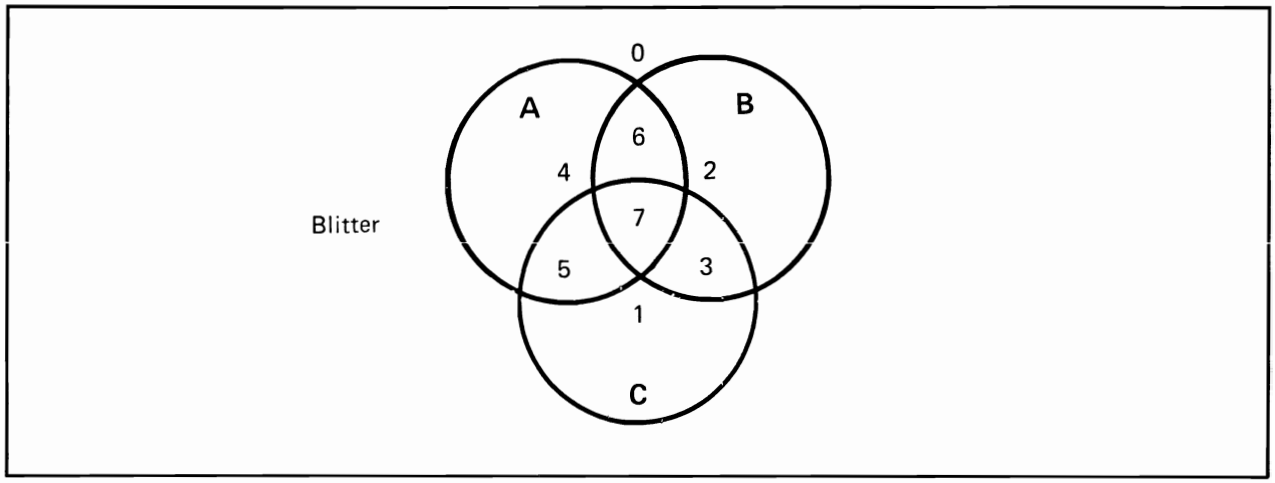


Figure 6-9: Blitter Minterm Venn Diagram

Examples of Venn Diagram Interpretation

1. If you wish to select a function D = A; that is, destination = A source only, you can select only the minterms that are totally enclosed by the A-circle in the figure above. This is the set of minterms 7, 6, 5, and 4. When written as a set of 1's for the selected minterms and 0's for those not selected, the value becomes:

```
7 6 5 4 3 2 1 0 - MINTERM NUMBERS

1 1 1 1 0 0 0 0 - SELECTED MINTERMS
  F       0       equals hex F0.
```

2. If you wish to select a function that is a combination of two sources, you then look for the minterms by <u>both</u> of the circles in their common area. For example, the combination AB (A AND B) is represented by the area common to both the A and B circles. This area encloses both minterms 7 and 6.

```
7 6 5 4 3 2 1 0

1 1 0 0 0 0 0 0   equals hex C0.
```

3. If you wish to use a function that is "not" one of the sources, such as $\overline{A}$, you take all of the minterms <u>not</u> enclosed by the circle represented by A on the figure.
4. If you wish to combine minterms, you only need to "or" them together. For example, the equation AB + BC results in:

```
AB = 1 1 0 0 1 0 0 0
BC = 1 0 0 0 1 0 0 0
```

Result of "or" 1 1 0 0 1 0 0 0 = hex C8.

Chapter 7

SYSTEM CONTROL HARDWARE

7.1. INTRODUCTION

This chapter covers the control hardware of the Amiga system. The following topics are discussed in this chapter:

- How playfield priorities may be specified relative to the sprites
- How collisions between objects are sensed
- How system direct memory access (DMA) is controlled
- How interrupts are controlled and sensed

7.2. VIDEO PRIORITIES

You can control the priorities of various objects on the screen to give the illusion of three dimensions. The section below shows how playfield priority may be changed relative to sprites.

7.2.1. Fixed Sprite Priorities

You cannot change the relative priorities of the sprites. They will always appear on the screen with the lower numbered sprites appearing in front of (having higher screen priority than) the higher numbered sprites. This is shown in the Figure 7-1 below. Each box represents the image of the sprite number shown in that box.

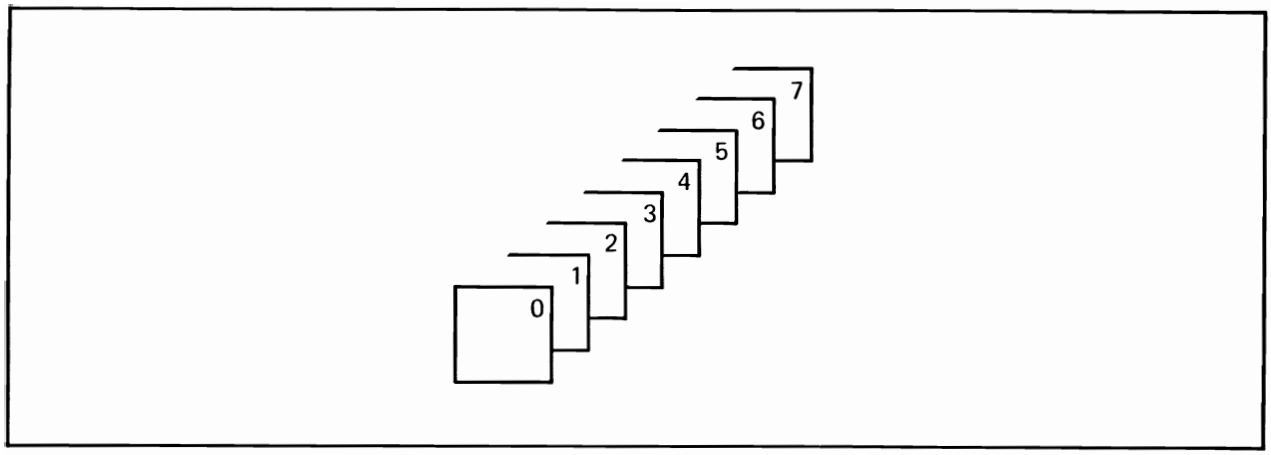


Figure 7-1: Inter-Sprite Fixed Priorities

7.2.2. How Sprites Are Grouped

Sprites are treated as four groups of two sprites each. These groupings are for playfield priority and collision purposes only. The groups of sprites are:

Sprite 0 and 1
Sprite 2 and 3
Sprite 4 and 5
Sprite 6 and 7

7.2.3. Understanding Video Priorities

The concept of video priorities is easy to understand if you imagine that four fingers of one of your hands represent the 4 pairs of sprites. Also imagine that two fingers of your other hand represent the two playfields.

Just as you cannot change the sequence of the four fingers on the one hand, neither can you change the relative priority of the sprites. However, just as you can intertwine the two fingers of one hand in many different ways relative to the four fingers of the other hand, so can you position the playfields in front of or behind the sprites. This is illustrated in Figure 7-2.

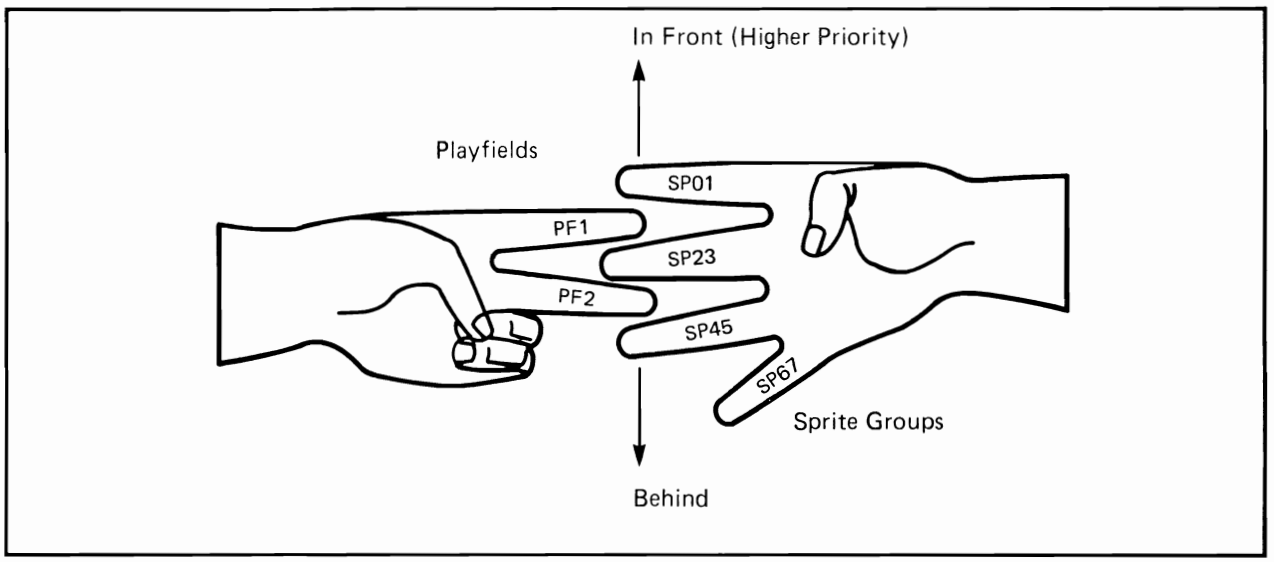


Figure 7-2: Analogy for Video Priority

There are 5 possible positions which you can choose for each of the two "playfield fingers". For example, you can place playfield 1:

on top of sprites 0 and 1,

between sprites 0 and 1 and sprites 2 and 3,

between sprites 2 and 3 and sprites 4 and 5,

between sprites 4 and 5 and sprites 6 and 7, or

beneath sprites 6 and 7

You have the same possibilities for playfield 2.

The numbers 0 through 4 shown for the priority positions above are the actual values you use to select the playfield priority positions. See "Setting the Priority Control Register" below.

You can also control the priority of playfield 2 relative to playfield 1. This gives you additional choices for the way you can design the screen priorities.

7.2.4. Setting the Priority Control Register

This register lets you define how objects will pass in front of each other or hide behind each other. Normally, playfield 1 appears in front of playfield 2. The PF2PRI bit reverses this relationship, making playfield 2 more important. You control the video priorities by using the bits in BPLCON2, Bit-Plane Control Register number 2, as follows.

BIT NUMBER	NAME	FUNCTION
15-7		Not used (keep at 0).
6	PF2PRI	Playfield 2 Priority
5-3	PF2P2 - PF2P0	Playfield 2 Placement with respect to the sprites
2-0	PF1P2 - PF1P0	Playfield 1 Placement with respect to the sprites

For the bits named PF1P2-PF1P0, the binary values which you give them determine where playfield 1 occurs in the priority chain as shown in the table below. This matches the description given in the previous section.

VALUE	PLACEMENT (from most important to least important)				
000	PF1	SP01	SP23	SP45	SP67
001	SP01	PF1	SP23	SP45	SP67
010	SP01	SP23	PF1	SP45	SP67
011	SP01	SP23	SP45	PF1	SP67
100	SP01	SP23	SP45	SP67	PF1

Where:

PF1 stands for playfield 1,

SP01 stands for the group of sprites numbered 0 and 1,

SP23 are sprites 2 and 3 as a group,

SP45 are sprites 4 and 5 as a group, and

SP67 are sprites 6 and 7 as a group

Similarly, bits PF2P2-PF2P0 let you position playfield 2 among the sprite priorities exactly the same way. However, it is the PF2PRI bit which determines which of the two playfields appears in front of the other on the screen. Here is a sample of the BPLCON2 register contents which would create something a little unusual:

BITS	15-7	PF2PRI	PF2P2-0	PF1P2-0
VALUE	0's	1	010	000

This will result in a sprite/playfield priority placement of:

PF1 SP01 SP23 PF2 SP45 SP67

In other words, where objects pass across each other, playfield 1 is in front of Sprite 0 or 1; and sprites 0 through 3 are in front of playfield 2. However, playfield 2 is in front of playfield 1 in any area where they overlap and where playfield 2 is not blocked by sprites 0 through 3.

7.3. COLLISION DETECTION

You can use the hardware to detect collisions between:

one sprite group and another sprite group

OR

any sprite group and either of the playfields

OR

the two playfields

or any combination of these items.

The first kind of collision is typically used in a game operation to determine if a missile has collided with a moving player.

The second kind of collision is typically used to keep a moving object within specified on-screen boundaries.

The third kind of collision detection allows you to define sections of playfield as individual objects, which you may move using the blitter. This is called playfield animation. If one playfield is defined as the backdrop or playing area and the other playfield is used to define objects (in addition to the sprites), you can sense collisions between the playfield-objects and the sprites or between the playfield-objects and the other playfield.

7.3.1. How Collisions Are Determined

The video output is formed when the input data from all of the bit-planes and the sprites is combined into a common data stream for the display. For each of the pixel positions on the screen, the color of the highest priority object is displayed. Collisions are detected when 2 or more objects attempt to overlap in the same pixel position. This will set a bit in the collision data register.

7.3.2. How to Interpret the Collision Data

The collision data register, CLXDAT, is read-only, and its contents are automatically cleared to 0 after it is read. Its bits are as shown in the table below.

BIT NUMBER	COLLISIONS REGISTERED
15	not used
14	Sprite 4 (or 5) to Sprite 6 (or 7)
13	Sprite 2 (or 3) to Sprite 6 (or 7)
12	Sprite 2 (or 3) to Sprite 4 (or 5)
11	Sprite 0 (or 1) to Sprite 6 (or 7)
10	Sprite 0 (or 1) to Sprite 4 (or 5)
9	Sprite 0 (or 1) to Sprite 2 (or 3)
8	Even bit-planes to Sprite 6 (or 7)
7	Even bit-planes to Sprite 4 (or 5)
6	Even bit-planes to Sprite 2 (or 3)
5	Even bit-planes to Sprite 0 (or 1)
4	Odd bit-planes to Sprite 6 (or 7)
3	Odd bit-planes to Sprite 4 (or 5)
2	Odd bit-planes to Sprite 2 (or 3)
1	Odd bit-planes to Sprite 0 (or 1)
0	Even bit-planes to Odd bit-planes

The notes in parentheses in the table above refer to collisions which will register only if you want them to show up. The collision control register, shown below, lets you either ignore or include the odd-numbered sprites in the collision detection.

Notice that in this table, collision detection does not change when you select either single or dual playfield mode. Collision detection depends only on the actual bits present in the odd-numbered or even-numbered bit-planes. The collision control register specifies how to handle the bit-planes during collision detect.

7.3.3. How Collision Detection is Controlled

The collision control register, CLXCON, contains the bits which define certain characteristics of collision detection. Its bits are shown in the table below.

BIT NUMBER	NAME	FUNCTION
15	ENSP7	Enable Sprite 7 (OR with Sprite 6)
14	ENSP5	Enable Sprite 5 (OR with Sprite 4)
13	ENSP3	Enable Sprite 3 (OR with Sprite 2)
12	ENSP1	Enable Sprite 1 (OR with Sprite 0)
11	ENBP6	Enable Bit-Plane 6 (match required for collision)
10	ENBP5	Enable Bit-Plane 5 (match required for collision)
9	ENBP4	Enable Bit-Plane 4 (match required for collision)
8	ENBP3	Enable Bit-Plane 3 (match required for collision)
7	ENBP2	Enable Bit-Plane 2 (match required for collision)
6	ENBP1	Enable Bit-Plane 1 (match required for collision)
5	MVBP6	Match value for Bit-Plane 6 collision
4	MVBP5	Match value for Bit-Plane 5 collision
3	MVBP4	Match value for Bit-Plane 4 collision
2	MVBP3	Match value for Bit-Plane 3 collision
1	MVBP2	Match value for Bit-Plane 2 collision
0	MVBP1	Match value for Bit-Plane 1 collision

Bits 15-12 let you specify that collisions with a sprite pair are to include the odd-numbered sprite of a pair of sprites. The even-numbered sprites always are included in the collision detection.

Bits 11-6 let you specify whether to include or exclude specific bit-planes from the collision detection.

Bits 5-0 let you specify the polarity (true-false condition) of bits which will cause a collision. For example, you may wish to register collisions only when the object collides with "something green" or "something blue". This feature, along with the collision enable bits, allows you to specify the exact bits, and their polarity, for the collision to be registered.

NOTES

This register is write-only.

If all bit-planes are excluded (disabled), then a bit-plane collision will always be detected.

7.4. BEAM POSITION DETECTION

Sometimes you might want to synchronize the 68000 processor to the video beam which is creating the screen display. In some cases, you may wish to update a part of the display memory after the system has already accessed the data from the memory for the display area.

This address is provided so that you can determine the value of the video beam counter and perform certain operations based on the beam position. Note, however, that the system coprocessor is already capable of watching display position for you and doing certain register-based operations automatically. Refer to "Copper Interrupts" below and Chapter 2, "Coprocessor Hardware", for further information.

In addition, when you are using a light-pen with this system, this same address is used to read the light-pen position rather than the beam position. This is fully described in Chapter 8, "Interface Hardware".

7.4.1. Using the Beam Position Counter

There are four addresses that access the beam position counter. Their usage is described below.

VPOSR *Read-only.* Read the high bit of the vertical position (V8), and the frame-type bit.

Bit 15 LOF - Long Frame Bit
Used to initialize interlace displays.

Bits 14-1 - Unused.

Bit 0 - Contains the high bit of the vertical position (V8).
This ninth bit allows PAL line counts (313) to appear In PAL versions of this computer.

VHPOSR *Read-only.* Read vertical and horizontal position of the counter which is producing the beam on the screen. (Also reads the light pen.)

Bits 15-8 - Contain the low bits of the vertical position, bits V7-V0.

Bits 7-0 - Contain the horizontal position, bits H8-H1.
(Horizontal resolution is 1/160th of the screen width.)

VPOSW *Write only.* Bits same as VPOSR above.

VHPOSW *Write only.* Bits same as VHPOSR above. Used for counter synchronization with chip test patterns.

As usual, the address pairs VPOSR,VHPOSR and VPOSW,VHPOSW can be read from and written to as long words with the most significant addresses being VPOSR and VPOSW.

7.5. INTERRUPTS

This system supports the full range of 68000 processor interrupts. The various kinds of interrupts generated by the hardware are brought into the peripherals chip and are translated into 6 of the 7 available interrupts of the 68000.

7.5.1. Non-Maskable Interrupt

Interrupt level 7 is the non-maskable interrupt and is not generated anywhere in the current system. The raw interrupt lines of the 68000, IPL2 through IPL0, are brought out to the expansion connector and can be used to generate this level 7 interrupt for debug purposes.

7.5.2. Maskable Interrupts

Interrupt levels 1 through 6 are generated. Control registers within the peripherals chip allow you to mask certain of these sources and prevent them from generating a 68000 interrupt.

7.5.3. User Interface to the Interrupt System

The system software has been designed to correctly handle all system hardware interrupts at levels 1 through 6. A separate set of input lines, designated INT2* and INT6* have been routed to the expansion connector for use by external hardware for interrupts. These are known as the external low and external high level interrupts.

These interrupt lines are connected to the peripherals chip and create interrupt levels 2 and 6, respectively. It is recommended that you take advantage of the interrupt handlers built into the operating system by using these external interrupt lines rather than generating interrupts directly on the processor interrupt lines.

7.5.4. Interrupt Control Registers

There are two interrupt registers, interrupt enable (mask) and interrupt request (status). Each register has both a read and a write address.

The following are the names of the interrupt addresses:

INTENA
: Interrupt enable (mask) - *write only.* Sets or clears specific bits of INTENA.

INTENAR
: Interrupt enable (mask) read - *read only.* Read contents of INTENA.

INTREQ
: Interrupt request (status) - *write only.* Used by the processor to force a certain kind of interrupt to be processed (software interrupt). Also used to clear interrupt request flags once the interrupt process is completed.

INTREQR
: Interrupt request (status) read - *read only.* Contains the bits that define which items are requesting interrupt service.

  The bit positions in the interrupt request register correspond directly to those same positions in the interrupt enable register. The only difference between the read-only and the write-only addresses shown above is that Bit 15 has no meaning in the read-only addresses.

7.5.5. Setting and Clearing Bits

Here are the meanings of the bits in the interrupt control registers and how you use them.

Set and Clear

The interrupt registers, as well as the DMA control register, use a special way of selecting which of the bits are to be set or cleared. Bit 15 of these registers is called the SETCLR bit.

When you wish to set a bit (make it a 1), you must place a 1 in the position you want to set, *and a 1 into position 15.*

When you wish to clear a bit (make it a 0), you must place a 1 in the position you wish to clear, *and a 0 into position 15.*

Positions 14-0 are bit-selectors. You write a 1 to any one or more bits to select that bit. At the same time you write a 1 or 0 to bit 15 to either set or clear the bits which you have selected. Positions 14-0 that have 0 value will not be affected when you do the write. If you want to set some bits and clear others, you will have to write this register twice (once for setting some bits, once for clearing others).

Master Interrupt Enable

Bit 14, INTEN, of the interrupt registers is for interrupt enable.

This is the master interrupt enable bit. If this bit is a 0, it disables all other interrupts.

You may wish to clear this bit to temporarily disable all interrupts to do some critical processing task.

NOTE: This bit is used for enable/disable only. It creates no interrupt request.

External Interrupts

Bits 13 and 3 of the interrupt registers are reserved for external interrupts.

Bit 13, EXTER, becomes a 1 when the system line called INT6* becomes a logic 0. Bit 13 generates a level 6 interrupt.

Bit 3, PORTS, becomes a 1 when the system line called INT2* becomes a logic 0.

Bit 3 causes a level 2 interrupt.

Vertical Blanking Interrupt

Bit 5, VERTB, causes an interrupt at line 0 (start of vertical blank) of the video display frame.

The system is often required to perform many different tasks during the vertical blanking interval. Among these tasks are the updating of various pointer registers, rewriting lists of Copper tasks when necessary, and other system-control operations.

The minimum time of vertical blanking is 20 horizontal scan lines (begins at line 0 and ends at line 20). You additionally have control over where (after line 20) the display actually starts by using the DIWSTRT (display window start) register (see Chapter 3, "Playfield Hardware"). This can extend the effective vertical blanking time.

If you find that you still require additional time during vertical blanking, you can use the Copper to create an interrupt. This Copper interrupt would be timed to occur just after the last line of display on the screen (after the display window stop which you have defined by using the DIWSTOP register).

This causes a level 3 interrupt.

Copper Interrupt

Bit 4, COPER, is used by the Copper to issue an interrupt. The Copper can change the content of any of the bits of this register, as it can write any value into most of the machine registers. However, this bit has been reserved for specifically identifying the Copper as the interrupt source.

Generally, you use this bit when you want to sense that the display beam has reached a specific position on the screen, and you wish to change something in memory based on this occurrence.

This generates a level 3 interrupt.

Audio Interrupts

Bits 10 - 7, AUD3 - 0, are assigned to the audio channels. They are called AUD3, AUD2, AUD1, and AUD0, and are assigned to channels 3, 2, 1, and 0, respectively.

This interrupt signals "audio block done". When the audio DMA is operating in automatic mode, this interrupt occurs when the last word in an audio data stream has been accessed. In manual mode, it occurs when the audio data register is ready to accept another word of data.

See Chapter 5, "Audio Hardware", for more information about interrupt generation and timing.

This generates a level 4 interrupt.

Blitter Interrupt

Bit 6, BLIT, signals "blitter finished". If this bit is a 1, it indicates that the blitter has completed the requested data transfer. The blitter is now ready to accept another task.

This generates a level 3 interrupt.

Disk Interrupt

Bits 12 and 1 of the interrupt registers are assigned to disk interrupts.

Bit 12, DSKSYN, indicates that the sync register matches disk data. This generates a level 5 interrupt.

Bit 1, DSKBLK, indicates "disk block finished". It is used to indicate that the specified disk DMA task which you have requested has been completed. This generates a level 1 interrupt.

More information about disk data transfer and interrupts may be found in Chapter 8, "Interface Hardware".

Serial Port Interrupts

The following serial interrupts are associated with the specified bits of the interrupt registers.

Bit 11, RBF (for receive buffer full), specifies that the input buffer of the UART has data that is ready to read.

This generates a level 5 interrupt.

Bit 0, TBE (for transmit buffer empty), specifies that the output buffer of the UART needs more data and can now be written into.

This generates a level 1 interrupt.

7.6. DMA CONTROL

Many different direct memory access (DMA) functions occur during system operation. There is a read address as well as a write address to this register so you can tell which DMA channels are enabled.

Here are the address names for this register:

DMACONR - Direct Memory Access Control - *read-only.*
DMACON - Direct Memory Access Control - *write-only.*

The contents of this register are as follows (bit on if enabled):

BIT NUMBER	NAME	FUNCTION
15	SET/CLR	The set/reset control bit. See the description of Bit 15 under "Interrupts" above.
14	BBUSY	Blitter Busy Status - *read-only*
13	BZERO	Blitter zero status - *read-only.* Remains a 1 if, during a blitter operation, the blitter output was always zero.
12, 11		Unassigned.
10	BLTPRI	Blitter Priority. Also known as "blitter-nasty". When this is a 1, the blitter has full (instead of partial) priority over the 68000.
9	DMAEN	DMA Enable. This is a master DMA enable bit. It enables the DMA for all of the channels at bits 8 through 0.
8	BPLEN	Bit-Plane DMA enable
7	COPEN	Coprocessor DMA enable
6	BLTEN	Blitter DMA enable
5	SPREN	Sprite DMA enable
4	DSKEN	Disk DMA enable
3-0	AUDxEN	where x = 3 through 0. Audio DMA enable, channels 3-0.

For more information on using the DMA, see the following chapters:

sprites - Chapter 4, "Sprite Hardware"
bit-planes - Chapter 3, "Playfield Hardware"
blitter - Chapter 6, "Blitter Hardware"
disk - Chapter 8, "Interface Hardware"
audio - Chapter 5, "Audio Hardware"
Copper - Chapter 2, "Coprocessor Hardware"

Chapter 8

INTERFACE HARDWARE

8.1. INTRODUCTION

This chapter covers the ways in which the Amiga talks to the outside world. This includes the following features:

- o mouse/joystick/light pen ports
- o disk controller
- o keyboard
- o parallel I/O interface
- o RS-232 compatible serial interface (for external modems or serial devices)
- o RAM cartridge slot (for expansion to 512k)
- o Expansion bus interface
- o Audio output jacks
- o Video output connectors (RGB, NTSC, RF modulator)

8.2. CONTROLLER PORT INTERFACE

At the side of the computer, there are two 9-pin connectors which can be used for many different kinds of controllers. Figure 8-1 below shows one of the two computer connectors, and the corresponding face-on view of the typical controller plug.

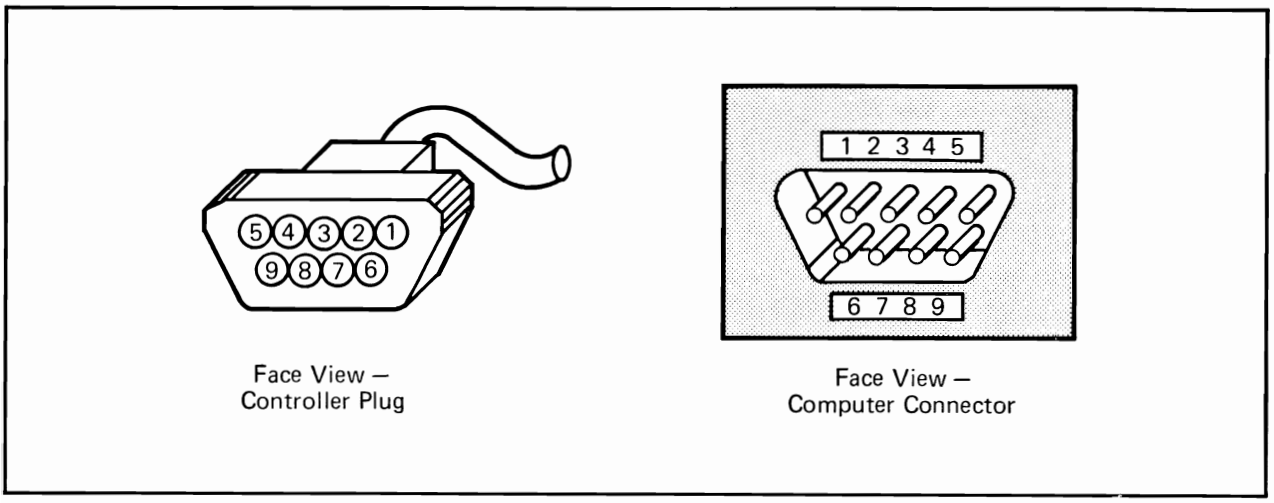


Figure 8-1: Controller Plug and Computer Connector

8.2.1. How to Read The Controller Port

This section shows how to read mouse controllers, joysticks, proportional controllers, and light pens. All of these controllers use the same connector, but sometimes have considerably different functions. Therefore, the pins are used differently depending on the the type of controller used.

Mouse/Trackball Controllers

The inputs for the mouse or trackball are shared with the input pins for the joystick switches as follows:

- Joystick "right" switch and "back" switch are shared with the mouse or trackball horizontal motion detection.
- Joystick "left" switch and "forward" switch are shared with the mouse or trackball vertical motion detection.

Pulses enter these inputs from the mouse or trackball, and are converted into an up-count or a down-count when motion occurs. In this section, only the mouse action is described. The trackball activity is identical.

Direction of Motion vs Count

Imagine that the mouse is being moved on the table over an exact image of the screen itself. The movement of the mouse to control an object corresponds exactly to the movements the user makes with the mouse itself (all directions of movement are exactly the same).

The counter counts up when the mouse is moved to the right or "down". The counter counts down when the mouse is moved to the left or "up". Up is away from you. Down is towards you.

This corresponds to the counting directions on the screen, where the coordinates X=0, Y=0 are at the upper left corner of the screen and X=Xmax, Y=Ymax are at the lower right hand corner.

How to Read the Counters

The mouse/trackball counter contents can be accessed by reading register addresses named JOY0DAT and JOY1DAT. These contain the counts for the left (0) and the right (1) controller ports.

The contents of each of these 16-bit registers is as follows:

Bits 15-8	Mouse/Trackball Vertical Count
Bits 7-0	Mouse/Trackball Horizontal Count

Counter Limitations

These counters will "wrap around" in either the positive or negative direction. If you wish to use the mouse to control something that is happening on the screen, you have to:

a. Read the counters once each vertical blanking period
b. Save the previous contents of the registers so that you can subtract to determine

   i. direction of movement
   ii. speed

The reason for reading counter contents once each vertical blanking time is to see if the user has moved the mouse in between the times you read the counters.

The mouse produces about 200 count pulses per inch of movement in either a horizontal or vertical direction. Vertical blanking happens once each 1/60th of a second. If you read the mouse once each vertical blanking period, you will most likely find a count difference (from the previous count) of less than 127. (Only if a user can move the mouse at a speed of more than 72 inches per second will it exceed this count ... an unlikely happening).

If you subtract the current count from the previous count, the absolute value of the difference will represent the speed. And the sign of the difference (if + or -), along with the sign of the previous and current values, lets you determine which direction the mouse is traveling.

The example shown here treats both counts as unsigned values, ranging from 0 to 255. A count of 100 pulses is measured in each case.

PREVIOUS COUNT	CURRENT COUNT	DIRECTION
200	100	UP (LEFT)
100	200	DOWN (RIGHT)
200	45	DOWN *
45	200	UP **

* 200-45 = 155. Because this is more than 127, the true count must be

255 - (200-45) = 100.

Therefore the count is 100, and the direction is DOWN.

** 45-200 = -155. Because the absolute value exceeds 100, the true count must be

255 + (-155) = 100, and the direction is UP.

There are two buttons on the Amiga Mouse. However, the control circuitry supports mice and trackballs with as many as three buttons if desired.

BUTTON 1 (Left button on Amiga mouse)

Button 1 is connected to pin 6 of the controller port.

Trigger-lines are read, for each of the controller ports, by reading PA7 (port 1 fire-button) or PA6 (port 0 fire-button) of the odd-addressed 8520 peripheral ports. A logic state of 1 means switch-open. A logic state of 0 means switch-closed.

BUTTON 2 (Right button on Amiga mouse)

Button 2 is connected to pin 9 of the controller ports. It is read as one of the potentiometer ports. See "Reading Proportional Controllers" for more information. High resistance indicates switch-open. Low resistance to ground indicates switch-closed.

BUTTON 3

Button 3, when used, would be connected as the other proportional controller input. This is pin 5 of the controller ports.

Joystick Controllers

The joystick controllers have 4 simple direction switches and one trigger button. The direction switches are connected to pins 1, 2, 3, and 4 as FORWARD, BACK, LEFT, and RIGHT. The trigger button is connected to pin 6.

The normal state of each of the switches is open. This places a logic 1 on each of the input lines. When a switch is closed, it is connected to ground (pin 8), placing a logic 0 on the line.

Reading the joystick input data logic states is not so simple, however, because the data register for the joysticks is the same as the counters used for the mouse or trackball controllers. These are named JOY0DAT (port 0) and JOY1DAT (port 1).

The following table shows how to interpret the data once you have read it from these registers. The true logic state of the switch data in these registers is "1 = switch closed".

DATA BIT	INTERPRETATION
1	True logic state of "right" switch
9	True logic state of "left" switch
1 (XOR) 0	You must calculate the exclusive-or of bits 1 and 0 to obtain the logic state of the "back" switch
9 (XOR) 8	You must calculate the exclusive-or of bits 9 and 8 to obtain the logic state of the "forward" switch

Proportional Controllers

Each of the game controller ports can handle two variable-resistance input devices, also known as proportional input devices. This section describes how the positions of the proportional input devices can be determined.

Different Types of Proportional Controllers

There are two common types of proportional controllers: the "paddle" controller pair and the X-Y proportional joystick.

A paddle controller pair consists of two individual enclosures, each containing a single resistor and fire-button and each connected to a common controller port input connector. The typical connection is as shown in Figure 8-2.

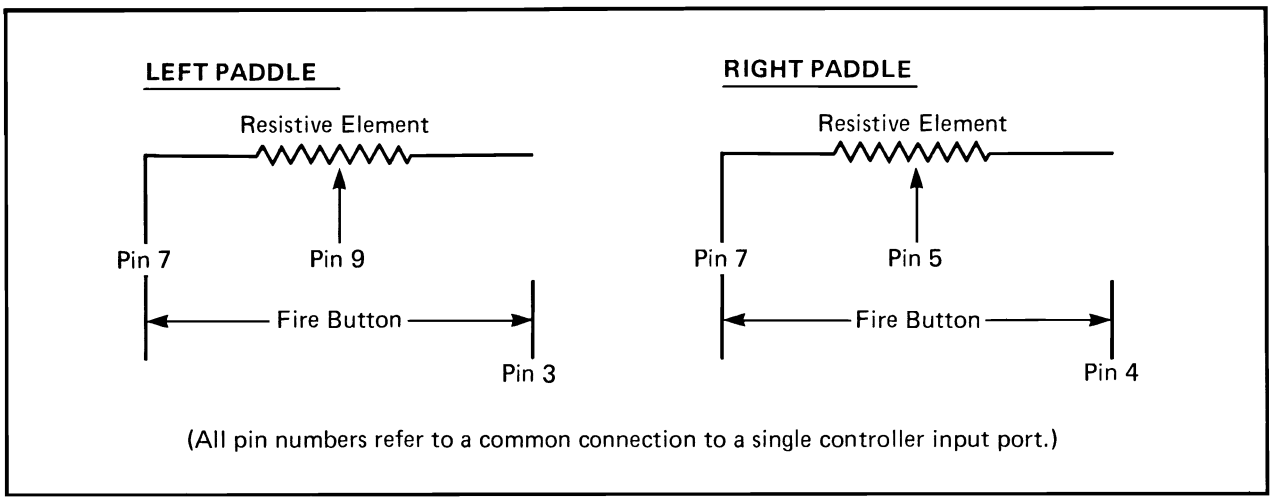


Figure 8-2: Typical Paddle Controller Connection

In an X-Y proportional joystick, the resistive elements are connected individually to the X and Y axes of a single controller enclosure, (instead of being in in separate enclosures). Typical connections are shown in the table below.

Pin	Joystick	Mouse, Trackball, Driving Controller	Proportional Controller (Pair)	X-Y Variable Proportional Joystick
1	Forward*	V-pulse	(unused)	(unused)
2	Back*	H-pulse	(unused)	(unused)
3	Left*	VQ-pulse	Left Button	Button 1
4	Right*	HQ-pulse	Right Button	Button 2
5	(unused)	Button (3) (if used)	Right POT	POT X
6	Button(1)	Button(1)	(unused)	(unused)
7	+5V	+5V	+5V	+5V
8	GND	GND	GND	GND
9	Button(2) (if used)	Button(2)	Left POT	POT Y

Reading Proportional Controller Buttons

For the two-control paddle controllers, the left and right joystick inputs serve as the fire buttons for the left and right controllers.

Interpreting Proportional Controller Position

To interpret the position of the controller requires some preliminary work by you. This is an activity normally done during the vertical blanking interval (and is part of the operating system function).

During vertical blanking, you write a value into an address called POTGO. For a standard X-Y joystick, this value is hex 0001. Writing to this register starts the operation of some special hardware which reads the potentiometer values. It also sets the values contained in the pot registers (described below) to zero.

The read-circuitry stays in a reset state for the first 7 or 8 horizontal video scan lines. Following the reset interval, the circuit allows a charge to begin building up on a timing capacitor whose charge-rate will be controlled by the position of the external controller resistance. Each horizontal scan line thereafter, the circuit compares the charge on the timing capacitor

to a preset value. If the charge is below that value, one count is added to the counter for that "pot". If it is more, that counter value will be stopped. The counter will be held at the stopped value until the next POTGO is issued.

Effects of Different Resistance on Charging Rate

You normally issue POTGO at the beginning of a video screen, then read the values in the POT registers during the next vertical blanking period, just before issuing POTGO again. (Again note that this is an automatic feature of the operating system).

There is nothing in the system to prevent the counters from overflowing (wrapping past a count of 255). However, the system is designed to insure that the counter cannot overflow within the span of a single screen. This allows you to know for certain whether an overflow is indicated by the controller.

There are 262 or 263 possible horizontal scan lines on a single NTSC video screen. But each of the POT counters is 8 bits wide. This allows a maximum of 255 in any counter. This is why the control circuitry is delayed 7 or 8 horizontal scan lines—to limit the maximum POT count value to 255.

Proportional Controller Registers

The following registers are used for the proportional controllers:

POT0DAT - Port 0 Data (vertical/horizontal)
POT1DAT - Port 1 Data (vertical/horizontal)

Bit Positions:

15-8 POT0Y value or POT1Y value
7-0 POT0X value or POT1X value

All counts are reset to zero when POTGO is written. Counts are normally read one screen after the scan circuitry is enabled.

Potentiometer Specifications

The resistance of the potentiometers should be a linear taper. Based on the design of the integrating analog to digital converter used, the maximum resistance should be no more than 528K (470K +/- 10% is suggested) for either the X or Y pots. This is based on a charge capacitor of 0.047uf, +/- 10%, and a maximum time of 16.6 milliseconds for charge to full value (one video frame time).

Light Pen

You only connect a light pen to the left controller port (port 0). Its connections are not shown on the drawing.

A light pen normally uses the following pins of the controller port:

PIN NUMBER	USAGE
7	+5V
8	GND
5	Pen-Pressed-To-Screen
6	Capture Beam Position

The signal called pen-pressed is generally a single switch to ground, normally open, which is actuated by a switch in the nose of the light pen. Note that this is connected to one of the potentiometer inputs and must be treated as such.

The signal called "Capture Beam Position" is connected as the trigger switch of a normal joystick.

The principles of light pen operation are as follows (assuming the light pen has been enabled):

1. Just as the system exits vertical blank, the capture circuitry for the light pen is automatically enabled.
2. The video beam starts to create the picture, sweeping from left to right for each horizontal line as it paints the picture from the top of the screen to the bottom.
3. The light pen creates a trigger signal at the moment that the video beam passes the window in the nose of the pen.
4. This trigger signal tells the internal circuitry to capture and save the current contents of the BEAM register, VPOSR.

This allows you to determine where the pen was placed by reading the exact horizontal and vertical value of the counter beam at the instant the beam passed the light pen.

How to Read the Light Pen Registers

The light pen register is at the same address as the beam counter, VPOSR and VHPOSR. The bits are:

VPOSR:	Bit 15	Long frame
	Bits 14-1	Unused
	Bit 0	V8 (most significant bit of vertical position)
VHPOSR:	Bits 15-8	V7-V0 (vertical position)
	Bits 7-0	H8-H1 (horizontal position)

The software can refer to this register set as a long word whose address is VPOSR.
The positional resolution of these registers is:

Vertical	1 scan line in non-interlace mode 2 scan lines in interlace mode
Horizontal	2 low-resolution pixels in either high- or low-resolution

To enable the light pen input, write a 1 into Bit 3 of bit-plane control register 0 (BPLCON0). Once the light pen input is enabled and the light pen issues a trigger signal, the value in VPOSR is frozen (The counters still count; just the read value is frozen.) This freeze is released at the end of internal vertical blanking (vertical position 20). There is no single bit in the system that can tell you that the light pen has been triggered, but fear not, here is the way:

1. Read (long) VPOSR twice.
2. If both values are not the same, the light pen has not triggered since the last top-of-screen (V = 20).
3. If both values are the same, then mask off the upper 15 bits of the 32-bit word and compare it with the hex value of $10500 (V=261).
4. If the VPOSR value is greater than $10500, then the light pen did not trigger since the last top-of-screen. If the value is less, then the light pen did trigger and the value read is the screen position of the light pen.

There is a somewhat simplified method of determining the truth of the light pen value if you instruct the system software to read the register only during the internal vertical blanking period of $0 < V 20$.

1. Read (long) VPOSR once, during the period of $0 < V 20$.
2. Mask off the upper 15 bits of the 32-bit word and compare it with the hex value of $10500 ($V = 261$).
3. If the VPOSR value is greater than $10500, then the light pen did not trigger since the last top-of-screen. If the value is less, then the light pen did trigger and the value read is the screen position of the light pen.

Adapting to Special Controllers

The Amiga can read and interpret controllers other than the standard joystick or proportional controller by using the control lines built into the POTGO register (address DFF034) to redefine the functions of some of the controller port pins.

Here is a copy of part of the POTINP/POTGO register bit description paraphrased from Appendix A of this manual:

> POTGO (DFF034) Write-only address for the pot control register.
> POTINP (DFF016) Read-only address for the pot control register.
>
> This register controls a 4 bit bi-directional I/O port that shares the same 4 pins as the 4 pot inputs.

BIT NUMBER	NAME	FUNCTION
15	OUTRY	Output enable for Right Port Pin 9
14	DATRY	I/O data Right Port, Pin 9
13	OUTRX	Output enable for Right Port Pin 5
12	DATRX	I/O data Right Port Pin 5
11	OUTLY	Output enable for Left Port Pin 9
10	DATLY	I/O data Left Port, Pin 9
09	OUTLX	Output enable for Left Port Pin 5
08	DATLX	I/O data Left Port, Pin 5
07-01	X	Reserved for chip identification
00	START	Start pots (dump capacitors, start counters)

Instead of using the pot pins as variable-resistive inputs, you can use these pins as a 4-bit input/output port. This provides you with the equivalent of two additional pins on each of

the two controller ports for general purpose I/O as shown in the table above.

If you set any of the "OUT.." bits high, it disconnects the potentiometer control circuitry from the port. Whatever is the current state, 1 or 0 of the "DAT.." pins in this register will appear on the specified port pin. You set the state of the OUT.. and DAT.. pins by writing into this register through the POTGO address. There are large capacitors on these lines, and it can take up to 300 microseconds for the line to change state.

To use this register as an input, sensing the current state of the pot pins, write all zeros to POTGO. Thereafter you can read the current state by using read-only address POTINP. Any bits set as inputs will be affected by the START bit of POTGO register. You can also use these signals for fire-buttons. To do this, drive the line high (set both OUT.. and DAT.. to 1). When the button is pressed, the line will be shorted to ground and reading POTINP will get you a 0. If the button isn't pressed, the reading will be 1.

8.3. DISK CONTROLLER

The disk controller in this system can handle four double-sided, 3 1/2 or 5 1/4 inch disk drives. A 3 1/2 inch drive is installed in the basic unit. The other drives are external to the main box.

Control of the disk operations is distributed among several registers in the system. Among the control types are:

- Selection, motor control, sensing
- Disk DMA channel control, DMA enable
- Disk data read/write
- Disk format control
- Interrupts generated

8.3.1. Disk Selection, Control and Sensing

The disk subsystem uses two 8520 ports plus one FLAGS interrupt port. The specific bits used are detailed below.

CIA A ($BFE001), port A, has four input bits allocated to the disk subsystem. CIA B ($BFD000), port B, is entirely dedicated to output bits for the disk.

PORT	PIN	NAME	FUNCTION
CIA A	PA5	DSKRDY*	Disk ready (active low)
CIA A	PA4	DSKTRACK0*	Disk heads currently positioned over track zero (active low).

CIA A	PA3	DSKPROT*	Disk is write protected (active low).
CIA A	PA2	DSKCHANGE*	Disk has been removed from the drive. The drives that support this signal latch it until the next time the heads are stepped (active low).
CIA B	PB7	DSKMOTOR*	Disk motor control (active low). This signal is non-standard on the Amiga system. Each drive will latch the motor signal at the time its SELn* signal turns on. The disk drive motor will stay in this state until the next time SELn* turns on, at which time it will latch the new value of DSKMOTOR*. All software that selects drives should set up the motor signal before selecting any drives. The drive will "remember" the state of its motor when it is not selected. All drive motors turn off after system reset.
CIA B	PB6	DSKSEL3*	Select drive 3 (active low).
CIA B	PB5	DSKSEL2*	Select drive 2 (active low).
CIA B	PB4	DSKSEL1*	Select drive 1 (active low).
CIA B	PB3	DSKSEL0*	Select drive 0 (internal drive) (active low).
CIA B	PB2	DSKSIDE*	Specify a particular head of the disk. Zero implies the upper head.
CIA B	PB1	DSKDIREC	Specify the direction to seek the heads. Zero implies seek towards the center spindle. Track zero is at the outside of the disk.
CIA B	PB0	DSKSTEP*	Step the heads of the disk. This signal should always be used as a pulse (first low, then high). Leaving this line low while changing the SEL lines confuses the change logic.
CIA B	FLAG	DSKINDEX*	Disk index pulse (BFDD00, Bit 4). Can be used to create level 6 interrupt.

Disk DMA Channel Control

Data is normally transferred to the disk by direct memory access. The disk DMA is controlled by four items:

- Pointer to the area into which or from which the data is to be moved
- Length of data to be moved by DMA
- Direction of data transfer (read/write)
- DMA enable

Pointer to Data

You specify the 19-bit-wide byte-address from which or to which the data is to be transferred. The lowest bit (bit 0) of this address is treated as a zero. (You cannot start data on an odd byte-boundary).

This address must be written into registers named DSKPTH and DSK PTL. DSKPTH gets the high 3 bits of the pointer, DSKPTL gets the low 16 bits of the pointer.

These registers are positioned at two consecutive word addresses on a long word boundary within the register space. This allows you to initialize both by a single write of a long word to the address of DSKPTH.

Length, Direction, DMA Enable

All of the control bits relating to this topic are contained in a single register, called DSKLEN. Its bits are as follows:

BIT NUMBER	NAME	USAGE
15	DMAEN	Disk DMA Enable
14	WRITE	Disk Write (Ram → disk if 1)
13-0	LENGTH	Number of WORDS to transfer

The bit called DMAEN must be set in order to allow disk DMA to occur. The system DMA control bit for the disk must be set as well. See Chapter 7, "System Control Hardware", for more information about system DMA controls.

The hardware requires a special sequence in order to start DMA to the disk. This sequence prevents accidental writes to the disk. In short, the DMAEN bit in the DSKLEN register must be turned on twice in order to actually enable the disk DMA hardware. Here is the sequence you should follow:

1. Set this register to 4000 hex, thereby forcing the DMA for the disk to be turned off.
2. Put the value you want into the DSKLEN register.
3. Write this value again into the DSKLEN register. This actually starts the DMA.
4. After the DMA is complete, set the DSKLEN register back to 4000 hex, to prevent accidental writes to the disk.

As each data word is transferred, the LENGTH value is decremented. As each transfer occurs, the pointer DSKPTH, DSKPTL value is incremented. This points to the area where the next word of data will be written or read. When the LENGTH value counts down to zero, the transfer stops.

The recommended method of reading from the disk is to read an entire track into a buffer, and then search for the sector(s) that you want. This means that you only have to read from the disk once for the entire track. In addition, there are no time-critical sections in reading this way, so that other high priority subsystems are allowed to run (such as graphics or audio, both of which have stringent real time constraints).

If you don't have enough memory for track buffering (or for some other reason decide not to read a whole track at once), the disk hardware supports a limited set of sector searching facilities. There is a register that may be polled to examine the disk input stream.

There is a hardware bug that causes the last 3 bits of data sent to the disk to be lost. Also, the last word in a disk read DMA operation may not come in (that is, one less word may be read than you asked for).

8.3.2. Other Registers Involved with Disk Operations

The following registers are also associated with disk operations, as specified below.

DSKBYTR Disk Data Byte and Status Read

This register is the disk-microprocessor data buffer. Data from the disk, when in read mode, is loaded into this register one byte at a time. As each byte is received into this register, the BYTEREADY bit is set true. BYTEREADY is cleared each time the DSKBYTR register is read.

DSKBYTR is the register normally used by system software to synchronize the processor to the disk rotation before issuing a read or write under DMA control. The bits are shown in the following table.

BIT NUMBER	NAME	FUNCTION
15	BYTEREADY	Indicates that this register contains a valid byte of data. (Reset by reading this register)
14	DMAON	The DMA bit (in DSKLEN) is enabled and the DMACON bits are on, too. All DMA bits must be on for this to be true.
13	DISKWRITE	This disk write bit (in DSKLEN) is enabled.
12	WORDEQUAL	Indicates the DISKSYNC register equals the disk input stream. This bit is true only while the input stream matches the sync register (as little as two microseconds).
11-8	unused	
7-0	DATA	Disk byte data

ADKCON and ADKCONR Audio and Disk Control Register

ADKCON is the write address and ADKCONR is the read address.

The bottom 8 bits of this register are used for the audio circuitry. The other bits are shown in the following table.

BIT NUMBER	NAME	FUNCTION
15	CLR/SET	Same use as in the DMA enable register. Bit 15 must be a 1 if the register bits are to be set. Bit 15 is a 0 if the bits are to be cleared.
14	PRECOMP1	MSB of Precomp Specifier
13	PRECOMP0	LSB of Precomp Specifier Value of 00 selects None. Value of 01 selects 140 ns. Value of 10 selects 280 ns. Value of 11 selects 560 ns.
12	MFMPREC	Value of 0 selects GCR Precomp Value of 1 selects MFM Precomp
11	UARTBRK	Value of 1 forces the output of Paula's serial port to a zero (an RS-232 break).
10	WORDSYNC	Value of 1 enables synchronizing and starting of DMA on disk read of a word. The word to synchronize on must be written into the DSKSYNC address (DFF07E).
9	MSBSYNC	Value of 1 enables sync on MSBit (GCR).
8	FAST	Value of 1 selects two microseconds per bit cell (usually MFM), 0 selects four microseconds per bit (usually GCR).
7-4	ATPER3-0	Audio-attach period controls (not disk related).
3-0	ATVOL3-0	Audio-attach volume controls (not disk related).

One form of GCR format is that format used by the Apple[TM] computer. Data bytes on Apple-formatted disks always have the most significant bit set to a 1. When reading a GCR formatted disk, the software must use a translate table called a nibble-izer to assure that all data written to the disk conforms with this bit-setting. Bit 9, when a 1, tells the disk controller to look for this sync-bit on every disk byte.

DSKSYNC Disk input synchronizer

The DSKSYNC register is used to synchronize the input stream. If WORDEQUAL is enabled in ADKCON, no data is transferred to memory until a word is found in the input stream that matches the word in the DSKSYNC register. In addition, the DSKSYNC bit in INTREQ is set when the input stream matches the DSKSYNC register. The DSKSYNC bit in INTREQ is independent of the WORDEQUAL enable.

DSKDAT and DSKDATR Disk DMA Data Registers

[FOR TEST ONLY]

DSKDAT is write-only and DISKDATR is a read-only early-read dummy address.

This register is the disk DMA data buffer. It contains 2 bytes of data that are either sent to (write) or received from (read) the disk. The write mode is enabled by bit 14 of the DSKLEN register. The DMA controller automatically transfers data to or from this register and RAM.

8.3.3. Disk Interrupts

The disk controller can issue two kinds of interrupts. These interrupts are:

- DSKSYNC (level 5, INTREQ bit 12)—the input stream matches the DSKSYNC register
- DSKBLK (level 1, INTREQ bit 1)—disk DMA has completed.

Each of these is explained further in the sections titled "Length, Direction, DMA Enable" and "Other Registers Involved with Disk Operations" above. See Chapter 7, "System Control Hardware", for more information about interrupts.

8.4. THE KEYBOARD

The keyboard is interfaced to the system through one pair of lines connected to the odd-addressed 8520 CIA chip. These lines are·

CNT - keyboard clock (input from keyboard)
SP - keyboard data (input or output)

8.4.1. How the Keyboard Data is Received

The CNT line is used as a clock for the keyboard. On each transition of this line, one bit of data is clocked in from the keyboard. The keyboard sends this clock while each data bit which it wishes to send is stable on the SP line. The clock is an active low pulse. The rising edge of this pulse clocks in the data.

The 8520 is set up to use the CNT line as a clock and the SP line as a data input to an internal serial shift register. Appendix F contains most of the data-sheet for the 8520, and provides more information for interested parties.

After a data byte has been received from the keyboard, an interrupt (from the 8520) is issued to the processor. The keyboard waits for a handshake signal from the system before transmitting any more keystrokes. (The handshake is issued by the processor pulsing the SP line low for a minimum of 75 microseconds.)

If another keystroke is received before the previous one has been accepted by the processor, the keyboard-processor (internal to keyboard) holds a type-ahead buffer approximately 10 "keycodes" long. (Keycodes are explained in the next section).

8.4.2. Type of Data Received

The keyboard data is NOT received in the form of ASCII characters. Instead, for maximum versatility, it is received in the form of keycodes. These codes include not only the down-transition of the key, but also the up-transition. This allows your software to use both sets of information to determine exactly what is happening on the keyboard.

Here is a list of the hexadecimal values which are assigned to the keyboard. A downstroke of the key transmits the value shown here. An upstroke of the key transmits this value + $80. The picture of the keyboard at the end of this section shows the positions which correspond to the description in the paragraphs below.

The 128 possible key codes are arranged into the logical groups shown below.

00-3F (hex)

These are key codes assigned to specific positions on the main body of the keyboard and numeric pad which contain graphic keys (that is, "A", not "Tab"). The key positions would

generally be labeled with country dependent keys. These keycodes are best described positionally as shown in Figure 8-3 at the end of the keyboard section.

40-4F (hex)

These are key codes with specific meanings common to most keyboards:

40	Space
41	Backspace
42	Tab
43	Enter (numeric pad)
44	Return
45	Escape
46	Delete
4A	Numeric Pad
4C	Cursor Up
4D	Cursor Down
4E	Cursor Forward
4F	Cursor Backward

50-5F (hex)

key codes for function keys:

50-59	Function keys F1-F10
5F	Help

60-67 (hex)

key codes for qualifier keys:

60 Left Shift
61 Right Shift
62 Caps Lock
63 Control
64 Left Alt
65 Right Alt
66 Left Command
67 Right Command

68-77 (hex)

Unassigned.

F0-FF (hex)

These key codes are used for 6500/01-68000 communication, and are not associated with a keystroke. They have no key transition flag, and are therefore described completely by 8 bit codes.

F9 last key code bad, next key is same code retransmitted
FA keyboard key buffer overflow
FC keyboard self-test fail
FD initiate power-up key stream (for stuck keys)
FE terminate key stream (from FD)

These key codes may be filtered out by the drivers.

8.4.3. Limitations of the Keyboard

The Amiga keyboard is a matrix of rows and columns, with a key switch at each intersection. Because of this, it is subject to a phenomenon called "ghosting".

Ghosting means that certain combinations of keys pressed simultaneously will cause extra ("ghost") key codes to be transmitted. For example, press "A" and "S" simultaneously and hold them down. Notice that "A" and "S" are transmitted. While still holding them down, press "Z" and observe that both "X" and "Z" are transmitted. In this case, "X" is a ghost key.

The keyboard is designed so that this will never happen during normal typing, only during unusual key combinations like the one just described. Normally, the keyboard will appear to have "N-key rollover", which means that you will run out of fingers before generating a ghost character.

NOTE: There are seven keys that are not part of the matrix, and thus do not contribute to generating ghosts. These keys are: CTRL, the two SHIFT keys, the two AMIGA keys, and the two ALT keys.

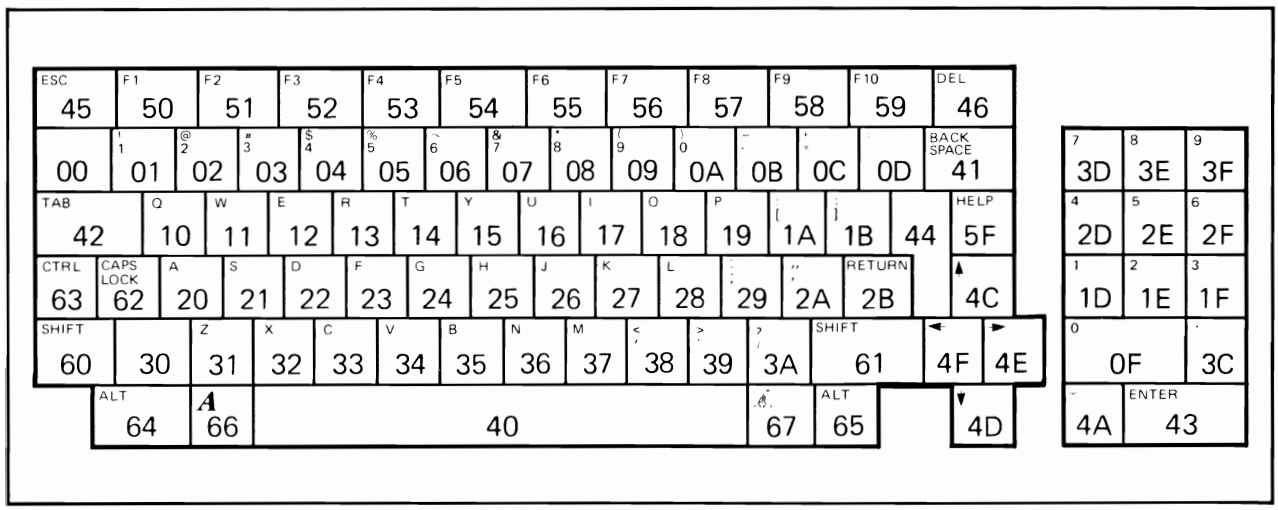


Figure 8-3: The Amiga Keyboard, Showing Keycodes in Hex

8.5. PARALLEL INPUT/OUTPUT INTERFACE

The general-purpose parallel interface is a 25-pin male connector on the back panel of the computer. This connector is generally used for a parallel printer.

For pin connections, see Appendix E.

8.6. SERIAL INTERFACE

There is a 25-pin D-type female connector on the back panel of the computer which serves as the general purpose serial interface. This connector can drive a wide range of different peripherals, including an external modem or a serial printer.

For pin connections, see Appendix E.

8.6.1. Introduction to Serial Circuitry

The circuit that controls the serial link to the outside world is called a UART, which is short for Universal Asynchronous Receiver/Transmitter. The UART is able to communicate at baud rates (bit-rate of transmission of data) which you preset. It can receive or send data with a programmable length of 8 or 9 bits.

It is also capable of detecting overrun errors. These occur when some other system sends in data faster than you remove it from the data receive register. There are also status bits that you can read to find out when the receive buffer is full, or when the transmit buffer is empty. An additional status bit is also provided which indicates "all bits sent". All of these topics are discussed below.

8.6.2. Setting the Baud Rate

Baud rate (rate of transmission) is controlled by the contents of the register named SERPER. Bits 14-0 of SERPER are the baud-rate divider bits. If you consider the contents of these bits to be the number N, then N+1 color clocks (each 279.4 ns) occur between each sample of the state of the input pin (for receive) or between transmissions of output bits (in the transmit mode).

8.6.3. Setting the Receive Mode

You have a choice of how many bits are to be received before the system tells you that the receive register is full. You may define either 8 or 9 bits for the receive register full signal. In either case, the receive circuitry expects to see:

- one start bit
- 8 or 9 data bits
- at least one stop bit

Receive mode is set by bit 15 of SERPER. Bit 15 is a 1 if 9 data bits, and a 0 if 8 data bits The normal state of this bit for most receive applications is a 0.

SERPER is a write-only register.

8.6.4. Contents of the Receive Data Register

The serial input data receive register is 16-bits wide. It contains not only the input data received, but also certain status bits which are explained below.

The data bit positions defined for read-data are taken from the "backup" register which is connected to the receive-data serial shift register.

The data is received, one bit at a time, into a serial-to-parallel shift register. When the proper number of bits has been received, the contents of this register are transferred to the serial data read register (SERDATR) shown below, and you are signaled that there is data ready for you.

It is called a backup register because immediately after the transfer of data takes place, the receive shift register again becomes ready to accept new data. You will therefore have up to one full character-receive time (8 to 10 bit times) after receiving the receiver-full interrupt to accept the data and clear the interrupt.

The following table shows the definitions of the various bit positions within SERDATR.

BIT NUMBER	NAME	FUNCTION
15	OVRUN	OVERRUN bit (Mirror—also appears in the interrupt request register). Indicates that another byte of data has been received before the previous byte has been picked up by the processor. To prevent this condition, it is necessary to reset the RBF (Bit 11) (Receive-Buffer-Full) bit in the interrupt request register (INTREQ).
14	RBF	READ BUFFER FULL (Mirror—also appears in the interrupt request register). When it is a 1, it says that there is data ready to be picked up by the processor. After reading the contents of this data register, you must reset the RBF bit in INTREQ to prevent an overrun.
13	TBE	TRANSMIT BUFFER EMPTY

		(Not a mirror—interrupt occurs when the buffer *becomes* empty. When it is a 1, the data in the output data register (SERDAT) has been transferred to the serial output shift register, so SERDAT is ready to accept another output word. This is also true when the buffer *is* empty. This bit is normally used for full duplex operations.
12	TSRE	TRANSMIT SHIFT REGISTER EMPTY When this bit a 1, the output shift register has completed its task and all data has been transmitted and it is now idle. If you stop writing data into the output register (SERDAT), then this bit will become a 1 after both the word currently in the shift register AND the word placed into SERDAT have been transmitted. This bit is normally used for half-duplex operation.
11	RXD	Direct read of RXD pin on Paula chip.
10		Not used at this time
9	STP	Stop bit if 9 data bits are specified for receive.
8	STP	Stop bit if 8 data bits are specified for receive, OR
	DB8	9th data bit if 9 bits are specified for receive.
7-0	DB7-DB0	Low 8 data bits of received data. Data is TRUE (data you read is the same polarity as the data expected).

8.6.5. How Output Data is Transmitted

You send data out on the transmit lines by writing into the serial data output register (SERDAT). This register is write-only.

Data will be sent out at the same rate as you have established for the read, and is contained in the serial data period register (SERPER) shown above. Immediately after you write the data into this register, the system will begin the transmission at the baud rate you selected.

At the start of the operation, this data is transferred from SERDAT into a serial shift register. When the transfer to the serial shift register has been completed, SERDAT can accept new data. The TBE interrupt signals this fact.

Data will be moved out of the shift register, one bit each time interval, starting with the least significant bit. The shifting continues until, following the last shift, the UART detects the condition shift-register-empty. This condition is all-zeros remaining in the register.

This is a 16-bit register which allows you to control the format (appearance) of the transmitted data. To form a typical data sequence, such as 1 start-bit, 8 data bits, and 2 stop bits, you write the following contents into SERDAT.

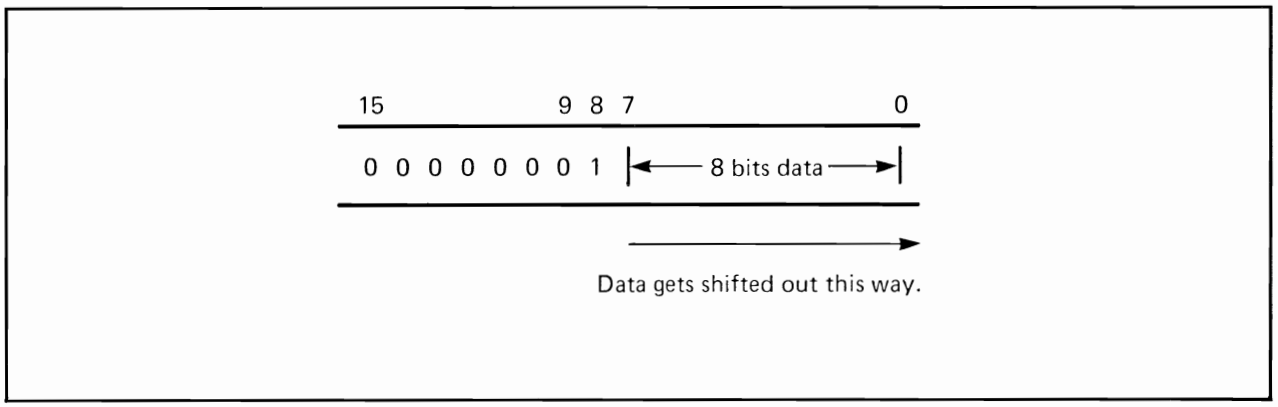


Figure 8-4: Starting appearance of SERDAT, and shift register

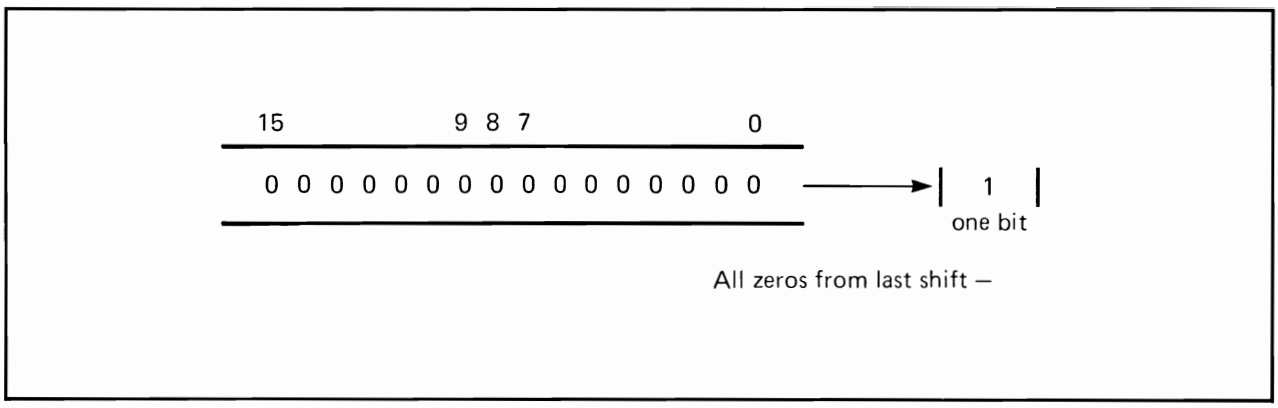


Figure 8-5: Ending appearance of shift register

The register stops shifting and signals "shift register empty" (TSRE) when there is a 1-bit present in the bit-shifted-out position AND the rest of the contents of the shift register are zeros. When new non-zero contents are transferred into this register, shifting begins again.

8.6.6. How to Specify the Register Contents

You should write the data you wish to transmit as the low 8 (or 9 if you wish) bits of this output register (SERDAT). Above the data bits (in bits 8 and above, or bit 9 and above) you write 1-bits for however many stop bits you transmit with the data. Normally, you send either 1 or 2 stop bits. (See the figure above, "Starting appearance of SERDAT").

The transmission of the start bit is independent of the contents of this register. One start bit is automatically generated before the first (bit 0) bit of the data is sent.

Writing this register starts the data transmission. If this register is written with all zeros, NO data transmission is initiated.

8.7. AUDIO OUTPUT CONNECTIONS

The Amiga has two different forms of audio output for the audio channels:

a. Stereo Output Jacks

A pair of "RCA" jacks, designed to be connected to a stereo amplifier.

b. RF-Audio

The channel 3/4 RF modulator will provide sound through the speaker of your television set when the television is used to provide the computer's display. Both channels of audio are provided at this connector. However, the RF modulator on initial shipments of Amiga computers combines the signals and transmits monaural sound.

8.8. DISPLAY OUTPUT CONNECTIONS

There is a 23-pin connector on the back of the Amiga which contains signals for two different types of video output. A separate cable assembly will be made up for each different type of video. The types are:

- RGB Monitors ("Analog RGB")

  Provides 4 outputs, specifically R, G, B, and Sync.

  Can generate up to 4096 different colors on-screen simultaneously using the circuitry presently available on the Amiga.

- DIGITAL RGB Monitors

  Provides 4 outputs, distinct from those shown above. Also named R, G, B, I, and Sync. All output levels are logic levels (0 or 1).

  R is Red-ON; G is Green-ON; B is Blue-ON; I is half-intensity.

  This allows up to 15 possible color combinations, where the values 0000 and 0001 map to the same output value. (Half intensity with no color present is the same as full intensity, no color.)

APPENDIX A - Amiga Hardware Manual

This appendix contains a short summary, in alphabetical order, of the register set and the usages of the individual bits.

The addresses shown here are used by the special chips for transferring data betweem themselves. Also, the Copper uses these addresses for writing to the special chip registers. In order to write to these registers wth the 68000, calculate the 68000 address using this formula:

68000 address = (chip address) + $DFF000

For example, for the 68000 to write to ADKCON (address = $09E), the 68000 would write to $DFF09E.

REGISTER	ADDR.	READ/ WRITE	AGNES/ DENISE/ PAULA	FUNCTION
ADKCON	09E	W	P	Audio, Disk, Control write
ADKCONR	010	R	P	Audio, disk, Control read

BIT#	USE	
15	SET/CLR	Set/Clear control bit. Determines if bits written with a 1 get set or cleared. Bits written with a zero are always unchanged.
14-13	PRECOMP 1-0	CODE PRECOMP VALUE 00 none 01 140 ns 10 280 ns 11 560 ns
12	MFMPREC	(1=MFM precomp 0=GCR precomp)
11	UARTBRK	Forces a UART break (clears TXD) if true
10	WORDSYNC	Enables disk read synchronizing on a word equal to DISK SYNC CODE, located in address (3F)*2
09	MSBSYNC	Enables disk read synchronizing on the MSB (most signif bit). Appl type GCR
08	FAST	Disk data clock rate control 1=fast(2us) 0=slow(4us) (fast for MFM, slow for MFM or GCR)
07	USE3PN	Use audio channel 3 to modulate nothing
06	USE2P3	Use audio channel 2 to modulate period of channel 3
05	USE1P2	Use audio channel 1 to modulate period of channel 2
04	USE0P1	Use audio channel 0 to modulate period of channel 1
03	USE3VN	Use audio channel 3 to modulate nothing
02	USE2V3	Use audio channel 2 to modulate volume of channel 3
01	USE1V2	Use audio channel 1 to modulate volume of channel 2
00	USE0V1	Use audio channel 0 to modulate volume of channel 1

NOTE: If both period and volume are modulated on the same channel, the period and volume will be alternated. First word xxxxxxxx V6-V0 , Second word P15-P0 (etc)

AUDxLCH	0A0	W	A	Audio channel x location (High 3 bits)
AUDxLCL	0A2	W	A	Audio channel x location (Low 15 bits)

This pair of registers contains the 18 bit starting address (location) of Audio channel x (x=0,1,2,3) DMA data. This is not a pointer register and therefore only needs to be reloaded if a different memory location is to be outputted.

AUDxLEN	0A4	W	P	Audio Channel x length

This register contains the length (number of words) of Audio Channel x DMA data.

AUDxPER	0A6	W	P	Audio channel x Period

This register contains the Period (rate) of Audio channel x DMA data transfer. The minimum period is 124 color clocks. This means that the smallest number that should be placed in this register is 124 decimal. This corresponds to a maximum sample frequency of 28.86 khz.

AUDxVOL	0A8	W	P	Audio Channel x Volume

This register contains the Volume setting for Audio Channel x. Bits 6,5,4,3,2,1,0 specify 65 linear volume levels as shown below.

BITS	USE
15-07	Not used
06	Forces volume to max (64 ones, no zeros)
05-00	Sets one of 64 levels (000000=no output (111111=63 Ones, one zero)

AUDxDAT	0AA	W	P	Audio channel x Data

This register is the Audio channel x (x=0,1,2,3) DMA data buffer. It contains 2 bytes of data that are each 2's complement and are outputted sequentially (with digital to analog conversion) to the audio output pins. (LSB = 3 MV) The DMA controller automatically transfers data to this register from RAM. The processor can also write directly to this register. When the DMA data is finished (words outputted=Length) and the data in this register has been used, an audio channel interrupt request is set.

BLTxPTH	050	W	A	Blitter Pointer to x (High 3 bits)
BLTxPTL	052	W	A	Blitter Pointer to x (Low 15 bits)

This pair of registers contains the 18 bit address of Blitter source (x=A,B,C) or dest. (x=D) DMA data. This pointer must be preloaded with the starting address of the data to be processed by the blitter. After the Blitter is finished it will contain the last data address (plus increment and modulo).

LINE DRAW BLTAPTL is used as an accumulator register
LINE DRAW and must be preloaded with the starting value
LINE DRAW of (2Y-X) where Y/X is the line slope.

LINE DRAW BLTCPT and BLTDPT (both H and L) must be
LINE DRAW preloaded with the starting address of the line.

BLTxMOD 064 W A Blitter Modulo x

This register contains the Modulo for Blitter source(x=A,B,C) or Dest (x=D). A Modulo is a number that is automatically added to the address at the end of each line, in order that the address then points to the start of the next line. Each source or destination has its own Modulo, allowing each to be a different size, while an identical area of each is used in the Blitter operation.

LINE DRAW BLTAMOD and BLTBMOD are used as slope storage
LINE DRAW registers and must be preloaded with the values
LINE DRAW (4Y-4X) and (4Y) respectivly. Y/X= line slope
LINE DRAW BLTCMOD and BLTDMOD must both be preloaded
LINE DRAW with the width (in bytes) of the image into
LINE DRAW which the line is being drawn.(normally 2 times
LINE DRAW the screen width in words)

BLTAFWM 044 W A Blitter first word mask for source A
BLTALWM 046 W A Blitter last word mask for source A

The patterns in these two registers are "anded" with the first and last words of each line of data from Source A into the Blitter. A zero in any bit overrides data from Source A. These registers should be set to all "ones" for fill mode or for line drawing mode.

BLTxDAT 074 W A Blitter source x data reg

This register holds Source x (x=A,B,C) data for use by the Blitter. It is normally loaded by the Blitter DMA channel, however it may also be preloaded by the microprocessor.

LINE DRAW BLTADAT is used as an index register and must
LINE DRAW be preloaded with 8000.
LINE DRAW BLTBDAT is used for texture, It must be preloaded
LINE DRAW with FF if no texture (solid line) is desired.

BLTDDAT Blitter destination data register

This register holds the data resulting from each word of Blitter operation until it is sent to a RAM destination. This is a dummy address and cannot be read by the micro. The transfer is automatic during Blitter operation.

BLTCONO 040 W A Blitter control register 0
BLTCON1 042 W A Blitter control register 1

These two control registers are used together to control Blitter operations. There are 2 basic modes, area and line, which are selected by bit 0 of BLTCON1, as shown below.

AREA MODE ("normal")

BIT#	BLTCONO	BLTCON1
15	ASH3	BSH3
14	ASH2	BSH2
13	ASH1	BSH1

12	ASA0	BSH0
11	USEA	X
10	USEB	X
09	USEC	X
08	USED	X
07	LF7	X
06	LF6	X
05	LF5	X
04	LF4	EFE
03	LF3	IFE
02	LF2	FCI
01	LF1	DESC
00	LF0	LINE(=0)

ASH3-0 Shift value of A source
BSH3-0 Shift value of B source
USEA Mode control bit to use Source A
USEB Mode control bit to use Source B
USEC Mode control bit to use Source C
USED Mode control bit to use Destination D
LF7-0 Logic function minterm select lines
EFE Exclusive fill enable
IFE Inclusive fill enable
FCI Fill carry input
DESC Descending (decreasing address)control bit
LINE Line mode control bit (set to 0)

LINE DRAW LINE MODE (line draw)

BIT#	BLTCONO	BLTCON1
15	START3	0
14	START2	0
13	START1	0
12	START0	0
11	1	0
10	0	0
09	1	0
08	1	0
07	LF7	0
06	LF6	SIGN
05	LF5	OVF
04	LF4	SUD
03	LF3	SUL
02	LF2	AUL
01	LF1	SING
00	LF0	LINE(=1)

LINE DRAW
LINE DRAW START3-0 Starting point of line (0 thru 15 hex)
LINE DRAW LF7-0 Logic function minterm select lines
LINE DRAW should be preloaded with 4A in order
LINE DRAW
LINE DRAW to select the equation $D=(\bar{A}C+AB\bar{C})$. Since A
LINE DRAW contains a single bit true (8000), most bits
LINE DRAW will pass the C field unchanged (not A and C),

```
                LINE DRAW     but one bit will invert the C Field and
                LINE DRAW     combine it with texture (A and B and not C).
                LINE DRAW     The A bit is automatically moved across the
                LINE DRAW     word by the hardware.
                LINE DRAW
                LINE DRAW     LINE   Line mode control bit (set to 1)
                LINE DRAW     SIGN   Sign flag
                LINE DRAW     OVF    Word overflow flag
                LINE DRAW     SING   Single bit per horiz. line
                LINE DRAW            for use with subsequent Area Fill
                LINE DRAW     SUD    Sometimes Up or Down (=AUD*)
                LINE DRAW     SUL    Sometimes Up or Left
                LINE DRAW     AUL    Always Up or Left
                LINE DRAW     The 3 bits above select the Octant for
                LINE DRAW     line draw:
                LINE DRAW           OCT     SUD  SUL  AUL
                LINE DRAW           ---     ---  ---  ---
                LINE DRAW            0       1    1    0
                LINE DRAW            1       0    0    1
                LINE DRAW            2       0    1    1
                LINE DRAW            3       1    1    1
                LINE DRAW            4       1    0    1
                LINE DRAW            5       0    1    0
                LINE DRAW            6       0    0    0
                LINE DRAW            7       1    0    0

BLTSIZE     058 W   A       Blitter start and size (window width, height)
                This register contains the width and height of the blitter
                operation (in line mode width must =2, height = line length)
                Writing to this register will start the Blitter, and should
                be done last, after all pointers and control registers have
                been initialized.
                BIT#  15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
                      h9 h8 h7 h6 h5 h4 h3 h2 h1 h0,w5 w4 w3 w2 w1 w0
                h=Height=Vertical lines (10 bits=1024 lines max)
                w=Width =Horiz pixels (6 bits=64 words=1024 pixels max)
                LINE DRAW   BLTSIZE controls the line length and starts
                LINE DRAW   the line draw when written to.  The h field
                LINE DRAW   controls the line length (10 bits gives
                LINE DRAW   lines up to 1024 dots long).  The w field
                LINE DRAW   must be set to 02 for all line drawing.

BPLxPTH     0E0 W   A       Bit plane x pointer (High 3 bits)
BPLxPTL     0E2 W   A       Bit plane x pointer (Low 15 bits)
                This pair of registers contains the 18 bit pointer to the
                address of Bit plane x (x=1,2,3,4,5,6) DMA data. This pointer
                must be reinitialized by the processor or Copper
                to point to the beginning of Bit Plane data every vertical
                blank time.

BPLxDAT     110 W     D     Bit plane x data (Parallel to serial convert)
                These registers receive the DMA data fetched from RAM
                by the Bit Plane address pointers described above.
                They may also be written by either micro.
                They act as a 6-word parallel-to-serial buffer
                for up to 6 memory "Bit Planes". (x=1-6) The parallel to
```



```
                serial conversion is triggered whenever bit plane #1
                is written, indicating the completion of all bit planes
                for that word (16 pic-cells). The MSB is output first, and
                is therefore always on the left.

BPL1MOD     108 W   A       Bit plane modulo (odd planes)
BPL2MOD     10A W   A       Bit Plane modulo (even planes)
                These registers contain the Modulos for the odd and even
                bit planes.  A Modulo is a number that is automatically added
                to the address at the end of each line, in order that
                the address then points to the start of the next line.
                Since they have separate modulos, the odd and even
                bit planes may have sizes that are different from each
                other, as well as different from the Display Window size.

BPLCON0     100 W   A D     Bit plane control reg.(misc control bits)
BPLCON1     102 W     D     Bit plane control reg.(horiz. scroll control)
BPLCON2     104 W     D     Bit Plane control reg.(video priority control)
                These registers control the operation of the Bit Planes
                and various aspects of the display.
                BIT#    BPLCON0    BPLCON1    BPLCON2
                ----    --------   --------   --------
                15      HIRES       X          X
                14      BPU2        X          X
                13      BPU1        X          X
                12      BPUO        X          X
                11      HOMOD       X          X
                10      DBLPF       X          X
                09      COLOR       X          X
                08      GAUD        X          X
                07       X         PF2H3       X
                06       X         PF2H2      PF2PRI
                05       X         PF2H1      PF2P2
                04       X         PF2HO      PF2P1
                03      LPEN       PF1H3      PF2PO
                02      LACE       PF1H2      PF1P2
                01      ERSY       PF1H1      PF1P1
                00       X         PF1HO      PF1PO
                HIRES=High resolution (640) mode
                BPU  =Bit plane use code 000-110 (NONE through 6 inclusive)
                HOMOD=Hold and Modify mode
                DBLPF=Double playfield (PF1=odd PF2=even bit planes)
                COLOR=Composite video COLOR enable
                GAUD=Genlock audio enable (muxed on BKGND pin during
                       vertical blanking
                LPEN =Light pen enable (reset on power up)
                LACE =Interlace enable (reset on power up)
                ERSY =External Resync (HSYNC, VSYNC pads become inputs)
                       (reset on power up)
                PF2PRI=Playfield 2 (even planes) has priority over
                       (appears in front of) Playfield 1 (odd planes).
                PF2P=Playfield 2 priority code (with respect to sprites)
                PF1P=Playfield 1 priority code (with respect to sprites)
                PF2H=Playfield 2 horizontal scroll code
                PF1H=Playfield 1 horizontal scroll code
```

CLXCON 098 W D Collision control

This register controls which Bitplanes are included (enabled) in collision detection, and their required state if included. It also controls the individual inclusion of odd numbered sprites in the collision detection, by logically OR-ing them with their corresponding even numbered sprite.

BIT#	FUNCTION	DESCRIPTION
15	ENSP7	Enable Sprite 7 (ORed with Sprite 6)
14	ENSP5	Enable Sprite 5 (ORed with Sprite 4)
13	ENSP3	Enable Sprite 3 (ORed with Sprite 2)
12	ENSP1	Enable Sprite 1 (ORed with Sprite 0)
11	ENBP6	Enable Bit Plane 6 (Match reqd. for collision)
10	ENBP5	Enable Bit Plane 5 (Match reqd. for collision)
09	ENBP4	Enable Bit Plane 4 (Match reqd. for collision)
08	ENBP3	Enable Bit Plane 3 (Match reqd. for collision)
07	ENBP2	Enable Bit Plane 2 (Match reqd. for collision)
06	ENBP1	Enable Bit Plane 1 (Match reqd. for collision)
05	MVBP6	Match value for Bit Plane 6 collision
04	MVBP5	Match value for Bit Plane 5 collision
03	MVBP4	Match value for Bit Plane 4 collision
02	MVBP3	Match value for Bit Plane 3 collision
01	MVBP2	Match value for Bit Plane 2 collision
00	MVBP1	Match value for Bit Plane 1 collision

NOTE: Disabled Bit Planes cannot prevent collisions. Therefore if all bit planes are disabled, collisions will be continuous, regardless of the match values.

CLXDAT 00E R D Collision data reg. (Read and clear)

This address reads (and clears) the collision detection register. The bit assignments are below.

NOTE: Playfield 1 is all odd numbered enabled bit planes. Playfield 2 is all even numbered enabled bit planes

BIT#	COLLISIONS REGISTERED
15	not used
14	Sprite 4 (or 5) to Sprite 6 (or 7)
13	Sprite 2 (or 3) to Sprite 6 (or 7)
12	Sprite 2 (or 3) to Sprite 4 (or 5)
11	Sprite 0 (or 1) to Sprite 6 (or 7)
10	Sprite 0 (or 1) to Sprite 4 (or 5)
09	Sprite 0 (or 1) to Sprite 2 (or 3)
08	Playfield 2 to Sprite 6 (or 7)
07	Playfield 2 to Sprite 4 (or 5)
06	Playfield 2 to Sprite 2 (or 3)
05	Playfield 2 to Sprite 0 (or 1)
04	Playfield 1 to Sprite 6 (or 7)
03	Playfield 1 to Sprite 4 (or 5)
02	Playfield 1 to Sprite 2 (or 3)
01	Playfield 1 to Sprite 0 (or 1)
00	Playfield 1 to Playfield 2

COLORxx 180 W D Color table xx

There are 32 of these registers (xx=00-31) and they are sometimes collectively called the "Color Palette".

They contain 12 bit codes representing RED,GREEN,BLUE colors for "RGB systems. One of these registers at a time is selected (by the BPLxDAT serialized video code) for presentation at the RGB video output pins.

The table below shows the color register bit usage.

BIT#	15,14,13,12	11,10,09,08	07,06,05,04	03,02,01,00
RGB	X X X X	R3 R2 R1 R0	G3 G2 G1 G0	B3 B2 B1 B0

B=blue, G=green, R=red,

COPCON 02E W A Copper control register

This is a 1 bit register that when set true, allows the Copper to access the Blitter hardware. This bit is cleared by power on reset, so that the Copper cannot access the Blitter hardware.

BIT#	NAME	FUNCTION
01	CDANG	Copper danger mode. Allows Copper access to Blitter if true.

COPJMP1 088 S A Copper restart at first location
COPJMP2 08A S A Copper restart at second location

These addresses are strobe addresses, that when written to cause the Copper to jump indirect using the address contained in the First or Second Location registers described below. The Copper itself can write to these addresses, causing its own jump indirect.

COP1LCH 080 W A Copper first location reg (High 3 bits)
COP1LCL 082 W A Copper first location reg. (Low 15 bits)
COP2LCH 084 W A Copper second location reg. (High 3 bits)
COP2LCL 086 W A Copper second location reg(Low 15 bits)

These registers contain the jump addresses described above.

COPINS 08C W A Copper inst. fetch identify

This is a dummy address that is generated by the Copper whenever it is loading instructions into its own instruction register. This actually occurs every Copper cycle except for the second (IR2) cycle of the MOVE instruction. The Three types of instructions are shown below.

MOVE Move immediate to dest
WAIT Wait until beam counter is equal to, or greater than. (keeps Copper off of bus until beam position has been reached)
SKIP Skip if beam counter is equal to, or greater than. (skips following MOVE inst. unless beam position has been reached)

	MOVE		WAIT UNTIL		SKIP IF	
BIT#	IR1	IR2	IR1	IR2	IR1	IR2
15	X	RD15	VP7	BFD *	VP7	BFD *
14	X	RD14	VP6	VE6	VP6	VE6
13	X	RD13	VP5	VE5	VP5	VE5

12	X	RD12	VP4	VE4	VP4	VE4
11	X	RD11	VP3	VE3	VP3	VE3
10	X	RD10	VP2	VE2	VP2	VE2
09	X	RD09	VP1	VE1	VP1	VE1
08	DA8	RD08	VP0	VE0	VP0	VE0
07	DA7	RD07	HP8	HE8	HP8	HE8
06	DA6	RD06	HP7	HE7	HP7	HE7
05	DA5	RD05	HP6	HE6	HP6	HE6
04	DA4	RD04	HP5	HE5	HP5	HE5
03	DA3	RD03	HP4	HE4	HP4	HE4
02	DA2	RD02	HP3	HE3	HP3	HE3
01	DA1	RD01	HP2	HE2	HP2	HE2
00	0	RD00	1	0	1	1

IR1=First instruction register
IR2=Second instruction register
DA =Destination Address for MOVE instruction. Fetched during IR1 time, used during IR2 time on RGA bus.
RD =RAM Data moved by MOVE instruction at IR2 time directly from RAM to the address given by the DA field.
VP =Vertical Beam Position comparison bit
HP =Horizontal Beam Position comparison bit
VE =Enable comparison(mask bit)
HE =Enable comparison(mask bit)
* NOTE BFD=Blitter finished disable. When this bit is true, the Blitter Finished flag will have no effect on the Copper. When this bit is zero the Blitter Finished flag must be true (in addition to the rest of the bit comparisons) before the Copper can exit from its wait state, or skip over an instruction. Note that the V7 comparison cannot be masked.

The Copper is basically a 2 cycle machine that requests the bus only during odd memory cycles. (4 memory cycles per in) This prevents collisions with Display, Audio, Disk, Refresh, and Sprites, all of which use only even cycles. It therefore needs (and has) priority over only the Blitter and Micro.

There are only three types of instructions: MOVE immediate, WAIT until, and SKIP if. All instructions (except for WAIT) require 2 bus cycles (and two instruction words). Since only the odd bus cycles are requested, 4 memory cycle times are required per instruction. (memory cycles are 280 ns)

There are two indirect jump registers COP1LC and COP2LC. These are 18 bit pointer registers whose contents are used to modify the program counter for initialization or jumps. They are transferred to the program counter whenever strobe addresses COPJMP1 or COPJMP2 are written. In addition COP1LC is automatically used at the beginning of each vertical blank time.

It is important that one of the jump registers be initialized and its jump strobe address hit, after power up but before

Copper DMA is initialized. This insures a determined startup address and state.

DIWSTRT 08E W A Display Window Start (upper left vert-hor pos)
DIWSTOP 090 W A Display Window Stop (lower right vert-hor pos)
These registers control the Display window size and position, by locating the upper left and lower right corners.
BIT# 15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
USE V7 V6 V5 V4 V3 V2 V1 V0 H7 H6 H5 H4 H3 H2 H1 H0
DIWSTRT is vertically restricted to the upper 2/3 of the display (V8=0), and horizontally restricted to the left 3/4 of the display(H8=0).
DIWSTOP is vertically restricted to the lower 1/2 of the display (V8=/=V7), and horizontally restricted to the right 1/4 of the display(H8=1).

DDFSTRT 092 W A Display data fetch start(Horiz.Position)
DDFSTOP 094 W A Display data fetch stop(Horiz.Position)
These registers control the horizontal timing of the beginning and end of the Bit Plane DMA display data fetch. The vertical Bit Plane DMA timing is identical to the Display windows described above.
The Bit Plane Modulos are dependent on the Bit Plane horizontal size, and on this data fetch window size.
Register bit assignment
BIT# 15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
USE X X X X X X X X H8 H7 H6 H5 H4 H3 X X
(X bits should always be driven with 0 to maintain upward compatibility)

The tables below show the start and stop timing for different register contents.
DDFSTRT(Left edge of display data fetch)

PURPOSE	H8,H7,H6,H5,H4
Extra wide (max) *	0 0 1 0 1
wide	0 0 1 1 0
normal	0 0 1 1 1
narrow	0 1 0 0 0

DDFSTOP(Right edge of display data fetch)

PURPOSE	H8,H7,H6,H5,H4
narrow	1 1 0 0 1
normal	1 1 0 1 0
wide (max)	1 1 0 1 1

DMACON 096 W A D P DMA control write(clear or set)
DMACONR 002 R A P DMA control (and blitter status) read
This register controls all of the DMA channels, and contains Blitter DMA status bits.
BIT# FUNCTION DESCRIPTION
---- --------- ----------

15	SET/CLR	Set/Clear control bit. Determines if bits written with a 1 get set or cleared. Bits written with a zero are unchanged.
14	BBUSY	Blitter busy status bit (read only)
13	BZERO	Blitter logic zero status bit.(read only)
12	X	
11	X	
10	BLTPRI	Blitter DMA priority(over CPU micro) (also called "Blitter Nasty") (disables /BLS pin, preventing micro from stealing any bus cycles while blitter DMA is running)
09	DMAEN	Enable all DMA below
08	BPLEN	Bit Plane DMA enable
07	COPEN	Copper DMA enable
06	BLTEN	Blitter DMA enable
05	SPREN	Sprite DMA enable
04	DSKEN	Disk DMA enable
03	AUD3EN	Audio channel 3 DMA enable
02	AUD2EN	Audio channel 2 DMA enable
01	AUD1EN	Audio channel 1 DMA enable
00	AUDOEN	Audio channel 0 DMA enable

DSKPTH 020 W A Disk pointer (High 3 bits)
DSKPTL 022 W A Disk pointer (Low 15 bits)

This pair of registers contains the 18 bit address of Disk DMA data. These address registers must be initialized by the processor or Copper before disk DMA is enabled.

DSKLEN 024 W P Disk length

This register contains the length (number of words) of Disk DMA data. It also contains 2 control bits. These are a DMA enable bit, and a DMA direction(read/write) bit.

BIT#		
15	DMAEN	Disk DMA Enable
14	WRITE	Disk Write(RAM to Disk) if 1
13-0	LENGTH	Length (# of words) of DMA data.

DSKDAT 026 W P Disk DMA Data write
DSKDATR 008 ER P Disk DMA Data read (early read dummy address)

This register is the Disk-DMA data buffer. It contains 2 bytes of data that are either sent to (write) or received from (read) the disk. The write mode is enabled by bit 14 of the LENGTH register. The DMA controller automatically transfers data to or from this register and RAM, and when the DMA data is finished (Length=0) it causes a Disk Block Interrupt. See interrupts below.

DSKBYTR 01A R P Disk Data byte and status read

This register is the Disk-Microprocessor data buffer. Data from the disk (in read mode) is loaded into this register one byte at a time, and bit 15 (DSKBYT) is set true.

BIT#		
15	DSKBYT	Disk byte ready (reset on read)

14	DMAON	Mirror of bit 15 (DMAEN) in DSKLEN, and-ed with Bit09 (DMAEN) in DMACON
13	DISKWRITE	Mirror of bit 14 (WRITE) in DSKLEN
12	WORDEQUAL	This bit true only while DSKSYNC register equals the data from disk
11-08	X	Not used
07-00	DATA	Disk byte data

DSKSYNC 07E W P Disk sync register, holds the match code for disk read synchronization. See ADKCON bit 10.

INTREQ 09C W P Interrupt Request bits (clear or set)
INTREQR 01E R P Interrupt request bits (read)

This register contains interrupt request bits(or flags). These bits may be polled by the processor, and if enabled by the bits listed in the next register, they may cause processor interrupts. Both a set and clear operation are required to load arbitrary data into this register. These status bits are not automatically reset when the interrupt is serviced, and must be reset when desired by writing to this address. The bit assignments are identical to the Enable register below.

INTENA 09A W P Interrupt Enable bits (clear or set bits)
INTENAR 01C R P Interrupt Enable bits Read

This register contains interrupt enable bits. The bit assignment for both the request, and enable registers is given below.

BIT#	FUNCT	LEVEL	DESCRIPTION
15	SET/CLR		Set/Clear control bit. Determines if bits written with a 1 get set or cleared. Bits written with a zero are always unchanged.
14	INTEN		Master interrupt (enable only , no request)
13	EXTER	6	External interrupt
12	DSKSYN	5	Disk Sync register(DSKSYNC)matches Disk data
11	RBF	5	Serial port Receive Buffer Full
10	AUD3	4	Audio channel 3 block finished
09	AUD2	4	Audio channel 2 block finished
08	AUD1	4	Audio channel 1 block finished
07	AUDO	4	Audio channel 0 block finished
06	BLIT	3	Blitter finished
05	VERTB	3	Start of Vertical blank
04	COPER	3	Copper
03	PORTS	2	I/O Ports and timers
02	SOFT	1	Reserved for software initiated interrupt.
01	DSKBLK	1	Disk Block finished
00	TBE	1	Serial port Transmit Buffer Empty

JOYODAT 00A R D Joystick-mouse 0 data (left vert, horiz)
JOY1DAT 00C R D Joystick-mouse 1 data (right vert, horiz)

These addresses each read a pair of 8 bit mouse counters. 0=left controller pair, 1=right controller pair. (4 counters total). The bit usage for both left and right addresses is shown below. Each counter is clocked by signals

from 2 controller pins. Bits 1 and 0 of each counter may be read to determine the state of these 2 clock pins. This allows these pins to double as joystick switch inputs.

Mouse counter usage (pins 1,3 = Yclock, pins 2,4 = Xclock)

```
BIT#  15,14,13,12,11,10,09,08  07,06,05,04,03,02,01,00
ODAT  Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0  X7 X6 X5 X4 X3 X2 X1 X0
1DAT  Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0  X7 X6 X5 X4 X3 X2 X1 X0
```

The following table shows the Mouse/Joystick connector pin usage. The pins (and their functions) are sampled (multiplexed) into the DENISE chip during the clock times shown in the table. This table is for reference only, and should not be needed by the programmer. (Note that the joystick functions are all "active low" at the conn. pins.)

Conn Pin	Joystick Function	Mouse Function	Sampled by DENISE Pin	Name	Clock
L1	FORW*	Y	38	MOV	at CCK
L3	LEFT*	YQ	38	MOV	at CCK*
L2	BACK*	X	9	MOH	at CCK
L4	RIGH*	XQ	9	MOH	at CCK*
R1	FORW*	Y	39	M1V	at CCK
R3	LEFT*	YQ	39	M1V	at CCK*
R2	BACK*	X	8	M1H	at CCK
R4	RIGH*	XQ	8	M1H	at CCK*

After being sampled, these Connector Pin signals are used in quadrature to clock the Mouse Counters. The LEFT and RIGHT joystick functions (active high) are directly available on the Y1 and X1 bits of each counter. In order to recreate the FORWARD and BACK joystick functions; however, it is necessary to logically combine (exclusive OR) the lower two bits of each counter. This is illustrated in the following table.

To detect	read these counter bits
Forward	Y1 xor Y0 (BIT#09 xor BIT#08)
Left	Y1
Back	X1 xor X0 (BIT#01 xor BIT#00)
Right	X1

JOYTEST 036 W D Write to all 4 Joystick-mouse counters at once.

Mouse counter write test data

```
BIT#  15,14,13,12,11,10,09,08  07,06,05,04,03,02,01,00
ODAT  Y7 Y6 Y5 Y4 Y3 Y2 xx xx  X7 X6 X5 X4 X3 X2 xx xx
1DAT  Y7 Y6 Y5 Y4 Y3 Y2 xx xx  X7 X6 X5 X4 X3 X2 xx xx
```


POTODAT 012 R P Pot counter data left pair (vert,horiz)
POT1DAT 014 R P Pot counter data right pair (vert,horiz)

These addresses each read a pair of 8 bit pot counters. (4 counters total). The bit assignment for both addresses is shown below. The counters are stopped by signals from 2 controller connectors(left-right) with 2 pins each.

```
BIT#  15,14,13,12,11,10,09,08  07,06,05,04,03,02,01,00
----- ----------------------- -----------------------
RIGHT Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0  X7 X6 X5 X4 X3 X2 X1 X0
LEFT  Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0  X7 X6 X5 X4 X3 X2 X1 X0
```

CONNECTORS loc.	dir.	sym	pin	PAULA pin#	pin name
RIGHT	Y	RY	9	36	(POT1Y)
RIGHT	X	RX	5	35	(POT1X)
LEFT	Y	LY	9	33	(POT0Y)
LEFT	X	LX	5	32	(POT0X)

POTGO 034 W P Pot Port (4 bit) Direction and Data, and Pot Counter start.

POTINP 016 R P Pot pin data read

This register controls a 4 bit bi-directional I/O port that shares the same 4 pins as the 4 pot counters above.

BIT#	FUNCT	DESCRIPTION
15	OUTRY	Output enable for Paula pin 36
14	DATRY	I/O data Paula pin 36
13	OUTRX	Output enable for Paula pin 35
12	DATRX	I/O data Paula pin 35
11	OUTLY	Output enable for Paula pin 33
10	DATLY	I/O data Paula pin 33
09	OUTLX	Output enable for Paula pin 32
08	DATLX	I/O data Paula pin 32
07-01	0	Reserved for chip ID code (presently 0)
00	START	Start pots (dump capacitors, start counters)

REFPTR 028 W A Refresh pointer

This register is used as a Dynamic RAM refresh address generator. It is writeable for test purposes only, and should never be written by the microprocessor.

SERDAT 030 W P Serial Port Data and stop bits write (Transmit data buffer)

This address writes data to a Transmit data buffer. Data from this buffer is moved into a serial shift register for output transmission, whenever it is empty. This sets the Interrupt Request TBE (transmit buffer empty). A stop bit must be provided as part of the data word. The length of the data word is set by the position of the stop bit.

```
BIT#  15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
USE    0  0  0  0  0  0  S D8 D7 D6 D5 D4 D3 D2 D1 D0
```

note: S= stop bit =1 , D= data bits

SERDATR 018 R P Serial Port Data and Status read

(Receive data buffer)
This address reads data from a Receive data buffer.
Data in this buffer is loaded from a receiving shift register whenever it is full. Several interrupt request bits are also read at this address, along with the data, as shown below.

BIT#		
15	OVRUN	Serial port receiver overrun Reset by resetting Bit 11 of INTREQ
14	RBF	Serial port Receive Buffer Full (mirror)
13	TBE	Serial port Transmit Buffer Empty (mirror)
12	TSRE	Serial port Transmit shift reg. empty Reset by loading into buffer.
11	RXD	RXD pin receives UART serial data for direct bit test by the micro
10	0	Not used
09	STP	Stop bit
08	STP-DB8	Stop bit if LONG, Data bit if not.
07	DB7	Data bit
06	DB6	Data bit
05	DB5	Data bit
04	DB4	Data bit
03	DB3	Data bit
02	DB2	Data bit
01	DB1	Data bit
00	DB0	Data bit

SERPER 032 W P Serial Port Period and control
This register contains the control bit LONG referred to above, and a 15 bit number defining the serial port Baud Rate. If this number is N, then the Baud Rate is 1 bit every (N+1)*.2794 Microseconds.

BIT#		
15	LONG	Defines Serial Receive as 9 bit word.
14-00	RATE	Defines Baud Rate=1/((N+1)*.2794 microsec.)

SPRxPTH 120 W A Sprite x pointer (High 3 bits)
SPRxPTL 122 W A Sprite x pointer (Low 15 bits)
This pair of registers contains the 18 bit address of Sprite x (x=0,1,2,3,4,5,6,7) DMA data. These address registers must be initialized by the processor or Copper every vertical blank time.

SPRxPOS 140 W A D Sprite x Vert-Horiz start position data
SPRxCTL 142 W A D Sprite x Vert stop position and control data
These 2 registers work together as position, size and feature Sprite control registers. They are usually loaded by the Sprite DMA channel, during horizontal blank, however they may be loaded by either processor any time.
SPRxPOS register:

BIT#	SYM	FUNCTION
15-08	SV7-SV0	Start vertical value. High bit(SV8) is in SPRxCTL reg below.
07-00	SH8-SH1	Start horizontal value. Low bit(SH0) is in SPRxCTL reg. below.

SPRxCTL register (writing this address dissables sprite horizontal comparator circuit):

BIT#	SYM	FUNCTION
15-08	EV7-EV0	End (stop) vert. value. low 8 bits
07	ATT	Sprite attach control bit(odd sprites)
06-04	X	Not used
02	SV8	Start vert. value high bit
01	EV8	End (stop) vert. value high bit
00	SH0	Start horiz. value Low bit

SPRxDATA 144 W D Sprite x image data register A
SPRxDATB 146 W D Sprite x image data register B
These registers buffer the Sprite image data. They are usually loaded by the Sprite DMA channel but may be loaded by either processor at any time. When a horizontal comparison occurs the buffers are dumped into shift registers and serially outputted to the display, MSB first on the left.
NOTE: Writing to the A buffer enables (arms) the sprite. Writing to the SPRxCTL register dissables the sprite.
If enabled, data in the A and B buffers will be outputted whenever the beam counter equals the sprite horizontal position value in the SPRxPOS register.

STREQU 038 S D Strobe for horiz sync with VB and EQU
STRVBL 03A S D Strobe for horiz sync with VB (Vert. Blank)
STRHOR 03C S D P Strobe for horiz sync
STRLONG 03E S D Strobe for identification of long horiz. line.
One of the first 3 strobe addresses above is placed on the dest.addr.bus during the first refresh time slot.
The 4th strobe shown above is used during the second refresh time slot of every other line, to identify lines with long counts(228). There are 4 refresh time slots, and any not used for strobes will leave a null (FF) address on the dest.addr.bus.

VPOSR 004 R A Read Vert most sig. bit (and frame flop)
VPOSW 02A W A Write Vert most sig. bit (and frame flop)

BIT#	15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
USE	LOF-- -- -- -- -- -- --,-- -- -- -- -- -- -- V8

LOF=Long frame (auto toggle control bit in BPLCON0)

VHPOSR 006 R A Read Vert and Horiz Position of beam or lightpen
VHPOSW 02C W A Write Vert and Horiz Position of beam or lightpen

BIT#	15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
USE	V7 V6 V5 V4 V3 V2 V1 V0,H8 H7 H6 H5 H4 H3 H2 H1

RESOLUTION=1/160 OF SCREEN WIDTH (280 NS)

APPENDIX B - Amiga Hardware Manual

This appendix contains information about the register set, in address order.

&=Register used by DMA channel only.
%=Register used by DMA channel usually, processors sometimes.
+=Address register pair. Low word uses DB1-DB15,High word DB0-DB2.
*=Address not writable by the Coprocessor.
~=Address not writable by the Coprocessor unless COPCON is set true.
A=Agnus chip, D=Denise chip, P=Paula chip
W=Write, R=Read,
ER=Early read. This is a DMA data transfer to RAM, from either the Disk or from the Blitter. Ram timing requires data to be on the bus earlier than microprocessor read cycles. These transfers are therefore initiated by Agnus timing, rather than a read address on the dest.adr.bus.
S=Strobe (write address with no register bits)
PTL,PTH=18 bit Pointer that addresses DMA data. Must be reloaded by a processor before use (Vertical blank for Bit Plane and Sprite pointers, and prior to starting the Blitter for Blitter pointers).
LCL,LCH=18 bit Location (starting address) of DMA data. Used to automatically restart pointers, such as the Coprocessor program counter (during vertical blank), and the Audio sample counter (whenever the audio length count is finished).
MOD=15 bit Modulo. A number that is automatically added to the memory address at the end of each line to generate the address for the beginning of the next line. This allows the Blitter (or the Display Window) to operate on (or display) a window of data that is smaller than the actual picture in memory. (memory map) Uses 15 bits, plus sign extend.

NAME		ADD	R/W	CHIP		FUNCTION
BLTDDAT	&	*000	ER	A		Blitter dest. early read (dummy address)
DMACONR		*002	R	A	P	DMA control (and blitter status) read
VPOSR		*004	R	A		Read Vert most sig. bit (and frame flop)
VHPOSR		*006	R	A		Read Vert and horiz Position of beam
DSKDATR	&	*008	ER		P	Disk data early read (dummy address)
JOYODAT		*00A	R	D		Joystick-mouse 0 data (vert,horiz)
JOY1DAT		*00C	R	D		Joystick-mouse 1 data (vert,horiz)
CLXDAT		*00E	R	D		Collision data reg.(Read and clear)
ADKCONR		*010	R		P	Audio, disk control register read
POTODAT		*012	R		P	Pot counter pair 0 data (vert,horiz)
POT1DAT		*014	R		P	Pot counter pair 1 data (vert,horiz)
POTINP		*016	R		P	Pot pin data read
SERDATR		*018	R		P	Serial Port Data and Status read
DSKBYTR		*01A	R		P	Disk Data byte and status read
INTENAR		*01C	R		P	Interrupt Enable bits Read
INTREQR		*01E	R		P	Interrupt Request bits read
DSKPTH	+	*020	W	A		Disk pointer (High 3 bits)
DSKPTL	+	*022	W	A		Disk pointer (Low 15 bits)
DSKLEN		*024	W		P	Disk length
DSKDAT	&	*026	W		P	Disk DMA Data write
REFPTR	&	*028	W	A		Refresh pointer
VPOSW		*02A	W	A		Write Vert most sig. bit (and frame flop)
VHPOSW		*02C	W	A		Write Vert and horiz Position of beam

COPCON		*02E	W	A		Coprocessor control register (CDANG)
SERDAT		*030	W		P	Serial Port Data and stop bits write
SERPER		*032	W		P	Serial Port Period and control
POTGO		*034	W		P	Pot count start,pot pin drive enable and data
JOYTEST		*036	W	D		Write to all 4 Joystick-mouse counters at once.
STREQU	&	*038	S	D		Strobe for horiz sync with VB and EQU
STRVBL	&	*03A	S	D		Strobe for horiz sync with VB (Vert. Blank)
STRHOR	&	*03C	S	D	P	Strobe for horiz sync
STRLONG	&	*03E	S	D		Strobe for identification of long horz. line.
BLTCONO		~040	W	A		Blitter control register 0
BLTCON1		~042	W	A		Blitter control register 1
BLTAFWM		~044	W	A		Blitter first word mask for source A
BLTALWM		~046	W	A		Blitter last word mask for source A
BLTCPTH	+	~048	W	A		Blitter Pointer to source C (High 3 bits)
BLTCPTL	+	~04A	W	A		Blitter Pointer to source C (Low 15 bits)
BLTBPTH	+	~04C	W	A		Blitter pointer to source B (High 3 bits)
BLTBPTL	+	~04E	W	A		Blitter pointer to source B (Low 15 bits)
BLTAPTH	+	~050	W	A		Blitter Pointer to source A (High 3 bits)
BLTAPTL	+	~052	W	A		Blitter Pointer to source A (Low 15 bits)
BLTDPTH	+	~054	W	A		Blitter Pointer to destn. D (High 3 bits)
BLTDPTL	+	~056	W	A		Blitter Pointer to destn. D (Low 15 bits)
BLTSIZE		~058	W	A		Blitter start and size (window width, height)
		~05A				
		~05C				
		~05E				
BLTCMOD		~060	W	A		Blitter Modulo for source C
BLTBMOD		~062	W	A		Blitter Modulo for source B
BLTAMOD		~064	W	A		Blitter Modulo for source A
BLTDMOD		~066	W	A		Blitter Modulo for destn. D
		~068				
		~06A				
		~06C				
		~06E				
BLTCDAT	&	~070	W	A		Blitter source C data reg
BLTBDAT	&	~072	W	A		Blitter source B data reg
BLTADAT	&	~074	W	A		Blitter source A data reg
		~076				
		~078				
		~07A				
		~07C				
DSKSYNC		~07E	R		P	Disk sync pattern register for disk read.
COP1LCH	+	080	W	A		Coprocessor first location reg (High 3 bits)
COP1LCL	+	082	W	A		Coprocessor first location reg. (Low 15 bits)
COP2LCH	+	084	W	A		Coprocessor second location reg. (High 3 bits)
COP2LCL	+	086	W	A		Coprocessor second location reg(Low 15 bits)
COPJMP1		088	S	A		Coprocessor restart at first location
COPJMP2		08A	S	A		Coprocessor restart at second location
COPINS		08C	W	A		Coprocessor inst. fetch identify
DIWSTRT		08E	W	A		Display Window Start (upper left vert-hor pos)
DIWSTOP		090	W	A		Display Window Stop (lower right vert-hor pos)
DDFSTRT		092	W	A		Display bit plane data fetch start(hor pos)
DDFSTOP		094	W	A		Display bit plane data fetch stop(hor pos)
DMACON		096	W	A	D P	DMA control write(clear or set)
CLXCON		098	W	D		Collision control
INTENA		09A	W		P	Interrupt Enable bits (clear or set bits)
INTREQ		09C	W		P	Interrupt Request bits (clear or set bits)

ADKCON		09E	W		P		Audio, Disk, UART Control
AUDOLCH	+	0A0	W	A			Audio channel 0 location (High 3 bits)
AUDOLCL	+	0A2	W	A			Audio channel 0 location (Low 15 bits)
AUDOLEN		0A4	W		P		Audio Channel 0 length
AUDOPER		0A6	W		P		Audio channel 0 Period
AUDOVOL		0A8	W		P		Audio Channel 0 Volume
AUDODAT	&	0AA	W		P		Audio channel 0 Data
		0AC					
		0AE					
AUD1LCH	+	0B0	W	A			Audio channel 1 location (High 3 bits)
AUD1LCL	+	0B2	W	A			Audio channel 1 location (Low 15 bits)
AUD1LEN		0B4	W		P		Audio Channel 1 length
AUD1PER		0B6	W		P		Audio channel 1 Period
AUD1VOL		0B8	W		P		Audio Channel 1 Volume
AUD1DAT	&	0BA	W		P		Audio channel 1 Data
		0BC					
		0BE					
AUD2LCH	+	0C0	W	A			Audio channel 2 location (High 3 bits)
AUD2LCL	+	0C2	W	A			Audio channel 2 location (Low 15 bits)
AUD2LEN		0C4	W		P		Audio Channel 2 length
AUD2PER		0C6	W		P		Audio channel 2 Period
AUD2VOL		0C8	W		P		Audio Channel 2 Volume
AUD2DAT	&	0CA	W		P		Audio channel 2 Data
		0CC					
		0CE					
AUD3LCH	+	0D0	W	A			Audio channel 3 location (High 3 bits)
AUD3LCL	+	0D2	W	A			Audio channel 3 location (Low 15 bits)
AUD3LEN		0D4	W		P		Audio Channel 3 length
AUD3PER		0D6	W		P		Audio channel 3 Period
AUD3VOL		0D8	W		P		Audio Channel 3 Volume
AUD3DAT	&	0DA	W		P		Audio channel 3 Data
		0DC					
		0DE					
BPL1PTH	+	0E0	W	A			Bit plane 1 pointer (High 3 bits)
BPL1PTL	+	0E2	W	A			Bit plane 1 pointer (Low 15 bits)
BPL2PTH	+	0E4	W	A			Bit plane 2 pointer (High 3 bits)
BPL2PTL	+	0E6	W	A			Bit plane 2 pointer (Low 15 bits)
BPL3PTH	+	0E8	W	A			Bit plane 3 pointer (High 3 bits)
BPL3PTL	+	0EA	W	A			Bit plane 3 pointer (Low 15 bits)
BPL4PTH	+	0EC	W	A			Bit plane 4 pointer (High 3 bits)
BPL4PTL	+	0EE	W	A			Bit plane 4 pointer (Low 15 bits)
BPL5PTH	+	0F0	W	A			Bit plane 5 pointer (High 3 bits)
BPL5PTL	+	0F2	W	A			Bit plane 5 pointer (Low 15 bits)
BPL6PTH	+	0F4	W	A			Bit plane 6 pointer (High 3 bits)
BPL6PTL	+	0F6	W	A			Bit plane 6 pointer (Low 15 bits)
		0F8					
		0FA					
		0FC					
		0FE					
BPLCONO		100	W	A	D		Bit plane control register(misc control bits)
BPLCON1		102	W		D		Bit plane control reg (scroll value PF1, PF2)
BPLCON2		104	W		D		Bit plane control reg (priority control)
		106					
BPL1MOD		108	W	A			Bit plane modulo (odd planes)
BPL2MOD		10A	W	A			Bit Plane modulo (even planes)
		10C					

		10E					
BPL1DAT	&	110	W		D		Bit plane 1 data (Parallel to serial convert)
BPL2DAT	&	112	W		D		Bit plane 2 data (Parallel to serial convert)
BPL3DAT	&	114	W		D		Bit plane 3 data (Parallel to serial convert)
BPL4DAT	&	116	W		D		Bit plane 4 data (Parallel to serial convert)
BPL5DAT	&	118	W		D		Bit plane 5 data (Parallel to serial convert)
BPL6DAT	&	11A	W		D		Bit plane 6 data (Parallel to serial convert)
		11C					
		11E					
SPR0PTH	+	120	W	A			Sprite 0 pointer (High 3 bits)
SPR0PTL	+	122	W	A			Sprite 0 pointer (Low 15 bits)
SPR1PTH	+	124	W	A			Sprite 1 pointer (High 3 bits)
SPR1PTL	+	126	W	A		L	Sprite 1 pointer (Low 15 bits)
SPR2PTH	+	128	W	A			Sprite 2 pointer (High 3 bits)
SPR2PTL	+	12A	W	A			Sprite 2 pointer (Low 15 bits)
SPR3PTH	+	12C	W	A			Sprite 3 pointer (High 3 bits)
SPR3PTL	+	12E	W	A			Sprite 3 pointer (Low 15 bits)
SPR4PTH	+	130	W	A			Sprite 4 pointer (High 3 bits)
SPR4PTL	+	132	W	A			Sprite 4 pointer (Low 15 bits)
SPR5PTH	+	134	W	A			Sprite 5 pointer (High 3 bits)
SPR5PTL	+	136	W	A			Sprite 5 pointer (Low 15 bits)
SPR6PTH	+	138	W	A			Sprite 6 pointer (High 3 bits)
SPR6PTL	+	13A	W	A			Sprite 6 pointer (Low 15 bits)
SPR7PTH	+	13C	W	A			Sprite 7 pointer (High 3 bits)
SPR7PTL	+	13E	W	A			Sprite 7 pointer (Low 15 bits)
SPR0POS	%	140	W	A	D		Sprite 0 Vert-Horiz start position data
SPR0CTL	%	142	W	A	D		Sprite 0 Vert stop position and control data
SPR0DATA	%	144	W		D		Sprite 0 image data register A
SPR0DATB	%	146	W		D		Sprite 0 image data register B
SPR1POS	%	148	W	A	D		Sprite 1 Vert-Horiz start position data
SPR1CTL	%	14A	W	A	D		Sprite 1 Vert stop position and control data
SPR1DATA	%	14C	W		D		Sprite 1 image data register A
SPR1DATB	%	14E	W		D		Sprite 1 image data register B
SPR2POS	%	150	W	A	D		Sprite 2 Vert-Horiz start position data
SPR2CTL	%	152	W	A	D		Sprite 2 Vert stop position and control data
SPR2DATA	%	154	W		D		Sprite 2 image data register A
SPR2DATB	%	156	W		D		Sprite 2 image data register B
SPR3POS	%	158	W	A	D		Sprite 3 Vert-Horiz start position data
SPR3CTL	%	15A	W	A	D		Sprite 3 Vert stop position and control data
SPR3DATA	%	15C	W		D		Sprite 3 image data register A
SPR3DATB	%	15E	W		D		Sprite 3 image data register B
SPR4POS	%	160	W	A	D		Sprite 4 Vert-Horiz start position data
SPR4CTL	%	162	W	A	D		Sprite 4 Vert stop position and control data
SPR4DATA	%	164	W		D		Sprite 4 image data register A
SPR4DATB	%	166	W		D		Sprite 4 image data register B
SPR5POS	%	168	W	A	D		Sprite 5 Vert-Horiz start position data
SPR5CTL	%	16A	W	A	D		Sprite 5 Vert stop position and control data
SPR5DATA	%	16C	W		D		Sprite 5 image data register A
SPR5DATB	%	16E	W		D		Sprite 5 image data register B
SPR6POS	%	170	W	A	D		Sprite 6 Vert-Horiz start position data
SPR6CTL	%	172	W	A	D		Sprite 6 Vert stop position and control data
SPR6DATA	%	174	W		D		Sprite 6 image data register A
SPR6DATB	%	176	W		D		Sprite 6 image data register B
SPR7POS	%	178	W	A	D		Sprite 7 Vert-Horiz start position data
SPR7CTL	%	17A	W	A	D		Sprite 7 Vert stop position and control data
SPR7DATA	%	17C	W		D		Sprite 7 image data register A

SPR7DATB	%	17E	W	D	Sprite 7 image data register B
COLOR00		180	W	D	Color table 00
COLOR01		182	W	D	Color table 01
COLOR02		184	W	D	Color table 02
COLOR03		186	W	D	Color table 03
COLOR04		188	W	D	Color table 04
COLOR05		18A	W	D	Color table 05
COLOR06		18C	W	D	Color table 06
COLOR07		18E	W	D	Color table 07
COLOR08		190	W	D	Color table 08
COLOR09		192	W	D	Color table 09
COLOR10		194	W	D	Color table 10
COLOR11		196	W	D	Color table 11
COLOR12		198	W	D	Color table 12
COLOR13		19A	W	D	Color table 13
COLOR14		19C	W	D	Color table 14
COLOR15		19E	W	D	Color table 15
COLOR16		1A0	W	D	Color table 16
COLOR17		1A2	W	D	Color table 17
COLOR18		1A4	W	D	Color table 18
COLOR19		1A6	W	D	Color table 19
COLOR20		1A8	W	D	Color table 20
COLOR21		1AA	W	D	Color table 21
COLOR22		1AC	W	D	Color table 22
COLOR23		1AE	W	D	Color table 23
COLOR24		1B0	W	D	Color table 24
COLOR25		1B2	W	D	Color table 25
COLOR26		1B4	W	D	Color table 26
COLOR27		1B6	W	D	Color table 27
COLOR28		1B8	W	D	Color table 28
COLOR29		1BA	W	D	Color table 29
COLOR30		1BC	W	D	Color table 30
COLOR31		1BE	W	D	Color table 31
RESERVED		1110X			
RESERVED		1111X			
NO-OP (NULL)		1FE			

APPENDIX C - Amiga Hardware Manual

CUSTOM IC PIN ALLOCATION LIST

NOTE: * Means active low signal.

AGNUS PIN ASSIGNMENT

PIN #	DESIGNATION	FUNCTION	DEFINITION
1	D8	DATA BUS 8	I/O
2	D7	DATA BUS 7	I/O
3	D6	DATA BUS 6	I/O
4	D5	DATA BUS 5	I/O
5	D4	DATA BUS 4	I/O
6	D3	DATA BUS 3	I/O
7	D2	DATA BUS 2	I/O
8	D1	DATA BUS 1	I/O
9	DO	DATA BUS O	I/O
10	VCC	+5 VOLT	I
11	RES*	SYSTEM RESET	I
12	INT3*	INTERRUPT LEVEL 3	O
13	DMAL	DMA REQUEST LINE	I
14	BLS*	BLITTER SLOWDOWN	I
15	DBR*	DATA BUS REQUEST	O
16	ARW*	AGNUS RAM WRITE	O
17	RGA8	REGISTER ADDRESS 8	I/O
18	RGA7	REGISTER ADDRESS 7	I/O
19	RGA6	REGISTER ADDRESS 6	I/O
20	RGA5	REGISTER ADDRESS 5	I/O
21	RGA4	REGISTER ADDRESS 4	I/O
22	RGA3	REGISTER ADDRESS 3	I/O
23	RGA2	REGISTER ADDRESS 2	I/O
24	RGA1	REGISTER ADDRESS 1	I/O
25	CCK	COLOR CLOCK	I
26	CCKQ	COLOR CLOCK DELAY	I
27	VSS	GROUND	I
28	DRAO	DYNAMIC RAM ADDRESS O	O
29	DRA1	DYNAMIC RAM ADDRESS 1	O
30	DRA2	DYNAMIC RAM ADDRESS 2	O
31	DRA3	DYNAMIC RAM ADDRESS 3	O
32	DRA4	DYNAMIC RAM ADDRESS 4	O
33	DRA5	DYNAMIC RAM ADDRESS 5	O
34	DRA6	DYNAMIC RAM ADDRESS 6	O
35	DRA7	DYNAMIC RAM ADDRESS 7	O
36	DRA8	DYNAMIC RAM ADDRESS 8	O
37	LP*	LIGHT PEN INPUT	I
38	VSY*	VERTICAL SYNC	I/O
39	CSY*	COMPOSITE SYNC	O
40	HSY*	HORIZONTAL SYNC	I/O
41	VSS	GROUND	I
42	D15	DATA BUS 15	I/O
43	D14	DATA BUS 14	I/O
44	D13	DATA BUS 13	I/O
45	D12	DATA BUS 12	I/O

46	D11	DATA BUS 11	I/O
47	D10	DATA BUS 10	I/O
48	D9	DATA BUS 9	I/O

DENISE PIN ASSIGNMENT

PIN #	DESIGNATION	FUNCTION	DEFINITION
1	D6	DATA BUS 6	I/O
2	D5	DATA BUS 5	I/O
3	D4	DATA BUS 4	I/O
4	D3	DATA BUS 3	I/O
5	D2	DATA BUS 2	I/O
6	D1	DATA BUS 1	I/O
7	DO	DATA BUS 0	I/O
8	M1H	MOUSE 1 HORIZONTAL	I
9	MOH	MOUSE 0 HORIZONTAL	I
10	RGA8	REGISTER ADDRESS 8	I
11	RGA7	REGISTER ADDRESS 7	I
12	RGA6	REGISTER ADDRESS 6	I
13	RGA5	REGISTER ADDRESS 5	I
14	RGA4	REGISTER ADDRESS 4	I
15	RGA3	REGISTER ADDRESS 3	I
16	RGA2	REGISTER ADDRESS 2	I
17	RGA1	REGISTER ADDRESS 1	I
18	BURST*	COLOR BURST	O
19	VCC	+5 VOLT	I
20	RO	VIDEO RED BIT 0	O
21	R1	VIDEO RED BIT 1	O
22	R2	VIDEO RED BIT 2	O
23	R3	VIDEO RED BIT 3	O
24	BO	VIDEO BLUE BIT 0	O
25	B1	VIDEO BLUE BIT 1	O
26	B2	VIDEO BLUE BIT 2	O
27	B3	VIDEO BLUE BIT 3	O
28	GO	VIDEO GREEN BIT 0	O
29	G1	VIDEO GREEN BIT 1	O
30	G2	VIDEO GREEN BIT 2	O
31	G3	VIDEO GREEN BIT 3	O
32	N/C	NOT CONNECTED	N/C

33	ZD*	BACKGROUND INDICATOR	O
34	N/C	NOT CONNECTED	N/C
35	7M	7.15909 MHZ	I
36	CCK	COLOR CLOCK	I
37	VSS	GROUND	I
38	MOV	MOUSE 0 VERTICAL	I
39	M1V	MOUSE 1 VERTICAL	I
40	D15	DATA BUS 15	I/O
41	D14	DATA BUS 14	I/O
42	D13	DATA BUS 13	I/O
43	D12	DATA BUS 12	I/O
44	D11	DATA BUS 11	I/O
45	D10	DATA BUS 10	I/O
46	D9	DATA BUS 9	I/O
47	D8	DATA BUS 8	I/O
48	D7	DATA BUS 7	I/O

PAULA PIN ASSIGNMENT

PIN #	DESIGNATION	FUNCTION	DEFINITION
1	D8	DATA BUS 8	I/O
2	D7	DATA BUS 7	I/O
3	D6	DATA BUS 6	I/O
4	D5	DATA BUS 5	I/O
5	D4	DATA BUS 4	I/O
6	D3	DATA BUS 3	I/O
7	D2	DATA BUS 2	I/O
8	VSS	GROUND	I
9	D1	DATA BUS 1	I/O
10	DO	DATA BUS O	I/O
11	RES*	SYSTEM RESET	I
12	DMAL	DMA REQUEST LINE	O
13	IPLO*	INTERRUPT LINE O	O
14	IPL1*	INTERRUPT LINE 1	O
15	IPL2*	INTERRUPT LINE 2	O
16	INT2*	INTERRUPT LEVEL 2	I
17	INT3*	NTERRUPT LEVEL 3	I
18	INT6*	INTERRUPT LEVEL 6	I
19	RGA8	REGISTER ADDRESS 8	I
20	RGA7	REGISTER ADDRESS 7	I
21	RGA6	REGISTER ADDRESS 6	I
22	RGA5	REGISTER ADDRESS 5	I
23	RGA4	REGISTER ADDRESS 4	I
24	RGA3	REGISTER ADDRESS 3	I
25	RGA2	REGISTER ADDRESS 2	I
26	RGA1	REGISTER ADDRESS 1	I
27	VCC	+5 VOLT	I
28	CCK	COLOR CLOCK	I
29	CCKQ	COLOR CLOCK DELAY	I
30	AUDB	RIGHT AUDIO	O
31	AUDA	LEFT AUDIO	O
32	POTOX	POT OX	I/O
33	POTOY	POT OY	I/O
34	VSSANA	ANALOG GROUND	I
35	POT1X	POT 1X	I/O
36	POT1Y	POT 1Y	I/O
37	DKRD*	DISK READ DATA	I
38	DKWD*	DISK WRITE DATA	O
39	DKWE	DISK WRITE ENABLE	O
40	TXD	SERIAL TRANSMIT DATA	O
41	RXD	SERIAL RECEIVE DATA	I
42	D15	DATA BUS 15	I/O
43	D14	DATA BUS 14	I/O
44	D13	DATA BUS 13	I/O
45	D12	DATA BUS 12	I/O
46	D11	DATA BUS 11	I/O
47	D10	DATA BUS 10	I/O
48	D9	DATA BUS 9	I/O

APPENDIX D - Amiga Hardware Manual

This appendix contains the system memory map.

SYSTEM MEMORY MAP

ADDRESS RANGE	NOTES
000000-03FFFF	256k bytes of RAM
040000-07FFFF	256k bytes of display RAM (option card)
080000-1FFFFF	Do not use.
200000-9FFFFF	External expansion space
A00000-BEFFFF	Do not use.
BFD000-BFDF00 = =	8520-B (access only at EVEN byte addresses)
BFE001-BFEF01 = =	8520-A (access only at ODD byte addresses)

The underlined digit chooses which of the 16 internal registers of the 8520 is to be accessed.

Register Names are given below.

C00000-DFEFFF	Reserved for future use.
DFF000-DFFFFF	Special purpose chips, where the last three digits specify the chip register WORD address. The chip addresses are specified in separate pages immediately following this overall memory map.
E00000-E7FFFF	Reserved for future use - do not use.
E80000-EFFFFF	Expansion slot decoding.
F00000-F7FFFF	Reserved - do not use.
F80000-FFFFFF	SYSTEM ROM

DEVELOPMENT SYSTEM ROMs located at start address FE0000.

FINAL SYSTEM ROMs will probably be located at FC0000.

The names of the registers within the 8520's are as follows. The address at which each is to be accessed is given here in this list.

Address for: 8520-A	8520-B	NAME	EXPLANATION (write)/(read mode)
BFE001	BFD000	PRA	Peripheral Data Register A
BFE101	BFD100	PRB	Peripheral Data Register B
BFE201	BFD200	DDRB	Data Direction Register "A"
BFE301	BFD300	DDRA	Data Direction Register "B"
BFE401	BFD400	TALO	TIMER A Low Register
BFE501	BFD500	TAHI	TIMER A High Register
BFE601	BFD600	TBLO	TIMER B Low Register
BFE701	BFD700	TBHI	TIMER B High Register
BFE801	BFD800		Event LSB
BFE901	BFD900		Event 8 - 15
BFEA01	BFDA00		Event MSB
BFEB01	BFDB00		No connect
BFEC01	BFDC00	SDR	Serial Data Register
BFED01	BFDD00	ICR	Interrupt Control Register
BFEE01	BFDE00	CRA	Control Register A
BFEF01	BFDF00	CRB	Control Register B

APPENDIX E - Amiga Hardware Manual

This appendix consists of three distinct parts, all related to the way in which the Amiga talks to the outside world.

The first part of this appendix specifies the pinouts of the externally accessible connectors and the power available at each connector. It does not, however, provide timing or loading information. Timing and loading information can be found in Appendix G.

The second part of this appendix contains a list of the connections for certain internal connectors, noteably the disk.

The third part of this appendix specifies how various signals relate to the available ports of the 8520. This information enables the programmer to relate the port addresses to the outside-world items (or internal control signals) which are to be affected. The second and third parts of the appendix are primarily for the use of the systems programmer and should generally not be utilized by applications programmers. Systems software normally is configured to handle the setting of particular signals, no matter how the physical connections may change. In other words, if you have a version of the system software which matches the rev. level of the machine (normally a true condition), when you ask that a particular bit be set, you don't care which port that bit is connected to. Thus applications programmers should rely on system documentation rather than going directly to the ports. Note also that in this, a multi-tasking operating system, many different tasks may be competing for the use of the system resources. Applications programmers should follow the established rules for resource access in order to assure compatiblity of their software with the system.

********** PART 1 - OUTSIDE WORLD CONNECTORS **********

This is a list of the Connections to the Outside World on the Amiga.

SERIAL COM ...DB25 FEMALE (J6) (the center column is the AMIGA connection, the others are specified in this table merely to show how the AMIGA "RS232" connection compares to other defined interconnect methods).

PIN	RS232	AMIGA	HAYES	DESCRIPTION
1	GND	GND		FRAME GROUND
2	TXD	TXD	TXD	TRANSMIT DATA
3	RXD	RXD	RXD	RECEIVE DATA
4	RTS	RTS		REQUEST TO SEND
5	CTS	CTS	CTS	CLEAR TO SEND
6	DSR	DSR	DSR	DATA SET READY
7	GND	GND	GND	SYSTEM GROUND
8	CD	CD	CD	CARRIER DETECT
9				
10				
11				
12	S.SD		SI	
13	S.CTS			
14	S.TXD	-5		- 5 VOLT POWER
15	TXC	AUDO		AUDIO OUT OF AMIGA
16	S.RXD	AUDI		AUDIO IN TO AMIGA
17	RXC	EB		BUFFERED PORT CLOCK 716kHz
18		INT2*		INTERRUPT LINE TO AMIGA
19	S.RTS			
20	DTR	DTR	DTR	DATA TERMINAL READY
21	SQD	+5		+ 5 VOLT POWER
22	RI		RI	
23	SS	+12		+12 volt power
24	TXC1	C2*		3.58 MHZ CLOCK
25		RESB*		BUFFERED SYSTEM RESET

PARALLEL COM ...DB25 MALE (J8)

1	DRDY*	14	GND
2	D0	15	GND
3	D1	16	GND
4	D2	17	GND
5	D3	18	GND
6	D4	19	GND
7	D5	20	GND
8	D6	21	GND
9	D7	22	GND
10	ACK*	23	+ 5
11	BUSY (data)	24	
12	POUT (clk)	25	RESET*
13	SEL		

KEYBOARD ...RJ11 (J9)

1	+5
2	CLOCK
3	DATA
4	GND

RGB ...DB23 MALE (J3)

1	XCLK*	13	GNDRTN (Return for XCLKEN*)
2	XCLKEN*	14	ZD*
3	RED	15	C1*
4	GREEN	16	GND
5	BLUE	17	GND
6	DI	18	GND
7	DB	19	GND
8	DG	20	GND
9	DR	21	-5 VOLT POWER
10	CSYNC*	22	+12 VOLT POWER
11	HSYNC*	23	+5 VOLT POWER
12	VSYNC*		

TV VIDEO ...8 PIN DIN (J2)

1	N.C.
2	GND
3	AUDIO LEFT
4	COMP VIDEO
5	GND
6	N.C.
7	+12 VOLT POWER
8	AUDIO RIGHT

DISK EXTERNAL ...DB23 FEMALE (J7)

1	RDY*	13	SIDEB*
2	DKRD*	14	WPRO*
3	GND	15	TKO*
4	GND	16	DKWEB*
5	GND	17	DKWDB*
6	GND	18	STEPB*
7	GND	19	DIRB
8	MTRXD*	20	SEL3B*
9	SEL2B*	21	SEL1B*
10	DRESB*	22	INDEX*
11	CHNG*	23	+12
12	+5		

RAMEX ...60 PIN EDGE (.156) (P1)

1	gnd	A	gnd
2	D15	B	D14
3	+5	C	+5
4	D12	D	D13
5	gnd	E	gnd
6	D11	F	D10
7	+5	H	+5
8	D8	J	D9
9	gnd	K	gnd
10	D7	L	D6
11	+5	M	+5
12	D4	N	D5
13	gnd	P	gnd
14	D3	R	D2
15	+5	S	+5
16	D0	T	D1
17	gnd	U	gnd
18	DRA4	V	DRA3
19	DRA5	W	DRA2
20	DRA6	X	DRA1
21	DRA7	Y	DRA0
22	gnd	Z	gnd
23	RAS*	AA	RRW*
24	gnd	BB	gnd
25	gnd	CC	gnd
26	CASU0*	DD	CASU1*
27	gnd	EE	gnd
28	CASL0*	FF	CASL1*
29	+5	HH	+5
30	+5	JJ	+5

EXPANSION ...86 PIN EDGE (.1) (P2)

1	gnd	44	IPL2*
2	gnd	45	A16
3	gnd	46	BERR*
4	gnd	47	A17
5	+5	48	VPA*
6	+5	49	gnd
7	exp	50	E
8	-5	51	VMA*
9	exp	52	A18
10	+12	53	RES*
11	exp	54	A19
12	CONFIG	55	HLT*
13	gnd	56	A20
14	C3*	57	A22
15	CDAC	58	A21
16	C1*	59	A23
17	OVR*	60	BR*
18	XRDY	61	gnd
19	INT2*	62	BGACK*
20	PALOPE*	63	PD15
21	A5	64	BG*
22	INT6*	65	PD14
23	A6	66	DTACK*
24	A4	67	PD13
25	gnd	68	PRW*
26	A3	69	PD12
27	A2	70	LDS*
28	A7	71	PD11
29	A1	72	UDS*
30	A8	73	gnd
31	FC0	74	AS*
32	A9	75	PD0
33	FC1	76	PD10
34	A10	77	PD1
35	FC2	78	PD9
36	A11	79	PD2
37	gnd	80	PD8
38	A12	81	PD3
39	A13	82	PD7
40	IPL0*	83	PD4
41	A14	84	PD6
42	IPL1*	85	gnd
43	A15	86	PD5

POWER ...7 PIN STRAIGHT (.156) (J14)

1	-5
2	+12
3	gnd
4	gnd
5	+5
6	+5
7	tick

JOY STICKS ...DB9 male (J11 = right J12 = left)

1	FORWARD*	(MOUSE V)
2	BACK*	(MOUSE H)
3	LEFT*	(MOUSE VQ)
4	RIGHT*	(MOUSE HQ)
5	POT X	(or button 3 ... if used)
6	FIRE*	(or button 1)
7	+5	
8	GND	
9	POT Y	(or button 2)

The following port power allocation list is based on many things, including known peripheral requirements and existing power supply capabilities. These numbers are maximums for each port when used independently, but the numbers can be accumulated (except for joysticks) when a particular system configuration will guarantee that it excusively uses more than one port.

The power pins of both joystick ports are tied together and to a current limited +5 supply. The current limit is currently set at 700 ma peak with a 400 ma foldback at steady state short circuit conditions. The combined utilization of both ports is limited to 250 ma to insure a minimum voltage drop at the pins.

PORT	+5(ma)	+12(ma)	-5(ma)
RF modulator	.	60	.
RGB	300	175	50
Serial	100	50	50
External disk	270	160	.
Parallel	100	.	.
Expansion	1000	50	50
Joystick 0	125	.	.
Joystick 1	125	.	.

********** PART 2 - INTERNAL CONNECTORS **********

DISK INTERNAL ...34 PIN RIBBON 'J10)

1	GND	18	DIRB
2	CHNG*	19	GND
3	GND	20	STEPB*
4	MTROD* (led)	21	GND
5	GND	22	DKWDB*
6	N.C.	23	GND
7	GND	24	DKWEB*
8	INDEX*	25	GND
9	GND	26	TKO*
10	SELOB*	27	GND
11	GND	28	WPRO*
12	N.C.	29	GND
13	GND	30	DKRD*
14	N.C.	31	GND
15	GND	32	SIDEB*
16	MTROD*	33	GND
17	GND	34	RDY*

DISK INTERNAL POWER ...4 PIN STRAIGHT (J13)

1	+12
2	GND
3	GND
4	+5

********** PART 3 - PORT SIGNAL ASSIGNMENTS FOR 8520 **********

Address BFFRO1 data bits 7-0 (A12*) (int2)

```
PA7..game port 1, pin 6 (fire button*)
PA6..game port 0, pin 6 (fire button*)
PA5..RDY*       disk ready*
PA4..TKO*       disk track 00*
PA3..WPRO*      write protect*
PA2..CHNG*      disk change*
PA1..LED*       led light (0=bright)
PA0..OVL        memory overlay bit

SP...KDAT       keyboard data
CNT..KCLK
```



```
PB7..P7         data 7
PB6..P6         data 6
PB5..P5         data 5     Centronics parallel interface
PB4..P4         data 4          data
PB3..P3         data 3
PB2..P2         data 2
PB1..P1         data 1
PB0..P0         data 0

PC...drdy*              centronics control
F....ack*
```

Address BFDRFE data bits 15-8 (A13*) (int6)

```
PA7..com line DTR*, driven output
PA6..com line RTS*, driven output
PA5..com line carrier detect*
PA4..com line CTS*
PA3..com line DSR*
PA2..SEL        centronics control
PA1..POUT       paper out ---+
PA0..BUSY       busy    ---+ |
                             | |
SP...BUSY       commodore -+ |
CNT..POUT       commodore ---+

PB7..MTR*       motor
PB6..SEL3*      select external 3rd drive
PB5..SEL2*      select external 2nd drive
PB4..SEL1*      select external 1st drive
PB3..SEL0*      select internal drive
PB2..SIDE*      side select*
PB1..DIR        direction
PB0..STEP*      step*

PC...not used
F....INDEX*     disk index*
```

APPENDIX F - Amiga Hardware Manual

This appendix contains information about the 8520 peripheral interface adapters.

.nf

INTERFACE SIGNALS

Clock Input

The O2 clock is a TTL compatible input used for internal device operation and as a timing reference for communicating with the system data bus.

CS - Chip Select Input

The CS input controls the activity of the 8520. A low level on CS while O2 is high causes the device to respond to signals on the R/W and address (RS) lines. A high on CS prevents these lines from controlling the 8520. The CS line is normally activated (low) at O2 by the appropriate address combination.

R/W - Read/Write Input

The R/W signal is normally supplied by the microprocessor and controls the direction of data transfers of the 8520. A high on R/W indicates a read (data transfer out of the 8520), while a low indicates a write (data transfer into the 8520).

RS3-RSO - Address Inputs

The address inputs select the internal registers as described by the Register Map.

DB7-DBO - Data Bus Inputs/Outputs

The eight data bus output pins transfer information between the 8520 and the system data bus. These pins are high impedance inputs unless CS is low and R/W and O2 are high, to read the device. During this read, the data bus output buffers are enabled, driving the data from the selected register onto the system data bus.

IRQ - Interrupt Request Output

IRQ is an open drain output normally connected to the processor interrupt input. An external pull-up resistor holds the signal high, allowing multiple IRQ outputs to be connected together. The IRQ output is normally off (high impedance) and is activated low as indicated in the functional description.

RES - Reset Input

A low on the RES pin resets all internal registers. The port pins are set as inputs and port registers to zero (although a read of the ports will return all highs because of passive pull-ups). the timer control registers are set to zero and the timer latches to all ones. All other registers are reset to zero.

REGISTER MAP

Each 8520 has 16 registers which you may read or write. Here is the list of registers and the access address of each within the memory space dedicated to the 8520:

RS3	RS2	RS1	RSO	Register #(hex)	NAME	MEANING
0	0	0	0	0	PRA	Peripheral Data Register A
0	0	0	1	1	PRB	Peripheral Data Register B
0	0	1	0	2	DDRA	Data Direction Register A
0	0	1	1	3	DDRB	Direction Register B
0	1	0	0	4	TALO	Timer A Low register
0	1	0	1	5	TAHI	Timer A High register
0	1	1	0	6	TBLO	Timer B Low register
0	1	1	1	7	TBHI	Timer B High register
1	0	0	0	8		Event LSB
1	0	0	1	9		Event 8-15
1	0	1	0	A		Event MSB
1	0	1	1	B		No Connect
1	1	0	0	C	SDR	Serial Data Register
1	1	0	1	D	ICR	Interrupt Control Register
1	1	1	0	E	CRA	Control Register A
1	1	1	1	F	CRB	Control Register B

SOFTWARE NOTE:

The operating system kernel has already allocated the use of all 4 of the timers TA and TB in the 8520's. If you are running under control of the system exec, be aware of the following allocation of system resources:

8520A, timer A -- Commodore serial communications (if no serial comm happening, timer becomes available).

8520A, timer B -- Video beam follower (used when synchronizing the blitter device to the video beam, see the description of QBSBlit in the system software manual). If no beam-sync'ed blits are in process, this timer will be available.

8520B, timer A -- Keyboard (used continuously, whenever system EXEC is in control).
8520B, timer B -- Virtual timer device (used continuously, whenever system EXEC is in control, used for task switching/interrupts).

REGISTER NAMES

The names of the registers within the 8520's are as follows. The address at which each is to be accessed is given here in this list.

Address for:

8520-A	8520-B	NAME	EXPLANATION (write)/(read mode)
BFE001	BFD000	PRA	Peripheral Data Register A
BFE101	BFD100	PRB	Peripheral Data Register B
BFE201	BFD200	DDRB	Data Direction Register "A"
BFE301	BFD300	DDRA	Data Direction Register "B"
BFE401	BFD400	TALO	TIMER A Low Register
BFE501	BFD500	TAHI	TIMER A High Register
BFE601	BFD600	TBLO	TIMER B Low Register
BFE701	BFD700	TBHI	TIMER B High Register
BFE801	BFD800		Event LSB
BFE901	BFD900		Event 8 - 15
BFEA01	BFDA00		Event MSB
BFEB01	BFDB00		No connect
BFEC01	BFDC00	SDR	Serial Data Register
BFED01	BFDD00	ICR	Interrupt Control Register
BFEE01	BFDE00	CRA	Control Register A
BFEF01	BFDF00	CRB	Control Register B

REGISTER FUNCTIONAL DESCRIPTION:

I/O PORTS (PRA, PRB, DDRA, DDRB)

Ports A and B each consist of an 8-bit Peripheral Data Register (PR) and an 8-bit data direction register (DDR). If a bit in the DDR is set to a 1, the corresponding bit position in the PR becomes an output. If a DDR bit is set to a 0, the corresponding PR bit is defined as an input.

When you READ a PR register, you read the actual current state of the I/O pins (PA0-PA7, PB0-PB7, regardless of whether you have set them to be inputs or outputs.

Ports A and B have passive pull-up devices as well as active pull-ups, providing both CMOS and TTL compatibility. Both ports have two TTL load drive capability.

In addition to their normal I/O operations, ports PB6 and PB7 also provide timer output functions.

HANDSHAKING

Handshaking occurs on data transfers using the PC output pin and the FLAG input pin PC will go low on the third cycle after a Port B access. This signal can be used to indicate "data ready" at PORT B or "data accepted" from PORT B Handshaking on 16-bit data transfers (using both ports A and B) is possible by always reading or writing PORT A first FLAG is a negative edge sensitive input which can be used for receiving the PC output from another 8520, or as a general purpose interrupt input. Any negative transition on FLAG will set the FLAG interrupt bit.

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
0	PRA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
1	PRB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
2	DDRA	DPA7	DPA6	DPA5	DPA4	DPA3	DPA2	DPA1	DPA0
3	DDRB	DPB7	DPB6	DPB5	DPB4	DPB3	DPB2	DPB1	DPB0

INTERVAL TIMERS (TIMER A, TIMER B)

Each interval timer consists of a 16-bit read-only Timer Counter and a 16-bit write-only Timer Latch. Data written to the timer is latched into the Timer Latch, while data read from the timer is the present contents of the Timer Counter.

The latch is also called a prescalar in that it represents the countdown value which must be counted before the timer reaches an underflow (no more counts) condition. This latch (prescalar) value is a divider of the input clocking frequency.

The timers can be used independently, or linked for extended operations. Various timer operating modes allow generation of long time delays, variable width pulses, pulse trains, and variable frequency waveforms. Utilizing the CNT input, the timers can count external pulses or measure frequency, pulse width, and delay times of external signals.

Each timer has an associated control register, providing independent control over each of the following functions:

START/STOP

A control bit allows the timer to be started or stopped by the microprocessor at any time.

PB On/Off

A control bit allows the timer output to appear on a PORT B output line (PB6 for timer A and PB7 for timer B). This function over-rides the DDRB control bit and forces the appropriate PB line to become an output.

Toggle/Pulse

A control bit selects the output applied to PORT B while the PB On/Off bit is ON. On every timer underflow, the output can either toggle or generate a single positive pulse of one cycle duration. The Toggle output is set high whenever the timer is started, and set low by RES.

One-Shot/Continuous

A control bit selects either timer mode. In one-shot mode, the timer will count down from the latched value to zero, generate an interrupt, reload the latched value, then stop. In continuous mode, the timer will count down from the latched value to zero, generate an interrupt, reload the latched value, and repeat the procedure continuously.

In one-shot mode, a write to Timer High (register 5 for Timer A, register 7 for Timer B) will transfer the timer latch to the counter and initiate counting regardless of the start bit.

Force Load

A strobe bit allows the timer latch to be loaded into the timer counter at any time, whether the timer is running or not.

INPUT MODES

Control bits allow selection of the clock used to decrement the timer. TIMER A can count 02 clock pulses or external pulses applied to the CNT pin. TIMER B can count 02 pulses, external CNT pulses, TIMER A underflow pulses, or TIMER A underflow pulses while the CNT pin is held high.

The timer latch is loaded into the timer on any timer underflow, on a force load, or following a write to the high byte of the pre-scalar while the timer is stopped. If the timer is running, a write to the high byte will load the timer latch, but not reload the counter.

BIT NAMES on READ-register

REG	NAME	D7	D6	D5	D4	D3	D2	D1	DO
4	TALO	TAL7	TAL6	TAL5	TAL4	TAL3	TAL2	TAL1	TALO
5	TAHI	TAH7	TAH6	TAH5	TAH4	TAH3	TAH2	TAH1	TAHO
6	TBLO	TBL7	TBL6	TBL5	TBL4	TBL3	TBL2	TBL1	TBLO
7	TBHI	TBH7	TBH6	TBH5	TBH4	TBH3	TBH2	TBH1	TBHO

BIT NAMES on WRITE-register

REG	NAME	D7	D6	D5	D4	D3	D2	D1	DO
4	TALO	PAL7	PAL6	PAL5	PAL4	PAL3	PAL2	PAL1	PALO
5	TAHI	PAH7	PAH6	PAH5	PAH4	PAH3	PAH2	PAH1	PAHO
6	TBLO	PBL7	PBL6	PBL5	PBL4	PBL3	PBL2	PBL1	PBLO
7	TBHI	PBH7	PBH6	PBH5	PBH4	PBH3	PBH2	PBH1	PBHO

TIME OF DAY CLOCK

TOD consists of a 24-bit binary counter. Positive edge transitions on this pin cause the binary counter to increment. The TOD pin has a passive pull-up on it.

A programmable ALARM is provided for generating an interrupt at a desired time. The ALARM registers are located at the same addresses as the corresponding TOD registers. Access to the ALARM is governed by a Control Register bit. The ALARM is write-only; any read of a TOD address will read time regardless of the state of the ALARM access bit.

A specific sequence of events must be followed for proper setting and reading of TOD. TOD is automatically stopped whenever a write to the register occurs. The clock will not start again until after a write to the LSB Event Register. This assures that TOD will always start at the desired time.

Since a carry from one stage to the next can occur at any time with respect to a read operation, a latching function is included to keep all Time Of Day information constant during a read sequence. All TOD registers latch on a read of MSB Event and remain latched until after a read of LSB Event. The TOD clock continues to count when the output registers are latched. If only one register is to be read, there is no carry problem and the register can be read "on the fly" provided that any read of MSB Event is followed by a read of LSB Event to disable the latching.

BIT NAMES for WRITE TIME/ALARM or READ TIME

REG	NAME								
8	LSB Event	E7	E6	E5	E4	E3	E2	E1	E0
9	Event 8-15	E15	E14	E13	E12	E11	E10	E9	E8
A	MSB Event	E23	E22	E21	E20	E19	E18	E17	E16

WRITE
CRB7 = 0
CRB7 = 1 ALARM

SERIAL PORT (SDR)

The serial port is a buffered, 8-bit synchronous shift register. A control bit selects input or output mode.

INPUT MODE

In input mode, data on the SP pin is shifted into the shift register on the rising edge of the signal applied to the CNT pin. After 8 CNT pulses, the data in the shift register is dumped into the Serial Data Register and an interrupt is generated.

OUTPUT MODE

In the output mode, TIMER A is used as the baud rate generator. Data is shifted out on the SP pin at 1/2 the underflow rate of TIMER A. The maximum baud rate possible is 02 divided by 4, but the maximum usable baud rate will be determined by line loading and the speed at which the receiver responds to input data.

To begin transmission, you must first set up TIMER A in continuous mode, and start the timer. Transmission will start following a write to the Serial Data Register. The clock signal derived from TIMER A appears as an output on the CNT pin. The data in the Serial Data Register will be loaded into the shift register, then shifted out to the SP pin when a CNT pulse occurs. Data shifted out becomes valid on the next falling edge of CNT and remains valid until the next falling edge.

After 8 CNT pulses, an interrupt is generated to indicate that more data can be sent. If the Serial Data Register was reloaded with new information prior to this interrupt, the new data will automatically be loaded into the shift register and transmission will continue.

If no further data is to be transmitted after the 8th CNT pulse, CNT will return high and SP will remain at the level of the last data bit transmitted.

SDR data is shifted out MSB first. Serial input data should appear in this same format.

BIDIRECTIONAL FEATURE

The bidirectional capability of the Serial Port and CNT clock allows many 8520's to be connected to a common serial communications bus on which one 8520 acts as a master, sourcing data and shift clock, while all other 8520 chips act as slaves. Both CNT and SP outputs are open drain to allow such a common bus. Protocol for master/slave selection can be transmitted over the serial bus or via dedicated handshake lines.

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
C	SDR	S7	S6	S5	S4	S3	S2	S1	S0

INTERRUPT CONTROL REGISTER (ICR)

There are 5 sources of interrupts on the 8520:

-underflow from TIMER A (timer counts down past 0)
-underflow from TIMER B
-TOD ALARM
-Serial Port Full/Empty
-FLAG

A single register provides masking and interrupt information. The Interrupt Control Register consists of a write-only MASK register and a read-only DATA register. Any interrupt will set the corresponding bit in the DATA register. Any interrupt that is enabled by a 1-bit in that position in the MASK will set the IR bit (MSB) of the DATA register, and bring the IRQ pin low. In a multi-chip system, the IR bit can be polled to detect which chip has generated an interrupt request.

When you read the DATA register, its contents are cleared (set to 0) and the IRQ line returns to a high state. Since it is cleared on a read, you must assure that your interrupt polling or interrupt service code can preserve and respond to all bits which may have been set in the DATA register at the time it was read. With proper preservation and response, it is easily possible to intermix polled and direct interrupt service methods.

You can set or clear one or more bits of the MASK register without affecting the current state of any of the other bits in the register. This is done by setting the appropriate state of the MSBit, which is called the SET/CLEAR bit. In bits 6-0, you yourself form a mask which specifies which of the bits you wish to affect. Then, using bit 7, you specify HOW the bits in corresponding positions in the mask are to be affected.

If bit 7 is a 1, then any bit 6-0 in your own mask word which is set to a 1 SETS the corresponding bit in the MASK register. Any bit which you have set to a 0 causes the MASK register bit to remain in its current state.

If bit 7 is a 0, then any bit 6-0 in your own mask word which is set to a 1 CLEARS the corresponding bit in the MASK register. Again, any 0 bit in your own mask word causes no change in the contents of the corresponding MASK register bit.

If an interrupt is to occur based on a particular condition, then that corresponding MASK bit must be a 1.

Example: Suppose you want to SET the TIMER A interrupt bit (enable the TIMER A interrupt), but want to be sure that all other interrupts are CLEARED. Here is the sequence you can use:

```
movi.b  01111110B,A0
mov.b   A0,ICR      ;MSB is 0, means clear
                    ;any bit whose value is
                    ;1 in the rest of the byte
movi.b  10000001B,A0
mov.b   A0,ICR      ;MSB is 1, means set
                    ;any bit whose value is
                    ;1 in the rest of the byte
                    ;(do not change any values
                    ; wherein the written value
                    ; bit is a zero)
```

Read Interrupt Control Register:

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
D	ICR	IR	0	0	FLG	SP	ALRM	TB	TA

Write Interrupt Control MASK:

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
D	ICR	S/C	x	x	FLG	SP	ALRM	TB	TA

CONTROL REGISTERS

There are two control registers in the 8520, CRA and CRB. CRA is associated with TIMER A and CRB is associated with TIMER B. The format of the registers is as follows:

CONTROL REGISTER A:

BIT	NAME	FUNCTION
0	START	1 = start TIMER A, 0 = stop TIMER A. This bit is automatically reset (= 0) when underflow occurs during one-shot mode.
1	PBON	1 = TIMER A output on PB6, 0 = PB6 is normal operation.
2	OUTMODE	1 = TOGGLE, 0 = PULSE.
3	RUNMODE	1 = one-shot mode, 0 = continuous mode.
4	LOAD	1 = FORCE LOAD (this is a STROBE input, there is no data storage; bit 4 will always read back a zero and writing a 0 has no effect.)
5	INMODE	1 = TIMER A counts positive CNT transitions, 0 = TIMER A counts 02 pulses.
6	SPMODE	1 = SERIAL PORT=output (CNT is the source of the shift clock) 0 = SERIAL PORT=input (external shift clock is required)

BIT MAP OF REGISTER CRA:

REG #	NAME	TOD IN	SPMODE	INMODE	LOAD	RUNMODE	OUTMODE	PBON	START
E	CRA	0=60Hz 1=50Hz	0=input 1=output	0=02 1=CNT	1=FORCE LOAD (STROBE)	0=cont. 1=one-shot	0=pulse 1=toggle	0=PB60FF 1=PB60N	0=stop 1=start

|<-------- TIMER A Variables ---------->|

All unused register bits are unaffected by a write and forced to 0 on a read.

CONTROL REGISTER B:

BIT	NAME	FUNCTION
0	START	1 = start TIMER B, 0 = stop TIMER B. This bit is automatically reset (= 0) when underflow occurs during one-shot mode.
1	PBON	1 = TIMER B output on PB7, 0 = PB7 is normal operation.
2	OUTMODE	1 = TOGGLE, 0 = PULSE.
3	RUNMODE	1 = one-shot mode, 0 = continuous mode.
4	LOAD	1 = FORCE LOAD (this is a STROBE input, there is no data storage; bit 4 will always read back a zero and writing a 0 has no effect.)

6,5 INMODE Bits CRB6 and CRB5 select one of four possible input modes for TIMER B, as follows:

CRB6	CRB5	Mode Selected
0	0	TIMER B counts 02 pulses
0	1	TIMER B counts positive CNT transitions
1	0	TIMER B counts TIMER A underflow pulses
1	1	TIMER B counts TIMER A underflow pulses while CNT pin is held high.

7 ALARM 1 = writing to TOD registers sets ALARM,
0 = writing to TOD registers sets TOD clock.
Reading TOD registers always reads TOD clock, regardless of the state of the ALARM bit.

BIT MAP OF REGISTER CRB:

REG #	NAME	ALARM	INMODE	LOAD	RUNMODE	OUTMODE	PBON	START
F	CRB	0=TOD	00=02	1=FORCE	0=cont.	0=pulse	0=PB7OFF	0=stop
		1=ALARM	01=CNT	LOAD	1=one-	1=toggle	1=PB7ON	1=start
			10=TIMER A	(STROBE)	shot			
			11=CNT+TIMER A					

|<----------TIMER B Variables----------->|

All unused register bits are unaffected by a write and forced to 0 on a read.

PORT SIGNAL ASSIGNMENTS

This part specifies how various signals relate to the available ports of the 8520. This information enables the programmer to relate the port addresses to the outside-world items (or internal control signals) which are to be affected. This part is primarily for the use of the systems programmer and should generally not be utilized by applications programmers. Systems software normally is configured to handle the setting of particular signals, no matter how the physical connections may change. In other words, if you have a version of the system software that matches the rev. level of the machine (normally a true condition), when you ask that a particular bit be set, you don't care which port that bit is connected to. Thus applications programmers should rely on system documentation rather than going directly to the ports. Note also that in this, a multi-tasking operating system, many different tasks may be competing for the use of the system resources. Applications programmers should follow the established rules for resource access in order to assure compatibility of their software with the system.

Address BFERFF data bits 7-0 (A12*) (int2)

```
PA7..game port 1, pin 6 (fire button*)
PA6..game port 0, pin 6 (fire button*)
PA5..RDY*       disk ready*
PA4..TK0*       disk track 00*
PA3..WPRO*      write protect*
PA2..CHNG*      disk change*
PA1..LED*       led light (0=bright)
PA0..OVL        memory overlay bit

SP...KDAT       keyboard data
CNT..KCLK

PB7..P7         data 7
PB6..P6         data 6
PB5..P5         data 5     Centronics parallel interface
PB4..P4         data 4          data
PB3..P3         data 3
PB2..P2         data 2
PB1..P1         data 1
PB0..P0         data 0

PC...drdy*              centronics control
F....ack*
```

Address BFDRFE data bits 15-8 (A13*) (int6)

```
PA7..com line DTR*, driven output
PA6..com line RTS*, driven output
PA5..com line carrier detect*
PA4..com line CTS*
PA3..com line DSR*
PA2..SEL        centronics control
PA1..POUT       paper out ---+
PA0..BUSY       busy    ---+ |
                           | |
SP...BUSY       commodore -+ |
CNT..POUT       commodore ---+

PB7..MTR*       motor
PB6..SEL3*      select external 3rd drive
PB5..SEL2*      select external 2nd drive
PB4..SEL1*      select external 1st drive
PB3..SEL0*      select internal drive
PB2..SIDE*      side select*
PB1..DIR        direction
PB0..STEP*      step*

PC...not used
F....INDEX*     disk index*
.fi
```

APPENDIX G - Amiga Hardware Manual

This appendix describes the use of the expansion connector.

BOXES AND BOARDS

In this document boards and boxes are logical entities, that is if you want to design a physical board that looks to the processor as if it is a box, or as two boxes, or a box of boards, feel free. In general, we think people will design boxes as boxes, and boards as boards.

Boxes reside on the bus, and they may or may not contain boards. They always contain an Autoconfig Control Block so that our autoconfig software can come out and build a table of who's there, and where to put them in Config Space or Memory Space. Config space is a 512K Byte space starting at E80000; it is broken up into 128 Slot Spaces consisting of 4K Bytes each. Every autoconfigured box and board gets it's own slot space. This is assigned by the autoconfig software by writing into the Slot Address Register of each of the boxes. The box slot is at the location specified in the Slot Address Reg (Fig 1), and each board if there is any, is assigned to the next adjacent 4K slot. The software knows (from the ROM in the box Auto-config Control Block) how big each box is, so it knows where in Slot Space to locate the boxes.

If a box or board is small enough to live in 4K or less, then it does not need any Memory Space and therefore will not have an Optional Memory Address Register (Fig 1). Those boxes that require more than 4K Bytes can get a larger address space from Memory Space by providing a Memory Address Register at the appropriate location in Config Space.

The result of all this is that boxes can do the address recognition for small (under 4K) boards, while big boards need an 8 bit latch (Mem Addr Reg) to locate them in the 8 Meg Memory Space. Note that all big boards, whether they are memory boards or not, get located in Memory Space.

AUTO-CONFIGURATION

Auto configuration of resources on the expansion bus is accomplished by an interrogation of the boxes and boards that the user has installed on the bus. All boxes should be designed to respond to Config Space (A23-A19=11101) ANDed with CONFIG In (Pin 12) active and Config Out inactive. At the beginning of this address range starting at E80000 will be the Auto Configuration Control Block (see Figure 1). CONFIG* is passed from one resource (box or board) to the next; it is daisy chained through the boxes, with each box passing it on to the next after the former has been configured. Each box is designed to assert SLOTSELn*

(Pin 12 on board connectors) to it's boards by decoding a subset of A18-A12. For instance in the case of a box with seven board slots, the box would decode A14-A12, and fan out SLOTSEL1* through SLOTSEL7* to it's boards. SLOTSEL0* from the the same 3:1 decoder would activate slot space on the box itself. When the software wants to configure the next box, it sets the "Pass CONFIG" bit in the current boxes control register. This asserts CONFIG* to the next board.

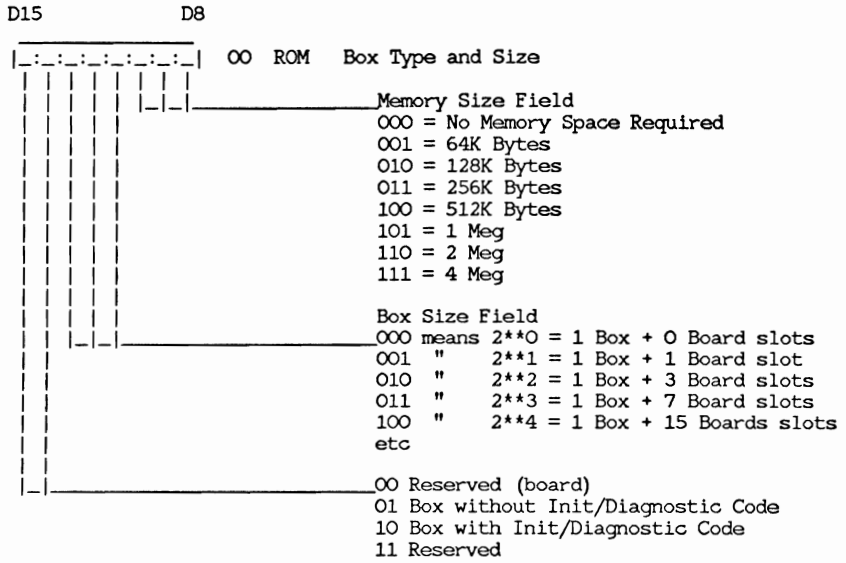
The Configuration Process

The Slot Address Registers are cleared to zero by System Reset, thus all boxes come up ready to respond to address E80nnn ANDed with CONFIG* being asserted to that box.

Because CONFIG* is daisy chained, only the currently selected box will respond to Config space E80nnn. All others have either been configured to respond to a different space by loading a non-zero number into the Slot Address Register, or CONFIG* has not gotten out to that box or board yet.

The configuration software reads the ROM portion of the Configuration Control Block in order to identify the box. It then places the box into some slot other than slot zero, and goes on to find the next box. Once all of the boxes and their sizes are known, the software can rearrange them in slot space to suit their various sizes.

Then the config software goes back and interrogates the boards, and maps large boxes and boards (eg more than 4K) into memory space. During the process of configuring the boxes and boards, a table of box and board identifiers and locations in slot and memory space is constructed, so that application software can later access the appropriate resources.



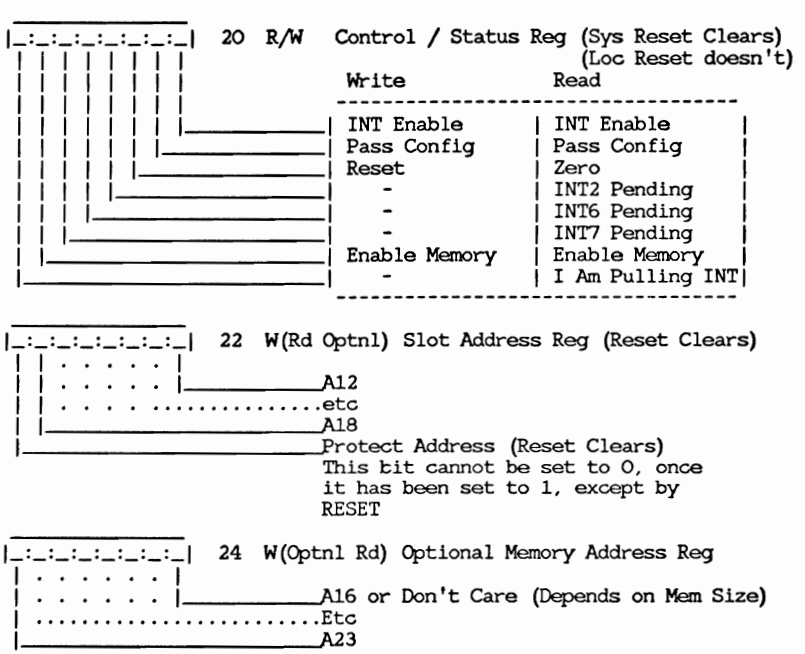


Figure 1 : Box Auto Configuration Control Block

Box Auto Configuration Control Block

This section discusses each field in the box auto-configuration control block. The block is made up of a combination of required and optional ROM bytes, and required and optional control registers. Obviously, individual board designs will have other control registers as well, for instance DMA control registers. The autoconfig block contains all of the information necessary for the CAI autoconfig software. There is a similar block for boards, called the Board Auto Configuration Control Block.

Box Type and Size

This ROM byte will identify the type and size of the box responding. Figure 1 explains the fields.

Product Number

This number is determined by the manufacturer and will be reported in the auto configuration system table.

Reserved 2 Bytes

These bytes must read zero.

Mfg Number (2 Bytes)

This number is assigned by Commodore, and will be reported in the auto configuration system table as well. Each manufacturer is assigned a unique number by Commodore.

Optional Serial Number (4 Bytes)

This number may be supplied or omitted by the manufacturer, whatever is found when reading this field will be reported in the auto config system table.

Optional Init/Diagnostic Vector

This 2 byte vector points to Init/Diagnostic Code. If such code is resident on the box, then the Box Type/Size field will tell the processor. The processor will use this vector to do a JSR to the code. The code should terminate with an RTS.

Reserved Byte

No response required.

Control/Status Register

This R/W register is required on all boxes. Some of the functions in this register may not apply to a particular design. In the case that none of these functions apply, the most significant bit must still be implemented as a R/W bit for auto config and diagnostic reasons.

If the box drives one or more interrupts to the processor, then the appropriate Enable and Pending bits must be implemented.

Pass CONFIG sends an active CONFIG* signal to the next box when Pass Config is set to a 1.

The Local Reset Pulse resets most control state on the box, just as a reset from the processor Reset line does. Note that this pulse does not reset the Control/Status Register.

The Enable Memory bit enables the optional memory space recognition circuitry if the box uses memory space.

Slot Address Register

This register is used by the auto config code to place the box in Slot Space (aka Config Space). This address also becomes the base address from which any boards in the box are offset (on 4K boundaries). In other words the box performs the slot address recognition for it's boards as well.

This register is required, it is cleared by Sys Reset only (not local). For diagnostic reasons and ease of software, you may want to make it R/W, only Write is required however.

Note the high order bit is Protect Address. This bit prevents the Slot Address Reg and the Memory Address Reg from being written, once this bit is set. This bit can only be cleared by System Reset.

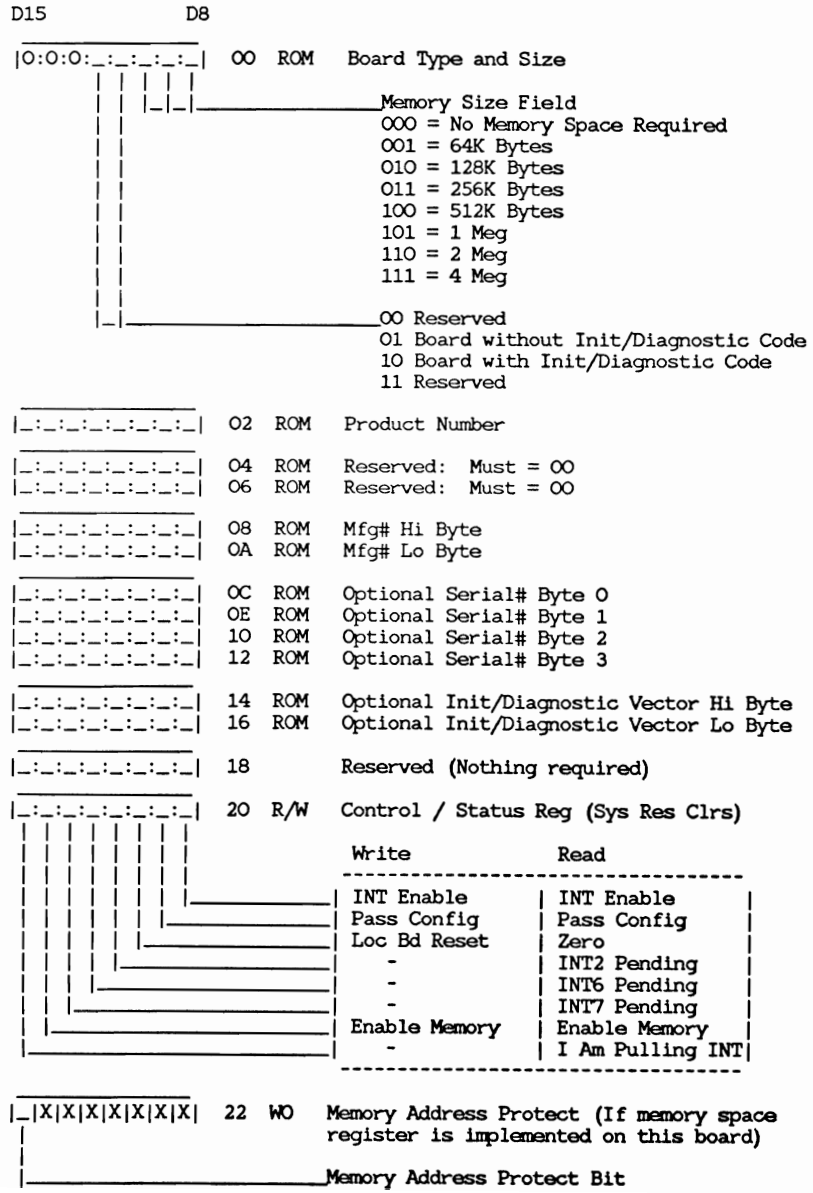
Optional Memory Address Register

This register is only required if the resources on the box itself need more than 4K Bytes of address space. If more space is required, put the appropriate size code in the Memory Size Field (Fig 1) of the Box Auto Configuration Control Block. This register will then be loaded with a base address for the required Memory Address Space. Loading this register is the logical equivalent of setting switches on a manually configured board. I used to have a great joke in this document right here, but Glenn made me take it out.

Once the Address Protect Bit has been set, this register can no longer be written.

Board Auto Configuration Control Block

There is a second version of the Autoconfig Control Block for boards. It uses the same principles as the box level does, but it is a subset because the boards need less information.



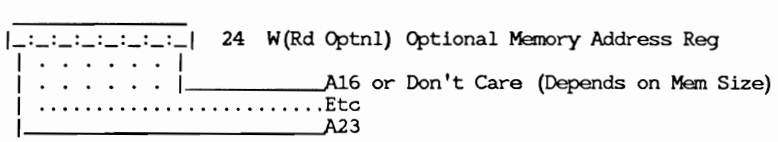


Figure 2 : Board Auto Configuration Control Block

Bad Configuration Protection

Because bad software could cause two bus slaves to respond to the same address (thus causing 3 state fights), it is necessary to put a protection circuit on each box. This circuit detects if two slaves are responding on the same bus cycle, and if so it asserts BERR* so that the bus drivers cannot burn each other up. The data bus drivers on all slave boards should be designed to go tri-state if BERR* is active (low). The tri-state fight detection circuit should assert BERR* low as soon as two simultaneous responses are detected. It should hold BERR* low for greater than .25 seconds.

The I_RESPONDn* lines are usually a copy of the tri-state enable signal to the data drivers. Logically they mean that the board or box is responding to this address cycle. Be careful that if you are designing a bus master, that you do not assert I_RESPOND* while you are the active master.

Notice that a RESPONSE* signal cascades down from upstream boxes. Any two responders on one bus cycle is an error and will assert the BERR* one shot to the system.

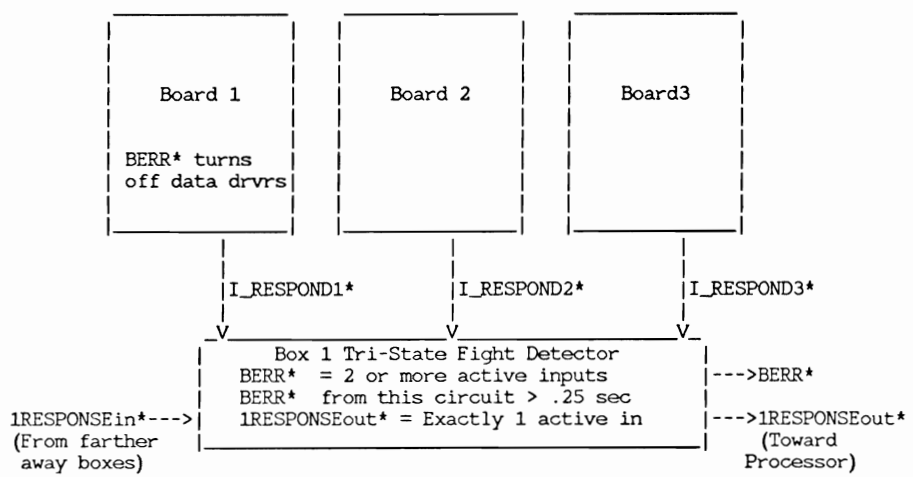


Fig. : Tri-State Fight Protection Scheme

DESIGNING IN BUS FAIRNESS

It is intended that bus masters for the Amiga will be designed in such a manner that the bus protocol is "fair". In a fair protocol, all requesters have an opportunity to utilize the bus, high priority masters do not starve low priority masters.

Monitoring Common Bus Request

BR* to the 68000 serves as a common bus request. The Amiga scheme for fairness uses a state machine (for each bus master) that employs the following rule. Once a master has completed a burst on the bus (burst size should be tunable by software) he relinquishes the bus if Common Bus Request is currently being asserted by another master. He does not assert BR* again until he sees BR* go inactive. The result of this is that if several masters are requesting the bus, they each get it, before any of them come back for seconds.

Note that there is one master who can get locked out in this scheme, the 68000 because it does not assert BR*. We have been thinking of designing a simple external circuit in an expansion box which would get the other masters off the bus, but at this time it is still speculation. You are welcome to implement the idea if you wish, it could probably be designed to greatly improve interrupt response time.

Both of the above ideas can be implemented very cheaply, and have big

payoffs in terms of system performance. There are subtleties for getting the best results, we would be glad to critique a specific implementation as time permits. If your scheme doesn't violate our plans, we would consider giving up 1 or 2 of the reserved pins on the 86 pin connector.

Board Form Factor

At this time we have not made a final decision on external expansion board form factors, other than the dimensions relating to the expansion connector itself. We are leaning toward a system of multiple sizes of boards, in order to accommodate a wide range of applications and markets. If you have strong preferences on this issue, we should discuss it soon. If we consider something with the area of an IBM PC biggest board as our unit board, then possibly one expansion box should be designed to house unit boards and 1/2 boards; and a second box designed to house 1/2 boards, unit boards, and double size boards. The doublesize boards would be the same length as IBM large size, and twice as high.

We are considering recommending a standard form factor expansion box so that small developers need not provide their own box. At this time we are not making a commitment to build or not build such a box ourselves. We are open to suggestions from third party developers. This standard form factor would house the size boards discussed in the above paragraph. The boards would include the 86 pin connector almost identical to the pin out on the processor, the following pins are worth noting:

Pin#	Name on Board	(Name on Box)	Description
7	EXP1	(EXP1)	Reserved
9	EXP2	(EXP2)	Reserved
11	ME*	(MEIN*,MEOUT*)	This signal indicates response to present Bus cycle.
12	BOARDSEL*	(CONFIG*)	Board select is done by box and passed to board via pin 12. This does not include select in Memory Space, which must be done by the board itself, if Memory Space is used.
20	EXP3	(EXP3)	Reserved (was CART*)

APPENDIX H - Amiga Hardware Manual

This appendix contains a description of the Amiga keyboard interface and the hardware of the Amiga keyboard.

KEYBOARD INTERFACE

The keyboard plugs into the computer via a 4-conductor cable similar to a telephone handset coily cord (in fact, a telephone handset cable may be substituted in a pinch). The four wires provide 5 volt power, ground, and two signals called KCLK (keyboard clock) and KDAT (keyboard data). KCLK is uni-directional and always driven by the keyboard; KDAT is driven by both the keyboard and the computer. Both signals are open-collector; there are pullup resistors in both the keyboard (inside the keyboard microprocessor) and the computer.

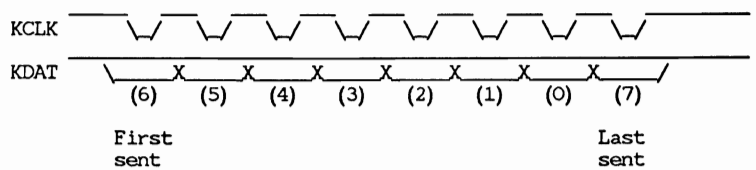
Keyboard communications:

The keyboard transmits eight bit data words serially to the main unit. Before the transmission starts, both KCLK and KDAT are high. The keyboard starts the transmission by putting out the first data bit (on KDAT), followed by a pulse on KCLK (low then high); then it puts out the second data bit and pulses KCLK; until all eight data bits have been sent. After the end of the last KCLK pulse, the keyboard pulls KDAT high again.

When the computer has received the eighth bit, it must pulse KDAT low for at least 75 microseconds, as a handshake signal to the keyboard.

All codes transmitted to the computer are rotated one bit before transmission. The transmitted order is therefore 6-5-4-3-2-1-0-7. The reason for this is to transmit the up/down flag last, in order to cause a key-up code to be transmitted in case the keyboard is forced to restore lost sync (explained in more detail below).

The KDAT line is active low; that is, a high level (+5V) is interpreted as 0, and a low level (0V) is interpreted as 1.



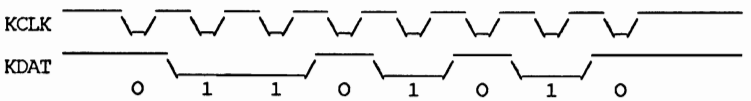
The keyboard processor sets the KDAT line about 20 microseconds before it pulls KCLK low. KCLK stay low for about 20 microseconds, then goes high again. The processor waits another 20 microseconds before changing KDAT.

Therefore, the bit rate during transmission is about 60 microseconds per bit, or 17 kbits/sec.

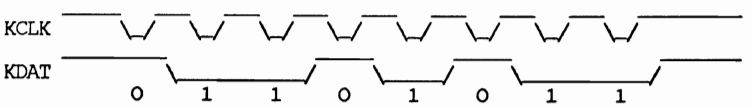
Keycodes:

Each key has a keycode associated with it (see accompanying table). Keycodes are always 7 bits long. The eighth bit is a "key-up"/"key-down" flag; a 0 (high level) means that the key was pushed down, and a 1 (low level) means the key was released (the CAPS LOCK key is different -- see below).

For example, here is a diagram of the "B" key being pushed down. The keycode for "B" is 35H = 00110101; due to the rotation of the byte, the bits transmitted are 01101010.



In the next example, the "B" key is released. The keycode is still 35H, except that bit 7 is set to indicate "key-up", resulting in a code of B5H = 10110101. After rotating, the transmission will be 01101011:



CAPS LOCK key:

This key is different from all the others in that it only generates a key-code when it is pushed down, never when it is released. However, the up/down bit is still used. When pushing the CAPS LOCK key turns on the CAPS LOCK LED, the up/down bit will be 0; when pushing CAPS LOCK shuts off the LED, the up/down bit will be 1.

"Out-of sync" condition:

Noise or other glitches may cause the keyboard to get out of sync with the computer. This means that the keyboard is finished transmitting a code, but the computer is somewhere in the middle of receiving it.

If this happens, then the keyboard will not receive its handshake pulse at the end of its transmission. If the handshake pulse does not arrive within 143 ms of the last clock of the transmission, then the keyboard will assume that the computer is still waiting for the rest of the transmission and is therefore out of sync. The keyboard will then attempt to restore sync by going into "resync mode". In this mode, the keyboard clocks out a 1 and waits for a handshake pulse. If none arrives within 143 ms, it clocks out another 1 and waits again. This process will continue until a handshake pulse arrives.

Once sync is restored, the keyboard will have clocked a garbage character into the computer. That is why the key-up/key-down flag is always transmitted last. Since the keyboard clocks out 1's to restore sync, the garbage character thus transmitted will appear as a key release, which is less dangerous than a key hit.

Whenever the keyboard detects that it has lost sync, it will assume that the computer failed to receive the keycode that it had been trying to transmit. Since the computer is unable to detect lost sync, it is the keyboard's responsibility to inform the computer of the disaster. It does this by transmitting a "lost sync" code (value F9H = 11111001) to the computer Then it retransmits the code that had been garbled.

Note: the only reason to transmit the "lost sync" code to the computer is to alert the software that something may be screwed up. The "lost sync" code does not help the recovery process, since the garbage keycode can't be deleted (water under the bridge), and the correct key code could simply be retransmitted without telling the computer that there was an error in the previous one.

Power-up sequence:

There are two possible ways for the keyboard to be powered up under normal circumstances:

Case i: Turn on computer with keyboard plugged in;
Case ii: Plug a keyboard into an already "on" computer.

The keyboard and computer must handle either case without causing any upset, particularly case ii.

The first thing the keyboard does on power-up is to perform a self-test. This involves a ROM checksum test, simple RAM test, and watchdog timer test.

Whenever the keyboard is powered up (or restarted -- see below), it must not transmit anything until it has achieved synchronization with the computer. The way it does this is by slowly clocking out 1 bits, as described above, until it receives a handshake pulse.

In Case i, the keyboard may continue this process for several minutes as the computer struggles to boot up and get running. The keyboard must stupidly continue clocking out 1's for however long is necessary, until it receives its handshake.

In Case ii, no more that eight clocks will be needed to achieve sync. In this case, however, the computer may be in any state imaginable, but must not be adversely affected by the garbage character it will receive. Again, since what it receives is a key release, the damage should be minimal. The keyboard driver must anticipate this happening and handle it, as should any application which uses raw keycodes.

Note: the keyboard must not transmit a "lost sync" code after resyncing due to a power-up or restart; only after resyncing due to a handshake time-out.

Once the keyboard and computer are in sync, the keyboard must inform the computer of the results of the self-test. If the self-test failed for any reason, a "selftest failed" code (value FCH = 11111100) is transmitted (the keyboard does not wait for a handshake pulse after sending the "selftest failed" code). After this, the keyboard processor goes into a loop where it blinks the CAPS LOCK LED to inform the user of the failure. The blinks are coded as bursts of 1, 2, 3, or 4 blinks, approx. one burst per second. 1 blink = ROM checksum failure; 2 blinks = RAM test failed; 3 blinks = watchdog timer test failed; 4 blinks = a short exists between two row lines or 1 of the 7 special keys (this last test isn't implemented yet).

If the self-test succeeds, then the keyboard will proceed to transmit any keys that are currently down. First, it sends an "initiate powerup key stream" code (value FDH = 11111101), followed by the key codes of all depressed keys (with keyup/down set to "down" for each key). After all keys are sent (usually there won't be any at all), a "terminate key stream" code (value FEH = 11111110) is sent. Finally, the CAPS LOCK LED is shut off. This marks the end of the startup sequence, and normal processing commences.

Note: these special codes, i.e., FCH et al, are 8-bit numbers; there is no up/down flag associated with them. However, the transmission bit order is the same as previously described.

The usual sequence of events will therefore be: power up; synchronize; transmit "initiate powerup key stream" (FDH); transmit "terminate key stream" (FEH).

Hard Reset

The keyboard has the additional task of resetting the computer on the command of the user. The user initiates hard reset by simultaneously pressing the CTRL key and the two "AMIGA" keys. The keyboard responds to this input by pulling KCLK low and starting a 500 ms timer. At the end of the 500 ms, the processor checks the three keys to see if they are still down, and if so, restarts the 500 ms timer. This continues until one or more of the three keys is released.

When one or more keys is released, then the processor will wait until the end of the 500 ms. Then it jumps to its startup code, which releases KCLK and restarts the keyboard.

Special Codes

I've mentioned some special codes that the keyboard uses to communicate with the main unit. I'll summarize them here:

Code	Name	Meaning
F9		Last key code bad, next code is same code retransmitted (used when keyboard and main unit get out of sync).
FA		Keyboard output buffer overflow
FB		6500/1 catastrophe, reports possibly lost (*)
FC		Keyboard selftest failed
FD		Initiate powerup key stream
FE		Terminate key stream
FF		Interrrupt main unit for key stream (Z8000 project only) (*)

(*) My algorithm doesn't use any of the asterisk'ed keycodes.

KEYBOARD HARDWARE

This is a description of the hardware insides of the Amiga keyboard.

This is only valid for the 2nd revision of the keyboard. This is the one with the watchdog timer.

PROCESSOR

The processor is a Rockwell/NCR/MOS Technologies 6500/1. It contains 2K bytes of ROM, 64 bytes of RAM, and 4 I/O ports of 8 bits each. It also has a 16 bit timer and edge detect capability on two of the I/O lines (port A bits 0 and 1). It has a built in crystal oscillator which we are running at 3.00 megahertz, which is divided internally to a 1.5 MHz internal clock.

RESET CIRCUITRY

There is a circuit for resetting the processor on power-on. The reset pulse lasts about 1 second after power is applied. The circuit also performs a "watchdog" function: once the processor starts scanning the key matrix, the watchdog timer is armed and will reset the processor if the scanning stops for more than about 50 milliseconds. The column 15 line is the trigger for the watchdog timer.

KEY MATRIX

There are 91 keys on the keyboard. 84 of them are arranged in a matrix of 6 rows and 15 columns (leaving 6 holes in the matrix). Each row is an input and has a pullup resistor to VCC on it (R=3.3K to 11K). Each column is an open-collector output with no pullup, i.e., it can only drive a column line low, not high. The program will drive columns one at a time and read rows.

The other 7 keys are special shift keys as follows: CTRL, left SHIFT, right SHIFT, left ALT, right ALT, left AMIGA, right AMIGA. Each of these keys has a dedicated input on the microprocessor. The actual port and bit numbers of all the keys are described below.

PORTS

As mentioned, there are 4 I/O ports of 8 bits each. The following table describes each port and the meaning of each bit:

PORT A -- 6500/1 address 080 hex

PA.0	In/Out	KDAT output/positive edge detect input (*)
PA.1	Out	KCLK output (*)
PA.2	In	Row 0 input (low = switch closed)
PA.3	In	Row 1 input
PA.4	In	Row 2 input
PA.5	In	Row 3 input
PA.6	In	Row 4 input
PA.7	In	Row 5 input

(*) These two bits are swapped from the previous code, to take advantage the positive edge detect capability of the PA.0 pin (easier to detect a handshake this way).

PORT B -- 6500/1 address 081 hex

PB.0	In	Right SHIFT key input (low = switch closed).
PB.1	In	Right ALT key input
PB.2	In	Right AMIGA key input
PB.3	In	CTRL key input
PB.4	In	Left SHIFT key input
PB.5	In	Left ALT key input
PB.6	In	Left AMIGA key input
PB.7	Out	CAPS LOCK LED control (high = LED on).

PORT C -- 6500/1 address 082 hex

PC.0	Out	Column 0 output (active low)
PC.1	Out	Column 1 output
PC.2	Out	Column 2 output
PC.3	Out	Column 3 output
PC.4	Out	Column 4 output
PC.5	Out	Column 5 output
PC.6	Out	Column 6 output
PC.7	Out	Column 7 output

PORT D -- 6500/1 address 083 hex

PD.0	Out	Column 8 output
PD.1	Out	Column 9 output
PD.2	Out	Column 10 output
PD.3	Out	Column 11 output
PD.4	Out	Column 12 output
PD.5	Out	Column 13 output
PD.6	Out	Column 14 output
PD.7	Out	Column 15 output (*)

(*) This keyboard has only 15 columns, numbered 0 to 14. However, the microprocessor software supports 16 columns, so we can use it in a future keyboard.

COUNTER PIN (input or output)

On the watchdog timer board, the counter pin is connected to the column 15 output. On the older non-watchdog version, the counter pin is unconnected. This provides the keyboard processor the ability to determine which type of board it it is installed in, so the new processor can work in old boards (with minor changes to the board).

NMI INPUT

This is connected to VCC and will therefore never turn on.

MATRIX TABLE

The following table shows which keys are readable in port A for each column you drive. The key code for each key is also included (in hex).

Column	Row 5 (Bit 7)	Row 4 (Bit 6)	Row 3 (Bit 5)	Row 2 (Bit 4)	Row 1 (Bit 3)	Row 0 (Bit 2)
15 (PD.7)	(spare) (0E)	(spare) (1C)	(spare) (2C)	(spare) (47)	(spare) (48)	(spare) (49)
14 (PD.6)	(spare) (5D)	(LEFT SHIFT) (30)	CAPS LOCK (62)	TAB (42)	~ ` (00)	ESC (45)
13 (PD.5)	(spare) (5E)	Z (31)	A (20)	Q (10)	! 1 (01)	(spare) (5A)
12 (PD.4)	9 (N.P.) (3F)	X (32)	S (21)	W (11)	@ 2 (02)	F1 (50)
11 (PD.3)	6 (N.P.) (2F)	C (33)	D (22)	E (12)	# 3 (03)	F2 (51)
10	3	V	F	R	$	F3

(PD.2)	(N.P.) (1F)	(34)	(23)	(13)	4 (04)	(52)
9 (PD.1)	. (N.P.) (3C)	B (35)	G (24)	T (14)	% 5 (05)	F4 (53)
8 (PD.0)	8 (N.P.) (3E)	N (36)	H (25)	Y (15)	^ 6 (06)	F5 (54)
7 (PC.7)	5 (N.P.) (2E)	M (37)	J (26)	U (16)	& 7 (07)	(spare) (5B)
6 (PC.6)	2 (N.P.) (1E)	< , (38)	K (27)	I (17)	* 8 (08)	F6 (55)
5 (PC.5)	ENTER (N.P.) (43)	> . (39)	L (28)	O (18)	(9 (09)	(spare) (5C)
4 (PC.4)	7 (N.P.) (3D)	? / (3A)	: ; (29)	P (19)	) 0 (0A)	F7 (56)
3 (PC.3)	4 (N.P.) (2D)	(spare) (3B)	" ' (2A)	{ [(1A)	_ - (0B)	F8 (57)
2 (PC.2)	1 (N.P.) (1D)	SPACE BAR (40)	(RET) (2B)	}] (1B)	+ = (0C)	F9 (58)
1 (PC.1)	0 (N.P.) (0F)	BACK SPACE (41)	DEL (46)	RET (44)	| \ (0D)	F10 (59)
0 (PC.0)	- (N.P.) (4A)	CURS DOWN (4D)	CURS RIGHT (4E)	CURS LEFT (4F)	CURS UP (4C)	HELP (5F)

The following table shows which keys are readable in port B (shift keys).

(Bit 6)	(Bit 5)	(Bit 4)	(Bit 3)	(Bit 2)	(Bit 1)	(Bit 0)
LEFT AMIGA (66)	LEFT ALT (64)	LEFT SHIFT (60)	CTRL (63)	RIGHT AMIGA (67)	RIGHT ALT (65)	RIGHT SHIFT (61)

GLOSSARY

aliasing distortion	A side effect of sound sampling, where two additional frequencies are produced, distorting the sound output.
ALT keys	Two keys on the keyboard to the left and right of the Amiga keys.
Amiga keys	Two keys on the keyboard to the left and right of the space bar.
AmigaDOS	The Amiga operating system.
amplitude	The voltage or current output expressed as volume from a sound speaker.
amplitude modulation	A means of increasing audio effects by using one audio channel to alter the amplitude of another.
attach mode	In sprites, a mode in which a sprite uses two DMA channels for additional colors. In sound production, combining two audio channels for frequency/amplitude modulation or for stereo sound.
automatic mode	The normal sprite mode in which the sprite DMA channel, once it starts up, automatically retrieves and displays all of the data for a sprite. The normal audio mode in which the system retrieves sound data automatically through DMA.
barrel shifter	Blitter circuit that allows movement of images on pixel boundaries.
baud rate	Rate of data transmission through a serial port.
beam counters	Registers that keep track of the position of the video beam.
bit-map	The complete definition of a display in memory, consisting of one or more bit-planes and information about how to organize the

rectangular display.

bit-plane — A contiguous series of display memory words, treated as if it were a rectangular shape.

bit-plane animation — A means of animating the display by moving around blocks of playfield data with the blitter.

blanking interval — Time period when the video beam is outside the display area.

blitter — DMA channel used for data copying and line drawing.

clear — To give a bit the value of 0.

CLI = Command Line Interpreter

clipping — When a portion of a sprite is outside the display window and thus is not visible.

collision — A means of detecting when sprites, playfields, or playfield objects attempt to overlap in the same pixel position or attempt to cross some pre-defined boundary.

color descriptor words — Pairs of words that define each line of a sprite.

color indirection — The method used by Amiga for coloring individual pixels in which the binary number formed from all the bits that define a given pixel refers to one of the 32 color registers.

color palette = color table

color register — One of 32 hardware registers containing colors that you can define.

color table — The set of 32 color registers.

Command Line Interpreter — The command line interface to system commands and utilities.

composite video — A video signal, transmitted over a single coaxial cable, which includes both picture and sync information.

controller — Hardware device, such as mouse or light pen, used to move the pointer or furnish some other input to the system.

coordinates — A pair of numbers shown in the form, (x,y), where x is an offset from the left side of the display or display window and y is an offset from the top.

Copper	Display synchronized coprocessor that resides on one of the Amiga custom chips and directs the graphics display.
coprocessor	Processor that adds its instruction set to that of the main processor.
cursor keys	Keys for moving something on the screen.
data fetch	The number of words fetched for each line of the display.
delay	In playfield horizontal scrolling, specifies how many pixels the picture will shift for each display field. Delay controls the speed of scrolling.
depth	Number of bit-planes in a display.
digital-to-analog converter	A device that converts a binary quantity to an analog level.
direct memory access	An arrangement whereby intelligent devices can read or write memory directly, without having to interrupt the processor.
display field	One complete scanning of the video beam from top to bottom of the video display screen.
display mode	One of the basic types of display; for example, high or low resolution, interlace or non-interlace, dual playfield.
display window	The portion of the bit-map selected for display. Also, the actual size of the on-screen display.

DMA = direct memory access

dual playfield mode	A display mode that allows you to manage two separate display memories, giving you two separately controllable displays at the same time.
equal tempered scale	A musical scale where each note is the 12th root of 2 above the note below it.
Exec	Low level primitives that support the AmigaDOS operating system.
font	A set of letters, numbers, and symbols sharing the same size and design.
display time	The amount of time to produce one display field, approximately 1/60th of a second.

frequency	The number of times per second a waveform repeats.
frequency modulation	A means of changing sound quality by using one audio channel to affect the period of the waveform produced by another channel. Frequency modulation increases or decreases the pitch of the sound.
genlock	An optional feature that allows you to bring in a graphics display from an external video source.
high resolution	A horizontal display mode where there are 640 pixels displayed across a horizontal line in a normal-size display.
hold-and-modify	A display mode that gives you extended color selection — up to 4,096 colors on the screen at one time.
horizontal blanking interval	Interval after the video beam has finished displaying one line and has not begun displaying another line.
interlace	A vertical display mode where 400 lines are displayed from top to bottom of the video display in a normal-size display.
joystick	A controller device that freely rotates and swings from left to right, pivoting from the bottom of the shaft; used to position something on the screen.
light pen	A controller device consisting of a stylus and tablet used for drawing something on the screen.
pointer register	Register that is continuously incremented to point to a series of memory locations.
low resolution	A horizontal display mode where 320 pixels are displayed across a horizontal line.
manual mode	Non-DMA output. In sprite display, a mode in which each line of a sprite is written in a separate operation. In audio output, a mode in which audio data words are written one at a time to the output.
minterm	One of eight possible logical combinations of data bits from three different data sources.
modulo	A number defining which data in memory belongs on each horizontal line of the display. Refers to the number of bytes in memory between the last word on one horizontal line and the beginning of the first word on the next line.

mouse	A controller device that can be rolled around to move something on the screen; also has buttons to give other forms of input.
multi-tasking	A system where many tasks can be operating at the same time, with no task forced to be aware of any other task.
non-interlace	A display mode where 200 lines are displayed from top to bottom of the video display in a normal-size display.
NTSC	National Television Standards Committee specification for composite video.
overscan	Area scanned by the video beam but not visible on the video display screen.
paddle controller	A game controller which uses a potentiometer (variable resistor) to position objects on the screen.
PAL	A European television standard similar to (but incompatible with) NTSC. Stands for "Phase Alternate Line".
parallel port	A connector on the back of the Amiga used to attach parallel printers and other parallel add-ons.
pitch	The quality of a sound expressed as its highness or lowness.
pixel	One of the small elements that makes up the video display. The smallest addressable element in the video display.
playfield	One of the basic elements in Amiga graphics; the background for all the other display elements.
playfield object	Sub-section of a playfield used in playfield animation.

playfield animation = bit-plane animation

polarity	True or false state of a bit.
potentiometer	An electrical analog device used to adjust some variable value.
primitives	Amiga graphics, text, and animation library functions.
quantization noise	Audio noise introduced by round-off errors when you are trying to reproduce a signal by approximation.

RAM	Random access (volatile) memory.
raster	The area in memory which completely defines a bit-map display.
read-only	Describes a register or memory area that can be read, but not written.
resolution	On a video display, the number of pixels that can be displayed in the horizontal and vertical directions.

ROM = read-only memory

sample	One of the segments of the time axis of a waveform.
sampling rate	The number of samples played per second.
sampling period	The value that determines how many clock cycles it takes to play one data sample.
scrolling	Moving a playfield smoothly in a vertical or horizontal direction.
serial port	A connector on the back of the Amiga used to attach modems and other serial add-ons.
set	To give a bit the value of 1.
shared memory	The RAM used in the Amiga for both display memory and executing programs.
sprite	Easily movable graphics object that is produced by one of the 8 sprite DMA channels and independent of the playfield display.
strobe address	An address you put out to the bus to in order to cause some other action to take place; the actual data written or read is ignored.
task	Operating system module or application program. Each task appears to have full control over its own virtual 68000 machine.
timbre	Tone quality of a sound.
trackball	A controller device that you spin with your hand to move something on the screen; may have buttons for other forms of input.
transparent	A special color register definition that allows a background color to show through. Used in dual playfield mode.

UART	The circuit that controls the serial link to peripheral devices; short for Universal Asynchronous Receiver/Transmitter.
video priority	Defines which objects (playfields and sprites) are shown in the foreground and which objects are in the background. Higher priority objects appear in front of lower priority objects.
video display	Everything that appears on the screen of a video monitor or television.
write-only	Describes a register that can be written to but cannot be read.

Index

68000
  interrupting, 2-16
aliasing
  audio, 5-23
animation, 6-3
area fill, 6-10
attachment
  audio, 5-19
  sprites, 4-23
audio
  channels, 5-6
  data, 5-6
  DMA, 5-12
  interrupt, 7-13
  non-DMA output, 5-28
  output jacks, 8-29
  RF, 8-30
  sampling period, 5-9
  volume, 5-8
background color, 3-8
barrel shifter, 6-8
basic playfield, 3-7
baud rate, 8-25
beam position, 7-9
beam position counter, 7-9
bit-plane pointers
  setting, 3-14
bit-planes
  in dual playfield mode, 3-29
  setting the number of, 3-9
blitter
  animation, 6-3
  area fill, 6-10
  ascending and descending addressing, 6-8
  copying, 6-2
  interrupt, 7-13
  line drawing, 6-13
  masking, 6-9
  minterms, 6-4
  modulos, 6-7
  multiple source, 6-3
  shifting, 6-8
  using the Copper, 2-16
  zero detection, 6-10
blitter registers
  in line draw mode, 6-14
collision, 7-6
color
  attached sprites, 4-23
  enabling, 3-23
  in dual playfield mode, 3-31
  in hold-and-modify, 3-52
  sample register contents, 3-56
  sprites, 4-7
color registers
  contents, 3-9
  loading, 3-9
  sprites, 4-32
color selection
  in high-resolution mode, 3-59
  in hold-and-modify, 3-59
  in low-resolution, 3-57
color table, 3-8
comparator, 4-26
controller port
  joystick, 8-2
  mouse, 8-2
  trackball, 8-2
controllers
  light pen, 8-10
  proportional, 8-6
Copper
  bus cycles used, 2-3
  horizontal beam position, 2-5
  in interlace mode, 2-15
  instruction lists, 2-9
  instructions
    summary, 2-17

interrupt, 7-13
interrupting the 68000, 2-16
loops and branches, 2-13
MOVE instruction, 2-4
SKIP instruction, 2-13
starting, 2-11
stopping, 2-11
vertical beam position, 2-6
wait instruction, 2-5
with the blitter, 2-16
Copper Danger Bit, 2-8
Copper registers
control register, 2-8
jump strobe addresses, 2-8
location registers, 2-7
copying data, 6-2
data fetch
high-resolution, 3-22
in basic playfield, 3-19
in horizontal scrolling, 3-47
data fetch start
normal, 3-19
data fetch stop
normal, 3-19
decibel values, 5-31
delay, 3-50
destinations, 6-3
disk
interrupt, 7-14
interrupts, 8-19
disk controller, 8-13
disk DMA, 8-15
display field, 3-2
display modes, 3-3
display window
maximum size, 3-42
normal, 3-18
starting positions
horizontal, 3-39
vertical, 3-40
stopping positions
horizontal, 3-40
vertical, 3-42
DMA
audio, 5-12
control, 7-15
Disk, 8-15
sprites, 4-28
dual playfields
bit-plane assignment, 3-29
color in, 3-31
enabling, 3-33
in high resolution mode, 3-31
priority, 3-32
scrolling, 3-32 , 3-27
exclusive area fill, 6-12
external interrupts, 7-12
field time, 3-2
genlock
effect on background color, 3-8
optional board, 3-53
high resolution
with dual playfields, 3-31
hold-and-modify, 3-52
inclusive area fill, 6-10
interlace
Copper in, 2-15
modulo, 3-22
setting interlace mode, 3-11
interrupt
disk, 7-14
interrupts
audio, 7-13
blitter, 7-13
control registers, 7-10
Copper, 7-13
disk, 8-19
external, 7-12
maskable, 7-10
non-maskable, 7-10
joystick
proportional, 8-7
reading, 8-5
keyboard
ghosting, 8-23
keycodes, 8-20
reading, 8-20
line drawing, 6-13
manual mode
in sprites, 4-25
masking
in the blitter, 6-9
memory allocation
formula for, 3-38

minterms, 6-4
modulation
  amplitude, 5-18
  frequency, 5-18 , 5-18
modulo
  blitter, 6-7
  in basic playfield, 3-20
  in horizontal scrolling, 3-47
  interlace, 3-22
  when memory picture is larger, 3-34
mouse
  buttons, 8-5
  reading, 8-4
musical scales, 5-29
noise
  audio, 5-23
normal playfield, 3-3
overscan, 3-17
paddle controller, 8-6
parallel port, 8-25
playfield
  displaying, 3-22
playfield DMA
  enabling, 3-22
playfields
  allocating memory, 3-12
  bit-plane pointers, 3-14
  collision, 7-6
  coloring the bit-planes, 3-14
  defining display window, 3-16
  dual playfield mode, 3-27
  memory required, 3-13
  multiple-playfield display, 3-53
playfields, scrolling, 3-43
playfield-sprite priority, 7-3
port
  parallel, 8-25
  serial, 8-25
ports
  disk, 8-13
priority
  dual playfields, 3-32
  playfield-sprite, 7-3
  sprites, 7-1
proportional controllers
  reading, 8-8
resolution
  setting resolution mode, 3-10
sampling
  period, 5-9
  rate, 5-21
scrolling
  horizontal, 3-45
  in dual field mode, 3-32
  vertical, 3-43
serial port, 8-25
sprites
  address pointers, 4-15
  attached, 4-23
  clipped, 4-5
  collision, 7-6
  color, 4-7
  comparator, 4-26
  data structure, 4-9
  DMA, 4-14
  manual, 4-25
  moving, 4-15
  overlapped, 4-20
  priorities, 7-1
  reuse, 4-18
  screen position, 4-2
  shape, 4-5
  size, 4-5
text
  packed, 6-9
Venn diagrams
  blitter minterms, 6-16
video beam position
  in Copper use, 2-5
video output, 8-30
volume, 5-8
waveforms
  audio, 5-1
zero detection, 6-10

Commodore Business Machines, Inc.
1200 Wilson Drive, West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue, Agincourt, Ontario, M1W 2K4