JAGUAR™

WORKSHOP SERIES

#1

Minimum Object List Update

Introduction

This application, MOU.COF, focuses on the most basic (and necessary) components of a Jaguar program, namely, the creation and maintenance of an object list that is used by the Object Processor (OP) to render screen images.

To follow along with this example you will need the following files included in the \JAGUAR\WORKSHOP\MOU directory:

- `mou_init.s`
- `mou_list.s`
- `mou.inc`
- `makefile`
- `jaguar.bin`

In addition I will assume that you have properly installed your developer's toolkit and have the header files supplied by Atari in your include file directory.

Goal

This example application will display a 16-bit CRY bitmap image (contained in JAGUAR.BIN) and do required maintenance during the vertical blanking period. The application will proceed through the following steps:

1. Do basic hardware initialization and define a stack
2. Copy the bitmap image to an absolute location in RAM.
3. Initialize the video hardware.
4. Create an object list.
5. Define a vertical-blank interrupt handler.
6. Turn on video and begin list processing.
7. Release control to the debugging stub.

With the exception of step four, this code can be found in MOU_INIT.S. Step four is coded in MOU_LIST.S.

Program Initialization

MOU_INIT.S begins by including the global header file, JAGUAR.INC, and a program-specific header file named MOU.INC. These header files provide all of the constants used in the source code. The first instruction executed is as follows:

```
move.l  #$00070007,G_END
```

This instruction ensures that the Graphics Processing Unit (GPU) is configured to use Motorola MSB-LSB (big-endian) for its I/O registers. This line of code is required for all Jaguar programs. A similar line is required for **D_END** if the DSP is needed (which this sample doesn't).

```
move.w  #$FFFF,VI

move.l  #stopob,d0
swap    d0
move.l  d0,OLP
```

The first line disables video interrupts and is required to prevent interrupts from occurring in the middle of your setup routines. The next lines temporarily set the current object list to be a single stop object.

The next line of code you will find common to most Jaguar sample programs is:

```
move.l  #INITSTACK,a7
```

Most Jaguar programs will want to setup a stack. In this case, the equate **INITSTACK** is used. **INITSTACK** is defined in JAGUAR.INC to be $1FFFFC (the top longword of DRAM).

capable of
ws:

Next, a generic subroutine, **InitVideo**, is called to initialize the video registers. **InitVideo** is configuring video for any non-interlaced pixel resolution. The code for this subroutine follov

```
InitVideo:
                movem.l d0-d6,-(sp)

                move.w  CONFIG,d0
                andi.w  #VIDTYPE,d0     ; 0 = PAL, 1 = NTSC
                beq     palvals

                move.w  #NTSC_HMID,d2   ; Values defined in JAGUAR.INC
                move.w  #NTSC_WIDTH,d0

                move.w  #NTSC_VMID,d6
                move.w  #NTSC_HEIGHT,d4

                bra     calc_vals
palvals:
                move.w  #PAL_HMID,d2    ; Values defined in JAGUAR.INC
```

```
                move.w  #PAL_WIDTH,d0

                move.w  #PAL_VMID,d6
                move.w  #PAL_HEIGHT,d4
calc_vals:
                move.w  d0,width            ; Width of screen in clocks
                move.w  d4,height           ; Height of screen in half-lines

                move.w  d0,d1
                asr     #1,d1               ; Width/2

                sub.w   d1,d2               ; Mid - Width/2
                add.w   #4,d2               ; (Mid - Width/2)+4

                sub.w   #1,d1               ; Width/2 - 1
                ori.w   #$400,d1            ; (Width/2 - 1)|$400

                move.w  d1,a_hde
                move.w  d1,HDE

                move.w  d2,a_hdb
                move.w  d2,HDB1
                move.w  d2,HDB2

                move.w  d6,d5
                sub.w   d4,d5
                move.w  d5,a_vdb

                add.w   d4,d6
                move.w  d6,a_vde

                move.w  a_vdb,VDB
                move.w  #$FFFF,VDE          ; REQUIRED!!!

                move.l  #0,BORD1            ; Black Border
                move.l  #0,BG               ; Black Background

                movem.l (sp)+,d0-d6
                rts
```

This routine first determines whether the console is a NTSC or PAL machine and loads four registers with pre-defined values for the right console type. The variables ***width*** and ***height*** are then loaded with two of those constants describing the width of the screen in pixel clocks and the height of the screen in pixels.

To obtain the actual horizontal resolution of the screen in pixels, we must first choose a pixel divisor. The following table lists the available pixel divisors and the approximate resulting overscanned and non-overscanned resolutions:

Pixel Divisor	Non-Overscanned	Overscanned
1	1064	1330
2	532	665
3	355	442
4	266	332
5	213	266
6	177	222
7	152	190
8	133	166

Most of the workshop examples (including this one) will use a pixel divisor of four. This mode yields the closest approximation to square pixels and gives us plenty of pixels to work with. Whenever we need to know the width of our screen in pixels, the following formula may be used:

$$\text{pixel width} = \frac{\textit{width}}{\text{pixel divisor}}$$

Computing the vertical height of the screen is even easier. The *height* variable, set by our video initialization subroutine, is already in pixels.

The last lines of the video initialization sets the video border and background colors. The border color is the color used on those parts of the screen outside of the displayable region. When overscanning, this color does not matter. You should note that the **BORD1** and **BORD2** registers specify a color in 24-bit RGB. By setting both registers (using a longword write) to zero in our sample code we make the border black.

If the **BGEN** bit (#7) is set in the Video Mode register (we'll do this later), the line-buffer is initialized to the color specified in the **BG** register at the beginning of every scanline. This only has an effect in **RGB16** or **CRY16** mode and the contents of **BG** will be a CRY or 16-bit RGB color pixel depending upon the mode you're in. This example will use 16-bit CRY mode but since we're setting it to black, zero will work in either mode.

A Simple Object List

Jaguar video display is accomplished using an object list. The object list is consulted by the Object Processor at the start of every horizontal scanline to determine what needs to be drawn. As the screen is drawn and each scanline is successively rendered, certain parts of the object list are destroyed. For this reason, the object list must be updated during each vertical blank. Generally, you should save copies of the phrases which will get destroyed when you first create the list, then you can simply restore those fields from the saved copies.

The object list in this example is the minimum necessary to generate a display. It is arranged as follows:

Phrase	Object Type	Description
1	Branch	This object causes a branch to the Stop object if the VC register points past the visible screen. The **VC** register contai[ns the line] which is currently being prepared for display. Its valu[e is specified in] half-lines.
2	Branch	This object causes a branch to the stop object if the VC register points before the beginning of the visible screen.
3 & 4	Bitmap	This object contains the data for the Jaguar logo we [want to display] on screen. Bitmap objects take two phrases (16 byt[es) and must be] double-phrase aligned.
5	Stop	This object ends object list processing for the curren[t scan-line.]

The first two branch objects simply skip the rest of the list and jump straight to th[e stop object if the] vertical region being updated is outside of the area we want to be visible. This is [a required component] of every object list you set up. Because of a bug in the Jaguar chipset, the OP mu[st run every scanline] (this is done by setting a_vde to $FFFF in the video initialization). Please trust us [on this, bad things will] happen in the system if you ignore this step.

The bitmap object is responsible for the display of the Jaguar logo. The stop obje[ct simply terminates list] processing for the current scan-line.

The sample code places the object list into a buffer referenced by the label ***main_obj_list***. The buffer is where the list is first created and where it will be updated during every vertical-b[lank.]

The subroutine **InitLister** builds the initial copy of the object list in the buffer ***m[ain_obj_list]***. The subroutine begins as follows:

```
movem.l d1-d5/a0,-(sp)

lea     InitLister,a0
move.l  a0,d2

add.l   #(LISTSIZE-1)*8,d2
```

Register A0.l will be used as a roving list pointer which will be advanced as ea[ch phrase of the list is] written. D2.l is initialized with this code to contain a pointer to the stop object. [This pointer will be] needed for constructing each object in the list.

Throughout the entire routine, D1.l and D0.l will be used to temporarily hold t[he high and low long of] the phrase being constructed. The first object to be written is a branch object. T[o review, a branch object] is arranged as follows:

Branch Object

63	55	47	39	31	23	15	7	0
Unused			Link Pointer (Bits 21–3)		Unused	CC	YPOS	TYPE

We will start by initializing D1 and D0 to contain the object **TYPE, CC** (condition code), and **LINK** fields as follows:

```
clr.l    d1
move.l   #BRANCHOBJ|O_BRLT,d0
jsr      format_link
```

The branch object only branches if a specified condition is met. This condition is encoded in the **CC** field of the object. The following table lists the five possible condition codes:

Equate	CC	Description
O_BREQ	0	Branch if **YPOS** == **VC** or **YPOS** == $7FF.
O_BRGT	1	Branch if **YPOS** > **VC**.
O_BRLT	2	Branch if **YPOS** < **VC**.
O_BROP	3	Branch if the Object Processor Flag (**OBF**) is set.
O_BRHALF	4	Branch if on second half of display line (**HC & 1 == 1**).

The last line calls a subroutine which takes the address we previously stored in D2.l and transforms it as necessary to place it in the **LINK** field of the phrase. The **LINK** field indicates the address of the next object to process if the branch condition is met. If the branch condition is not met the next object in the list is processed. The **format_link** subroutine is as follows:

```
format_link:
        movem.l d2-d3,-(sp)

        andi.l  #$3FFFF8,d2      ; Ensure alignment
        move.l  d2,d3            ; Make a copy

        swap    d2               ; Equivalent to << 21
        clr.w   d2
        lsl.l   #5,d2
        or.l    d2,d0

        lsr.l   #8,d3            ; copy >> 11
        lsr.l   #3,d3
        or.l    d3,d1

        movem.l d2-d3,-(sp)
        rts
```

The only remaining field of the branch object that has not been filled in is the **YPOS** field. We want the branch object to branch if the **VC** register is past the end of the visible screen. To do this, the **YPOS** field is initialized with the same value the **VDE** register was initialized with. This value was stored in a variable called *a_vde* by the **InitVideo** routine. The following code retrieves this value, shifts it into place and stores it. Next, the phrase is stored into the buffer.

```
move.w   a_vde,d3         ; YPOS = a_vde
lsl.w    #3,d3            ; Shift to bits 13-3
or.w     d3,d0            ; Store it
```

```
move.l  d1,(a0)+        ; Store the phrase
move.l  d0,(a0)+        ; in the list buffer
```

The next phrase is written in a similar manner. First, the **CC** and **YPOS** fields are stripped from the last phrase. This branch object will branch if **VC** hasn't reached the top of visible screen yet so **YPOS** will be set to ***a_vdb*** and **CC** will be set to **YPOS** > **VC**. The code follows:

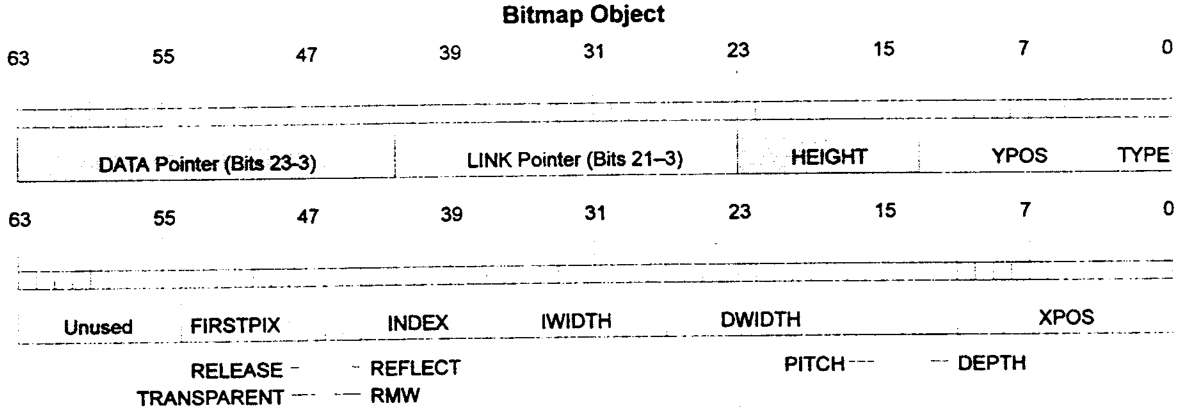
```
andi.l  #$FF000007,d0   ; Mask away YPOS and CC
ori.l   #O_BRGT,d0      ; YPOS > VC
move.w  a_vdb,d3        ; YPOS = a_vdb
lsl.w   #3,d3           ; Make it bits 13-3
or.w    d3,d0

move.l  d1,(a0)+        ; Store second branch object
move.l  d0,(a0)+
```

The Bitmap Object

The next object that needs to be written to the list buffer is the bitmap object. Bitmap object require two phrases of space and must be double-phrase aligned. Since our entire list is double-phrase aligned with the '.dphrase' statement and the bitmap object will be preceded with two phrases of branch objects we can be sure that the bitmap object will be properly aligned. The two phrases of a bitmap object are arranged as follows:

Bitmap Object



To begin processing the bitmap object, the temporary phrase storage registers must be cleared and the address of the stop object must be stored in the **LINK** field as follows:

```
clr.l   d1
clr.l   d0

jsr     format_link
```

The **LINK** field of a bitmap object contains the address of the next object to be processed. Because the address of the stop object remains in D2, a subroutine call to **format_link** is all that is necessary. You should note that the **TYPE** field does not need to be filled in because the bitmap object **TYPE** code is 0.

The next field to be filled in is **HEIGHT**. This field simply specifies the height of the bitmap in pixels. The sample code that follows takes the equate **BMP_HEIGHT** (defined in MOU.INC), shifts it into place, and stores it in our temporary phrase:

```
move.l   #BMP_HEIGHT,d5
lsl.l    #8,d5
lsl.l    #6,d5
or.l     d5,d0
```

The **YPOS** field of a bitmap object contains the vertical position where the bitmap will be displayed in half-lines. To center the bitmap in our example we use the following formula:

$$\mathrm{YPOS} = \left(\frac{\textit{height} - \mathrm{BMP\_HEIGHT}}{2}\right) \times 2 + \textit{a\_vdb}$$

Because **YPOS** must be specified in half-lines, the pixel result must be multiplied by two to convert it. ***a_vdb***, which is the topmost displayable scanline set by **InitVideo**, is already in half-lines. To simplify the code which sets **YPOS** below, both the division and multiplication may be removed because they cancel each other out in the equation. The constant **BMP_HEIGHT** is set in MOU.INC and is equal to the height of the bitmap in pixels. The result of the equation is AND'ed with $FFFE to ensure that the resulting value is even (which is required).

```
move.w   height,d3
sub.w    #BMP_HEIGHT,d3
add.w    a_vdb,d3
andi.w   #$FFFE,d3

lsl.w    #3,d3
or.w     d3,d0
```

The last field in the first phrase that needs to be completed is the **DATA** field. This field will contain a pointer to our sample bitmap.

For this example, the bitmap image is left in ROM (the Alpine board) and its address is assigned to the label **jagbits** by the linker. Under most circumstances you should copy bitmaps to RAM with the Blitter prior to displaying it. ROM access speed can be up to ten times slower than RAM (in the case of fetching object data, it is)! If you try to display more than a couple of bitmaps from ROM, the Object Processor will run out of time and your display will be distorted. The only reason we don't use a RAM copy in the first few examples is to avoid having to explore the Blitter as well as the Object Processor.

We also expect most bitmaps to be compressed in ROM. If you have enough ROM space to leave your bitmaps uncompressed then you should instead compress your bitmaps and enhance your game by adding a level, more music, etc.

You should note that the **DATA** field only encodes bits 23-3 of the bitmap address. Bits 2-0 aren't needed because the bitmap must be phrase-aligned. The following code forces the bitmap address to be phrase-aligned, shifts it into place, and stores it (note: if the bitmap isn't really phrase-aligned, it will just look funny on screen):

```
move.l   #jagbits,d3
andi.l   #$FFFFF0,d3
lsl.l    #8,d3
or.l     d3,d0
```

In the diagram of a bitmap object presented earlier, two fields had a gray background. These fields are modified by the Object Processor as it renders scanlines. For this reason, these portions of the object list must be updated during each vertical blank. This example does the least work possible by simply storing a copy of the phrase that gets destroyed so that it may be restored during the vertical blank. In order to do this, the following code stores the first phrase of the bitmap object with a copy in the variables ***bmp_highl*** and ***bmp_lowl***:

```
move.l   d1,(a0)+
move.l   d1,bmp_highl
move.l   d0,(a0)+
move.l   d0,bmp_lowl
```

The second phrase of a bitmap object contains more fields, however several may be set by simply OR'ing together equated values. The following code sets three fields. The **TRANS** bit is set causing the object processor to skip drawing pixels with the color $0000 effectively making these pixels transparent. The **DEPTH** field is set to **O_DEPTH16** indicating a 16-bit-per-pixel bitmap. The **PITCH** field is set to **O_NOGAP** which means that there is no gap between successive phrases of the bitmap data.

```
move.l   #O_TRANS,d1
move.l   #O_DEPTH16|O_NOGAP,d0
```

The next section of code creates the **XPOS** field. Again, we will center the bitmap horizontally in a similar manner to how we centered it vertically. There are some key differences, however. The value in width is the number of pixel clocks in a scanline. This must first be divided by the pixel divisor to determine the true horizontal screen resolution. You should also note that **XPOS** = 0 begins display at **HDB** so there is no reason to add the horizontal display offset as we did with **YPOS**. The constant **BMP_WIDTH** comes from MOU.INC and is equal to the bitmap width in pixels. Examine the following code:

```
move.w   width,d3          ; Width in clocks
lsr.w    #2,d3             ; /4 Pixel Divisor
sub.w    #BMP_WIDTH,d3     ; - BMP_WIDTH
lsr.w    #1,d3             ; /2 to center it
or.w     d3,d0             ; Store it
```

The last fields that must be set are **IWIDTH** and **DWIDTH. IWIDTH** contains the actual image width in phrases. **DWIDTH** contains the width (also in phrases) of the image to display. For now, these fields should be set to the same value. A later example will examine hardware clipping using these fields.

The following code sets the **IWIDTH** and **DWIDTH** fields to the constant **BMP_PHRASES** (defined in MOU.INC) and stores the second phrase of the bitmap object:

```
move.l  #BMP_PHRASES,d4
move.l  d4,d3

lsl.l   #8,d4           ; DWIDTH
lsl.l   #8,d4
lsl.l   #2,d4
or.l    d4,d0

lsl.l   #8,d4           ; IWIDTH Bits 31-28
lsl.l   #2,d4
or.l    d4,d0

lsr.l   #4,d3           ; IWIDTH Bits 37-32
```

```
move.l  d1,(a0)+        ; Stor
move.l  d0,(a0)+
```

Completing the List

The last object that is required in the object list is the stop

```
clr.l   d1
move.l  #(STOPOBJ|O_STOPINTS),

move.l  d1,(a0)+
move.l  d0,(a0)+
```

Besides the object **TYPE** field, the equate **O_STOPINT** processed (if we enable them later).

To complete the **InitLister** subroutine, the address of the pointer to the object list must be word-swapped) and retu

```
move.l  #main_obj_list,d0
swap    d0

movem.l (sp)+,d1-d5/a0
rts
```

Initializing Interrupts

The final subroutine called by the initialization segment i blank handler, enables video interrupts, and lowers the 6 allow CPU interrupts to occur.

All Jaguar interrupts appear to the CPU as Level 0 Auto Autovector interrupt occurs, the vector at address **LEVE** one type of interrupt is enabled, the **INT1** register must b

e phrase

object. The stop object is written as follows:

d0

S allows CPU stop object interrupts to be

list buffer is reloaded, word-swapped (the rned in D0 as shown by the following code:

is **InitVBint**. This routine installs the vertical 8000's interrupt priority level (IPL) to actually

vector interrupts. Whenever a Level 0 **L0** ($100) is jumped through. When more than be consulted to determine what type of interrupt actually caused the handler to be called. In this example that step isn't necessary because the only kind of interrupts we're concerned with are video interrupts.

The Jaguar Vertical Interrupt register (**VI** @ $F0004E) controls which half-scanline the vertical blank interrupt occurs (this must be an odd value). The following code installs the 68k Autovector handler and configures the **VI** register properly.

```
move.l   #UpdateList,LEVEL0

move.w   a_vde,d0
ori.w    #1,d0
move.w   d0,VI
```

The next section of code enables CPU video interrupts by setting the correct bit in INT1:

```
move.w   INT1,d0
ori.w    #C_VIDENA,d0
move.w   d0,INT1
```

Finally, the last section of the subroutine lowers the 68k IPL to level 0 to allow interrupts to occur.

```
move.w   sr,d0
andi.w   #$F8FF,d0
move.w   d0,sr
```

Enabling Video Processing

Only two more statements are required to enable the video display. The routine InitLister returned a pre-swapped pointer to the object list buffer in D0. This value must now be stored in the Object List Pointer (**OLP** @ $F00020). The final command reconfigures the video controller by correctly setting the Video Mode register (**VMODE** @ $F00028). Sample code follows:

```
move.l   d0,OLP
move.l   #CRY16|CSYNC|BGEN|PWIDTH4|VIDEN,VMODE
```

The **CRY16** equate enables 16-bit CRY mode. The **CSYNC** equate enables output to composite sync (which is required for television output). The **BGEN** equate causes the line buffer to be cleared to the background color prior to starting each scanline. The **PWIDTH4** equate enables a pixel divisor of four. Finally, the **VIDEN** equate enables video. Please note that Jaguar video should *never* be turned off by not setting the **VIDEN** flag.

The last instruction in our initialization is 'illegal'. This is a brute-force way to return control to the debugger. Most applications will enter their main logic loop at this point. Please note, however, that even though the debugger regains control, interrupts will continue to occur and be serviced by our handler.

The Vertical Blank Handler

The vertical blank handler for this sample is very simple. It must first restore any modified fields in the object list. Next, it must signal that it has handled the interrupt by using the sequence illustrated below:

```
UpdateList:
        move.l  a0,-(sp)
        move.l  #main_obj_list+BITMAP_OFF,a0

        move.l  bmp_highl,(a0)
        move.l  bmp_lowl,4(a0)

        move.w  #$101,INT1
        move.w  #$0,INT2

        move.l  (sp)+,a0
        rte
```

The constant **BITMAP_OFF** comes from MOU.INC and is the offset in bytes from the beginning of the list to the first phrase of the bitmap. Because this is an interrupt routine it must end with the 68k RTE instruction.

Assembly and Linkage

Though a MAKEFILE for the sample code is provided, different developers may choose different development environments for assembly and linkage. This section will only illustrate the command line switches used with MADMAC and ALN and why they were chosen.

Each assembly file is assembled with MADMAC with the command line options '**-fb**' and '**-g**'. The '-fb' switch causes MADMAC to output BSD format object files (the format strongly recommended for Jaguar development). The '-g' switch causes source-level information to be added to the object file.

The following table shows the flags used with the Atari Linker ALN and their purpose:

Switch	Meaning
-v -v	Enable medium-verbosity. The **-v** switch may be used from zero to three times for increasing levels of verbosity.
-e	Output a COFF format executable.
-g	Place source-level information in the output file.
-l	Include local as well as global symbols in the output file.
-rd	Align each object module to a double-phrase boundary.
-a 802000 x 4000	Create an absolute file with the TEXT segment starting at $802000, the DATA segment being contiguous with the TEXT segment, and the BSS segment starting at $4000.
-i jaguar.bin jagbits	Include a raw binary file named JAGUAR.BIN. The start address of the file will be assigned to the label *'jagbits'*. The end address of the label will be assigned the label *'jagbitsx'*.
-o mou.cof	Name the output file MOU.COF.

Running MOU.COF

Once MOU.COF has been successfully output, the sample program may be easily transferred to the ROMULATOR by typing 'rdbjag mou' or 'wdb mou' at a DOS or TOS command line prompt depending upon which debugger you prefer. You should ensure that the ROMULATOR's write-inhibit switch is not enabled or the file will not be correctly transferred. By the placing the name of the file on the command line it will be automatically loaded as an absolute file. To load the file after the debugger has started, type 'aread mou.cof'.

To start the sample program and display the Jaguar logo, simply type 'g 802000' and hit return. The sample program may also be started by resetting the Alpine while holding down the 'B' button on Joypad 1.

JAGUAR™

Copyright ©1994 Atari Corp.

WORKSHOP SERIES

#2

Moving a Bitmap with the Object Processor

Introduction

After reading through the first installment in this series you should now be able to construct a basic object list and maintain it during the vertical blank. This document will expand upon the first example, adding motion to the bitmap that is displayed. Each *Workshop Series* tutorial will not spend much time reviewing old material. Each installment will usually only talk about the differences between the current example and the last.

To follow along with this tutorial you will want the source code files to the MOVE.COF executable which may be found in the \JAGUAR\WORKSHOP\MOVE directory:

- `mov_init.s`
- `mov_list.s`
- `mov_move.s`
- `move.inc`
- `jaguar.bin`
- `makefile`

Goal

As with our last example, this sample code will display a 16-bit CRY Jaguar logo. This time, however, the code will update the position of the object during each vertical blank so it moves around, reversing direction each time it hits the edge of the display area.

Program Initialization

The source file MOV_INIT.S is identical to the last example's initialization code with the exception of the following line (highlighted in bold):

```
        jsr     InitVideo
        jsr     InitMoveVars
        jsr     InitLister
        jsr     InitVBint
```

The external subroutine **InitMoveVars** is located in MOV_MOVE.S. It initializes a few BSS variables that we will use to track the object's movement as follows:

```
InitMoveVars:
        move.l  d0,-(sp)

        move.w  #X_MOTION,x_motion
        move.w  #Y_MOTION,y_motion
```

```
        clr.w   frame_count

        clr.w   x_min

        move.w  width,d0
        lsr.w   #2,d0
        sub.w   #BMP_WIDTH,d0
        sub.w   #1,d0
        move.w  d0,x_max

        move.w  a_vdb,d0
        andi.w  #$FFFE,d0
        move.w  d0,y_min

        move.w  a_vde,d0
        sub.w   #BMP_LINES,d0
        andi.w  #$FFFE,d0
        sub.w   #2,d0
        move.w  d0,y_max

        move.l  (sp)+,d0
        rts
```

The variables ***x_motion*** and ***y_motion*** are initialized with constants stored in MOVE.INC. By altering these constants you can change the speed and initial direction of the bitmap's motion (negative values move up and to the left, positive values move down and to the right).

The variable ***frame_count*** is initialized to zero. This variable will be incremented each time a vertical blank occurs and is zeroed each time we actually move the object. This allows the sample code to set a frequency (some divisor of the frame rate) at which the bitmap will be updated.

The rest of the initialization sets up four variables that will contain the logical extents of the viewscreen. Each time the object is moved its position is compared to the values in these variables and its direction is reversed if necessary. You will also notice that the width and height of the bitmap are subtracted from the width and height of the bounding rectangle. This is to account for the fact that the movement constraints must be relative to the upper-left hand corner of the bitmap.

Building the Object List

In this example we can use the same object list that was used in MOU.COF. The only difference is that a copy of the bitmap's initial **XPOS** and **YPOS** are stored in the variables ***x_pos*** and ***y_pos***.

The Vertical Blank Handler

As with MOU.COF, the **UpdateList** routine is called during each vertical blank. It updates the fields of the object list that were *modified* by the object processor. Because this example requires very little work to be done to move a bitmap around, all of this processing is done during the vertical blank. This also allows us to return control to the debugger so we can manipulate the movement variables in real time.

The Programmable Interrupt Timer would normally be used to regulate the speed of processing game logic (or in this case, the speed of the moving bitmap) however, for this example, the frequency of the vertical blank itself will be used as the timer.

Moving the Object

After saving registers, the very first thing **UpdateList** does is to call the routine **MoveBitmap** which can be found in MOV_MOVE.S. **MoveBitmap** starts out by incrementing the variable **frame_count**. By comparing the ***frame_count*** variable with the pre-defined constant **UPDATE_FREQ** (defined in MOVE.INC) the sample code determines whether the subroutine will actually modify the object position variables or wait for more frames to occur first. The code to this logic follows:

```
MoveBitmap:
                movem.l d0-dl,-(sp)

                move.w  frame_count,d0
                add.w   #1,d0
                cmp.w   #UPDATE_FREQ,d0
                beq     do_move

                move.w  d0,frame_count
                bra     move_done
do_move:
                clr.w   frame_count
```

When the subroutine actually gets the chance to update the object's position it must first check to ensure that the object remains within the bounds set by the ***x_min***, ***x_max***, ***y_min***, and ***y_max*** variables. If the object reaches the limit of these boundaries, the appropriate motion variable is negated to reverse its direction. Finally, the motion variable for each direction is added to the object's position variable and the function returns. The remaining code for this function follows:

```
                move.w  x_pos,d0        ; verify X range
                cmp.w   x_min,d0
                ble     change_x        ; if at left edge

                cmp.w   x_max,d0        ; or at right edge
                blt     add_xmot
change_x:
                neg.w   x_motion        ; reverse X direction
add_xmot:
                add.w   x_motion        ; add motion amount

                move.w  y_pos,dl        ; verify Y range
                cmp.w   y_min,dl
                ble     change_y        ; if at top edge

                cmp.w   y_max,dl        ; or at bottom edge
                blt     add_ymot
change_y:
                neg.w   y_motion        l reverse Y direction
add_ymot:
                add.w   y_motion,dl     ; add motion amount

                move.w  d0,x_pos        ; store new values
                move.w  dl,y_pos
move_done:
                movem.l (sp)+,d0-dl
                rts
```

Updating the Object

During each vertical blank, the interrupt handler **UpdateList** restores the stored copy of the first bitmap phrase which was modified by the object processor. As an additional step, however, the YPOS portion of that phrase is stripped away with an AND instruction and replaced with the contents of the variable *y_pos*. The following code illustrates the updating of the first phrase.

```
move.l   #main_obj_list+BITMAP_OFF,a0

move.l   bmp_highl,(a0)          ; restore first longword
move.l   bmp_lowl,d0             ; grab long with YPOS

andi.l   #$FFFFC007,d0           ; strip old value

move.w   y_pos,d1                ; and replace new
lsl.w    #3,d1
or.w     d1,d0

move.l   d0,4(a0)                ; now store it
```

Next, the XPOS field in the second phrase of the bitmap must be updated. This time, however, the phrase to be modified comes directly from the object list buffer. This is possible since the Object Processor never modified this phrase. The following code updates the XPOS field in the second phrase of the bitmap and exits the interrupt handler.

```
move.l   12(a0),d0               ; Low long of phrase 2
andi.l   #$FFFFF000,d0           ; Extract XPOS

move.w   x_pos,d1                ; Fill in current XPOS
or.w     d1,d0

move.l   d0,12(a0)               ; Store it back

move.w   #$101,INT1
move.w   #0,INT2

movem.l  (sp)+,d0-d1/a0
rte
```

Running MOVE.COF

Use your favorite variation of MAKE to create MOVE.COF (the flags should be the same as MOU.COF) and load it into the debugger by typing 'wdb move' or 'rdbjag move'. Type 'g' and hit return to see the results of this sample program.

As an experiment you can try modifying the values for **X_MOTION**, **Y_MOTION**, and **UPDATE_FREQ** in MOVE.INC. You will get different horizontal and vertical speeds depending on the values you select.

❁❁❁❁❁

JAGUAR™

Copyright ©1994 Atari Corp.

WORKSHOP SERIES #3

Clipping a Bitmap Object with the Object Processor

Introduction

This example builds upon the original example in this series, MOU.COF, to demonstrate the built-in capability of the Object Processor to horizontally clip bitmap objects. Before examining this example, please familiarize your self with *Workshop Series #1: Minimum Object List Update*.

The following source code files to CLIP.COF may be found in the \JAGUAR\WORKSHOP\CLIP sub-directory:

- `clp_init.s`
- `clp_list.s`
- `clp_clip.s`
- `clip.inc`
- `jaguar.bin`
- `makefile`

Under Construction

The tutorial document for this example has not yet been created. Please refer to the source code comments in each of the files for specific information about this example.

JAGUAR™

WORKSHOP SERIES

#4

Scaling a Bitmap Object with the Object Processor

Introduction

This example builds upon the original example in this series, MOU.COF, to demonstrate the built-in capability of the Object Processor to scale bitmap objects. Before examining this example, please familiarize your self with *Workshop Series #1: Minimum Object List Update*.

The following source code files to SCALE.COF may be found in the \JAGUAR\WORKSHOP\SCALE sub-directory:

- `scl_init.s`
- `scl_list.s`
- `scl_scal.s`
- `scale.inc`
- `jaguar.bin`
- `makefile`

Under Construction

The tutorial document for this example has not yet been created. Please refer to the source code comments in each of the files for specific information about this example.

JAGUAR™

WORKSHOP SERIES

#6

GPU Interrupt Object Processing

Introduction

This example builds upon the original example in this series, MOU.COF, to demonstrate GPU interrupt objects. Before examining this example, please familiarize yourself with *Workshop Series #1: Minimum Object List Update.*

...may be found in the

WORKSHOP SERIES

#12

Rotating a Bitmap with the Blitter

Introduction

This example demonstrates bitmap rotation using the Blitter. Initialization and object list creation/maintenance is handled in the same manner as the first *Workshop Series* example, MOU.COF. Before examining this example, please familiarize yourself with *Workshop Series #1: Minimum Object List Update*.

The following source code files for JAGROT.COF may be found in the \JAGUAR\WORKSHOP\JAGROT directory:

- jr_init.s
- jr_list.s
- jr_grot.s
- jr.inc
- jaguar.bin
- makefile

Under Construction

The tutorial document for this example has not yet been created. Please refer to the source code comments in each of the files for specific information about this example.