Hardware

COLLABORATORS			
	TITLE : Hardware		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Hardware

1.1 Amiga® Hardware Reference Manual: 2 Coprocessor Hardware

In this chapter, you will learn how to use the Amiga's graphics coprocessor (or Copper) and its simple instruction set to organize mid-screen register value modifications and pointer register set-up during the vertical blanking interval. The chapter shows how to organize Copper instructions into Copper lists, how to use Copper lists in interlaced 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.

1.2 2 Coprocessor Hardware / About the Copper

The Copper is a general purpose coprocessor that 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 680x0 to execute program logic; it can also 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

680x0.

When the WAIT condition has been satisfied, the Copper steals memory cycles from either the blitter or the 680x0 to move the specified data into the selected special-purpose register.

The Copper is a two-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, needs priority over only the 680x0 and the blitter (the DMA channel that handles animation, line drawing, and polygon filling).

As with all the other DMA channels in the Amiga system, the Copper can retrieve its instructions only from the chip RAM area of system memory.

1.3 2 Coprocessor Hardware / What is a Copper Instruction?

As a coprocessor, the Copper adds its own instruction set to the instructions already provided by the 680x0 CPU. The Copper has only three instructions, but you can do a lot with them:

*   WAIT for aspecific screen position specified as x and y coordinates.

*   MOVE animmediate data 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 two memory cycles and two instruction words each. Because only the odd memory cycles are requested by the Copper, four memory cycle times are required per instruction. The WAIT instruction requires three memory cycles and six memory cycle times; it takes one extra memory cycle to wake up.

Although the Copper can directly affect only machine registers, it can also affect memory indirectly 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.

1.4 2 Coprocessor Hardware / 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 procedure is shown in detail in the section called Summary of Copper Instructions .

```
FIRST MOVE 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 MOVE 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 $20 or above. (Hexadecimal numbers are distinguished from decimal numbers by the $ prefix.)

* Any register whose address is between $10 and $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 $10.

Appendix B contains all of the machine register addresses.

The following example MOVE instructions set bitplane pointer 1 to $21000 and bitplane pointer 2 to $25000. (All sample code segments are in assembly language.)

```
        DC.W    $00E0,$0002     ;Move $0002 to register $0E0 (BPL1PTH)
        DC.W    $00E2,$1000     ;Move $1000 to register $0E2 (BPL1PTL)
        DC.W    $00E4,$0002     ;Move $0002 to register $0E4 (BPL2PTH)
        DC.W    $00E6,$5000     ;Move $5000 to register $0E6 (BPL2PTL)
```

Normally, the appropriate assembler ".i" files are included so that names, rather than addresses, may be used for referencing hardware registers. It is strongly recommended that you reference all hardware addresses via their defined names in the system include files. This will allow you to more easily adapt your software to take advantage of future hardware or enhancements. For example:

```
        INCLUDE "hardware/custom.i"

        DC.W    bplpt+$00,$0002 ;Move $0002 into register $0E0 (BPL1PTH)
        DC.W    bplpt+$02,$1000 ;Move $1000 into register $0E2 (BPL1PTL)
        DC.W    bplpt+$04,$0002 ;Move $0002 into register $0E4 (BPL2PTH)
        DC.W    bplpt+$06,$5000 ;Move $5000 into register $0E6 (BPL2PTL)
```

For use in the hardware manual examples, we have made a special include file (see Appendix I) that defines all of the hardware register names based off of the "hardware/custom.i" file. This was done to make the examples easier to read from a hardware point of view. Most of the examples in this manual are here to help explain the hardware and are, in most cases, not useful without modification and a good deal of additional code.

1.5 2 Coprocessor Hardware / 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 WAIT 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 WAIT 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 example WAIT instruction waits for scan line 150 ($96) with the horizontal position masked off.

```
        DC.W    $9601,$FF00     ;Wait for line 150,
                                ;   ignore horizontal counters.
```

The following example WAIT instruction waits for scan line 255 and horizontal position 254. This event will never occur, so the Copper stops until the next vertical blanking interval begins.

```
        DC.W    $FFFF,$FFFE     ;Wait for line 255,
                                ;   H = 254 (ends Copper list).
```

To understand why position VP=$FF HP=$FE will never occur, you must look at the comparison operation of the Copper and the size restrictions of the position information. Line number 255 is a valid line to wait for, in fact it is the maximum value that will fit into this field. Since 255 is the maximum number, the next line will wrap to zero (line 256 will appear as a zero in the comparison.) The line number will never be greater than $FF. The horizontal position has a maximum value of $E2. This means that the largest number that will ever appear in the comparison is $FFE2. When waiting for $FFFE, the line $FF will be reached, but the horizontal position $FE will never happen. Thus, the position will never reach $FFFE.

You may be tempted to wait for horizontal position $FE (since it will never happen), and put a smaller number into the vertical position field. This will not lead to the desired result. The comparison operation is waiting for the beam position to become greater than or equal to the entered position. If the vertical position is not $FF, then as soon as the line number becomes higher than the entered number, the comparison will evaluate to true and the wait will end.

The following notes on horizontal and vertical beam position apply to both the WAIT instruction and to the SKIP . instruction. The SKIP instruction is described below in the Advanced Topics section.

 Horizontal Beam Position
 Vertical Beam Position
 The Comparison Enable Bits

1.6 2 / The WAIT Instruction / Horizontal Beam Position

The horizontal beam position has a value of $0 to $E2. 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).

All lines are not the same length in NTSC. Every other line is a long line (228 color clocks , 0-$E3), with the others being 227 color clocks long. In PAL, they are all 227 long. The display sees all these lines as 227 1/2 color clocks long, while the Copper sees alternating long and short lines.

1.7 2 / The WAIT Instruction / Vertical Beam Position

The vertical beam position can be resolved to one line, with a maximum value of 255. There are actually 262 NTSC (312 PAL) possible vertical positions. Some minor complications can occur if you want something to happen within these last six or seven scan lines. Because there are only eight bits of resolution for vertical beam position (allowing 256 different positions), one of the simplest ways to handle this is shown below.

```
     Copper Instruction           Explanation
    --------------------          -----------
     WAIT  for position (0,255)   At this point, the vertical
                                  counter appears to wrap to 0
                                  because the comparison works
                                  on the least significant bits
                                  of the vertical count

     WAIT  for any horizontal       Thus the total of 256 + 6 = 262
    position with vertical        lines of video beam travel during
    position 0 through 5,         which  Copper instructions  can be
    covering the last 6 lines     executed
    of the scan before
     vertical blanking  occurs.
```

```
    Note that the vertical is like the horizontal.
    ----------------------------------------------
    There are alternating long and short lines, there are also long and
    short fields (interlace only). In NTSC, the fields are 262, then 263
    lines and in PAL, 312, then 313 lines.  This alternation of lines and
    fields produces the standard NTSC 4 field repeating pattern:

          short field ending on short line
          long  field ending on  long line
          short field ending on  long line
          long  field ending on short line
          and back to the beginning...
```

One horizontal count takes one cycle of the system clock (processor is twice this).

```
          NTSC- 3,579,545 Hz
          PAL - 3,546,895 Hz
          genlocked- basic clock frequency plus or minus about 2%
```

1.8 2 / The WAIT Instruction / The Comparison Enable Bits

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

1.9 2 Coprocessor Hardware / Using the Copper Registers

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

1.10 2 / Using the Copper Registers / Location Registers

The Copper has two sets of location registers:

```
          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 contains 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 680x0, 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, these 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 .

```
   About Copper restart.
   ---------------------
   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 the end of  vertical blanking  occurs,
   the Copper is automatically forced to restart its operations at the
   address contained in COP1LC.
```

1.11 2 / Using the Copper Registers / Jump Strobe Address

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

COPJMP2 strobes this indirect address 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 .
```

1.12 2 / Using the Copper Registers / Control Register

The Copper can access some special-purpose registers all of the time, some registers only when a special control bit is set to a 1, and some registers not at all. The registers that the Copper can always affect are numbered $80 through $FF inclusive. (See Appendix B for a list of registers in address order.) Those it cannot affect at all are numbered $00 to $3E inclusive. The Copper control register is within this group ($00 to $3E). The rest of the registers, from $40 to $7E, are protected by a bit in the Copper control register.

In the Copper control register, called COPCON, only bit 1 is currently in use by the system. This bit, called CDANG (for Copper Danger Bit) protects all registers numbered between $40 and $7E inclusive. This range includes the blitter control registers. When CDANG is 0, these registers cannot be written by the Copper. When CDANG is 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.

```
   Warning:
   --------
   Keep in mind that the CDANG bit is cleared after a reset.
```

1.13 2 Coprocessor Hardware / 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 bitplane 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 bitplane pointer 1
    MOVE  data to bitplane pointer 2
    MOVE  data to sprite pointer 1, and so on.
```

As another 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 reuses of a sprite; however, every multiple reuse normally uses 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 that 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 (226,262) NTSC (or (226,312) PAL) 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 given 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
```

then the Copper will perform both moves, even though the instructions are out of sequence. The "greater than" specification prevents 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 could just as easily be

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

because multiple MOVE s can follow a single WAIT .

Complete Sample Copper List

1.14 2 / Putting Together a Copper List / Complete Sample Copper List

The following example shows a complete Copper list. This list is for two bitplanes -- one at $21000 and one at $25000. At the top of the screen, the color registers are loaded with the following values:

Register	Color
COLOR00	white
COLOR01	red
COLOR02	green
COLOR03	blue

At line 150 on the screen, the color registers are reloaded:

Register	Color
COLOR00	black
COLOR01	yellow
COLOR02	cyan
COLOR03	magenta

The complete Copper list follows.

```
;
; Notes: 1. Copper lists must be in Chip RAM.
;        2. Bitplane addresses used in the example are arbitrary.
;        3. Destination register addresses in Copper move instructions
;           are offsets from the base address of the custom chips.
;        4. As always, hardware manual examples assume that your
;           application has taken full control of the hardware, and is not
;           conflicting with operating system use of the same hardware.
;        5. Many of the examples just pick memory addresses to be used.
;           Normally you would need to allocate the required type of
;           memory from the system with AllocMem()
;        6. As stated earlier, the code examples are mainly to help
;           clarify the way the hardware works.
;        7. The following INCLUDE files are required by all example code
;            in this chapter.
;
        INCLUDE "exec/types.i"
        INCLUDE "hardware/custom.i"
        INCLUDE "hardware/dmabits.i"
        INCLUDE "hardware/hw_examples.i"

COPPERLIST:
;
;  Set up pointers to two bitplanes
;
        DC.W    BPL1PTH,$0002   ;Move $0002 into register $0E0 (BPL1PTH)
        DC.W    BPL1PTL,$1000   ;Move $1000 into register $0E2 (BPL1PTL)
        DC.W    BPL2PTH,$0002   ;Move $0002 into register $0E4 (BPL2PTH)
        DC.W    BPL2PTL,$5000   ;Move $5000 into register $0E6 (BPL2PTL)
;
;  Load color registers
;
        DC.W    COLOR00,$0FFF   ;Move white into register $180 (COLOR00)
        DC.W    COLOR01,$0F00   ;Move red into register   $182 (COLOR01)
        DC.W    COLOR02,$00F0   ;Move green into register $184 (COLOR02)
        DC.W    COLOR03,$000F   ;Move blue into register  $186 (COLOR03)
;
;   Specify 2 Lores bitplanes
;
        DC.W    BPLCON0,$2200   ;2 lores planes, coloron
;
;  Wait for line 150
;
        DC.W    $9601,$FF00     ;Wait for line 150, ignore horiz. position
;
;  Change color registers mid-display
;
        DC.W    COLOR00,$0000  ;Move black into register $0180 (COLOR00)
        DC.W    COLOR01,$0FF0  ;Move yellow into register $0182 (COLOR01)
        DC.W    COLOR02,$00FF  ;Move cyan into register $0184 (COLOR02)
        DC.W    COLOR03,$0F0F  ;Move magenta into register $0186 (COLOR03)
;
; End Copper list by waiting for the impossible
;
        DC.W    $FFFF,$FFFE    ;Wait for line 255, H = 254 (never happens)
```

For more information about color registers , see Chapter 3, "Playfield Hardware."

1.15 2 Coprocessor Hardware / Starting and Stopping the Copper

Starting the Copper After Reset
Stopping the Copper

1.16 2 / 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 during each vertical blanking time. The following sequence of instructions shows how to initialize a location register . It is assumed that the user has already created the correct Copper instruction list at location "mycoplist."

```
;
; Install the copper list
;
     LEA     CUSTOM,a1            ; a1 = address of custom chips
     LEA     MYCOPLIST(pc),a0     ; Address of our copper list
     MOVE.L  a0,COP1LC(a1)        ; Write whole longword address
     MOVE.W  COPJMP1(a1),d0       ; Causes copper to load PC from  COP1LC
;
; Then enable copper and raster dma
;
     MOVE.W  #(DMAF_SETCLR!DMAF_COPPER!DMAF_RASTER!DMAF_MASTER),DMACON(a1)
;
```

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.

1.17 2 / Starting and Stopping the Copper / Stopping the Copper

No stop instruction is 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 " link 2-4} 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."

1.18 2 Coprocessor Hardware / Advanced Topics

The SKIP Instruction
Copper Loops and Branches and Comparison Enable

1.19 2 / Advanced Topics / 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 SKIP 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 SKIP 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.

The following example SKIP instruction skips the instruction following it
if VP ( vertical beam position ) is greater than or equal to 100 ($64).

        DC.W    $6401,$FF01     ;If VP >= 100,
                                ;  skip next instruction (ignore HP)
```

1.20 2 / Advanced Topics / 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 care only about the value in the last four bits of the vertical position, you set only the last four compare enable bits, bits (11-8) in instruction word 2.

Not all of the bits in the beam counter may be masked. If you look at the description of the IR2 (second instruction word) you will notice that bit 15 is the blitter-finished-disable bit . This bit is not part of the beam counter comparison mask, it has its own meaning in the Copper WAIT instruction. Thus, you can not mask the most significant bit in WAIT or SKIP instructions. In most situations this limitation does not come into play, however, the following example shows how to deal with it.

1.21 2 / Advanced Topics / A Copper Loop Example

This example will instruct the Copper to issue an interrupt every 16 scan lines. It might seem that the way to do this would be to use a mask of $0F and then compare the result with $0F. This should compare "true" for $1F, $2F, $3F, etc. Since the test is for greater than or equal to, this would seem to allow checking for every 16th scan line. However, the highest order bit cannot be masked, so it will always appear in the comparisons. When the Copper is waiting for $0F and the vertical position is past 128 (hex $80), this test will always be true. In this case, the minimum value in the comparison will be $80, which is always greater than $0F, and the interrupt will happen on every scan line. Remember, the Copper only checks for greater than or equal to.

In the following example, the Copper lists have been made to loop. The COP1LC and COP2LC values are either set via the CPU or in the Copper list before this section of Copper code. Also, it is assumed that you have correctly installed an interrupt server for the Copper interrupt that will be generated every 16 lines. Note that these are non-interlaced scan lines.

Here's how it works. Both loops are, for the most part, exactly the same. In each, the Copper waits until the vertical position register has $xF (where x is any hex digit) in it, at which point we issue a Copper interrupt to the Amiga hardware. To make sure that the Copper does not loop back before the vertical position has changed and cause another interrupt on the same scan line, wait for the horizontal position to be $E2 after each interrupt. Position $E2 is horizontal position 113 for the Copper and the last real horizontal position available. This will force the Copper to the next line before the next WAIT . The loop is executed by writing to the COPJMP1 register. This causes the Copper to jump to the address that was initialized in COP1LC .

The masking problem described above makes this code fail after vertical position 127. A separate loop must be executed when vertical position is greater than or equal 127. When the vertical position becomes greater than or equal to 127, the the first loop instruction is skipped, dropping the Copper into the second loop. The second loop is much the same as the first, except that it waits for $xF with the high bit set (binary 1xxx1111). This is true for both the vertical and the horizontal WAIT instructions. To cause the second loop, write to the COPJMP2 register. The list is put into an infinite wait when VP >= 255 so that it will end before the vertical blank. At the end of the vertical blanking period COP1LC is written to by the operating system, causing the first loop to start up again.

COP1LC is written at the end of vertical blanking .

The COP1LC register is written at the end of the vertical blanking period by a graphics interrupt handler which is in the vertical blank interrupt server chain. As long as this server is intact, COP1LC will be correctly strobed at the end of each vertical blank.

```
;
;  This is the data for the Copper list.
;
;  It is assumed that COPPERL1 is loaded into  COP1LC  and
;  that COPPERL2 is loaded into  COP2LC  by some other code.
;
COPPERL1:
        DC.W    $0F01,$8F00   ; Wait for VP=0xxx1111
        DC.W    INTREQ,$8010  ; Set the copper interrupt bit...

        DC.W    $00E3,$80FE   ; Wait for Horizontal $E2
                              ; This is so the line gets finished before
                              ; we check if we are there  (The wait above)

        DC.W    $7F01,$7F01   ; Skip if VP>=127
        DC.W    COPJMP1,$0    ; Force a jump to  COP1LC

COPPERL2:
        DC.W    $8F01,$8F00   ; Wait for VP=1xxx1111
        DC.W    INTREQ,$8010  ; Set the copper interrupt bit...

        DC.W    $80E3,$80FE   ; Wait for Horizontal $E2
                              ; This is so the line gets finished before
                              ; we check if we are there  (The wait above)

        DC.W    $FF01,$FE01   ; Skip if VP>=255
        DC.W    COPJMP2,$0    ; Force a jump to  COP2LC

; Whatever cleanup copper code that might be needed here...
; Since there are 262 lines in NTSC, and we stopped at 255, there is a
; bit of time available

        DC.W    $FFFF,$FFFE     ; End of Copper list
;
```

1.22 2 / Advanced Topics / Using the Copper In Interlaced Mode

An interlaced bitplane display has twice the normal number of vertical lines on the screen. Whereas a normal NTSC display has 262 lines, an interlaced NTSC display has 524 lines. PAL has 312 lines normally and 625 in interlaced mode. In interlaced mode, the video beam scans the screen twice from top to bottom, displaying, in the case of NTSC, 262 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. Figure 2-1 shows how an interlaced display is stored in memory.

```
         Odd field            Even field
         (time t)          (time t + 16.6ms)        Data in Memory
     -----------------     -----------------     -----------------
                                                  _________________
                                                 |                 |
                                                 |        1        |
                                                 |_________________|
                                                 |                 |
      _________________     _________________    |        2        |
     |                 |   |                 |   |_________________|
     |        1        |   |        2        |   |                 |
     |_________________|   |_________________|   |        3        |
     |                 |   |                 |   |_________________|
     |        3        |   |        4        |   |                 |
     |_________________|   |_________________|   |        4        |
     |                 |   |                 |   |_________________|
     |        5        |   |        6        |   |                 |
     |_________________|   |_________________|   |        5        |
                                                 |_________________|
                                                 |                 |
                                                 |        6        |
                                                 |_________________|
```

Figure 2-1: Interlaced Bitplane in RAM

The system retrieves data for bitplane 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 bitplane pointer must contain a different value for alternate fields of the interlaced display.

Simply, the organization of the data in memory matches the apparent organization on the screen (i.e., odd and even lines are interlaced together). This is accomplished by having a separate Copper instruction list for each field to manage displaying the data.

To get the Copper to execute the correct list, you set an interrupt to the 680x0 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."

1.23 2 / Advanced Topics / 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-finished 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 .

1.24 2 / Advanced Topics / The Copper and the 680x0

On those occasions when the Copper's instructions do not suffice, you can interrupt the 680x0 and use its instruction set instead. The 680x0 can poll for interrupt flags set in the INTREQ register by various devices. To interrupt the 680x0, use the Copper MOVE instruction to store a 1 into the following bits of INTREQ :

Table 2-1: Interrupting the 680x0

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 680x0.

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

1.25 2 Coprocessor Hardware / Summary of Copper Instructions

The table below shows a summary of the bit positions for each of the Copper instructions . See Appendix A for a summary of all registers.

Table 2-2: Copper Instruction Summary

	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

ECS Copper.

For information relating to the Copper in the Enhanced Chip Set (ECS), see Appendix C .