# •Beginners Mac Assembly•

##  Chapter 5 

The problem with using the OS to handle all your programs functions, is that sometimes there isn't a routine to do what you want to do, and sometimes the os is just too slow. For example - how do you clear the whole of the Mac's screen. If you were writing a game, would you be happy with it running slowly in a window, or would it be better to use the whole screen. Speaking for myself, my general aim when writing a fast machine code program is to dump the operating system at the earliest opportunity!

Why you may ask - well the operating system when running needs to use some of the processors' power - and games need all the power they can get.
When the Mac is running a game, you don't need it to be checking for network traffic, or if it needs to print a file, or copy files, or empty the trash, or a thousand and one other things it regularly checks for - you want it to run your game as fast as it can - that's why we stop the OS.

Suppose you needed to draw 50 aliens on the screen. If you were to call the OS to do this it could take 20 milliseconds to ddraw each alien, which is one fiftieth of a second, or 1 second to draw 50 off them - far too slow for a game. But if you were to write your own routine in machine code to do it, it would run a lot faster for two reasons. Firstly the majority of the Mac OS is written in a mixture of C and Pascal, which are (possibly) slower than machine code. Secondly, because the routines have to be very flexible, they are overly complicated for what we want - we can tailor the routine specifically for drawing our type of objects.

**Macintosh Ram Based Video**

This section is going to deal with the internal video of your Mac. If you drop into Macsbug and type "devlist" without the quotes you should see something like this:

```
Displaying GDevice at 800050BC
 800050BC  gdRefNum           FFCF
 800050BE  gdID               0000
 800050C0  gdType             0000
 800050C2  gdITable           0000211C -> 000D0D10 ->
 800050C6  gdResPref          0004
 800050C8  gdSearchProc       NIL
 800050CC  gdCompProc         NIL
 800050D0  gdFlags            BC01
 800050D2  gdPMap             00002110 -> 8000510C ->
 8000510C    baseAddr         50F40000 is a bad pointer


 80005110    rowBytes         8400
 80005112    bounds           #0 #0 #480 #640
```

```
 8000511A    pmVersion        0000
 8000511C    packType         0000
 8000511E    packSize         00000000
 80005122    hRes             00480000
 80005126    vRes             00480000
 8000512A    pixelType        0000
 8000512C    pixelSize        0004
 8000512E    cmpCount         0001
 80005130    cmpSize          0004
 80005132    planeBytes       00000000
 80005136    pmTable          0000210C -> 000C9460 ->
 000C9460      ctSeed         00000403
 000C9464      ctFlags        8000
 000C9466      ctSize         000F
 000C9468      ctTable
 000C9468      0 value        0800
 000C946A        rgb
 000C946A          red        #65535
 000C946C          green      #65535
 000C946E          blue       #65535
 000C9470      1 value        0800
 000C9472        rgb
 000C9472          red        #64512
 000C9474          green      #62333
 000C9476          blue       #1327
 8000513A    pmReserved       00000000
 800050D6  gdRefCon           00000000
 800050DA  gdNextGD           NIL
 800050DE  gdRect             #0 #0 #480 #640
 800050E6  gdMode             00000082
 800050EA  gdCCBytes          0000
 800050EC  gdCCDepth          0000
 800050EE  gdCCXData          00002118 -> 80003768 ->
 800050F2  gdCCXMask          00002114 -> 80003B70 ->
 800050F6  gdReserved         00000000
```

This is the details pertaining to your current graphics device. From this information we can program the Mac's screen directly, without having to go through the operating system.

If we can get at this information in our program, then we can work on any mac screen.
If you try this in Macsbug on your machine, the chances are that the addresses of this data will be different - why is that? Well each machine is different (state the obvious stu!), but the machine has to work the same as any other Mac. To get round this problem, we have what are called “system variables”. System variables are addresses way down at the low end of memory, where your Mac stores the addresses of important hardware and software entry points. One of these system variables is called “devicelist”, in here is the address of a memory location that holds the address of the video control data. The all important address of devlist is $08A8.

So if we get the address from here, into a0, then the contents of the address in a0 is where the video control data lives:

```
devicelist: equ         $08A8
            move.l      devicelist,a0     get devicelist in a0
```

```
            move.l      (a0),a0     address of video graphics device now in a0
```
By reading the address out of the system variables area, we ensure that this will run on any Mac that uses the internal video. For a complete list of system variables, see appendix A (not included in the shareware distribution).

**Pixels and colours.**

How do we set a point on the Mac screen to a certain colour?

The Mac's internal video works with the in built VRAM, or Video Random Access Memory. The more colours you have displayed, the more memory it takes, as each individual dot (pixel) on the screen can be 1 of many colours - the more colours the more bits of information are needed to define the colour.

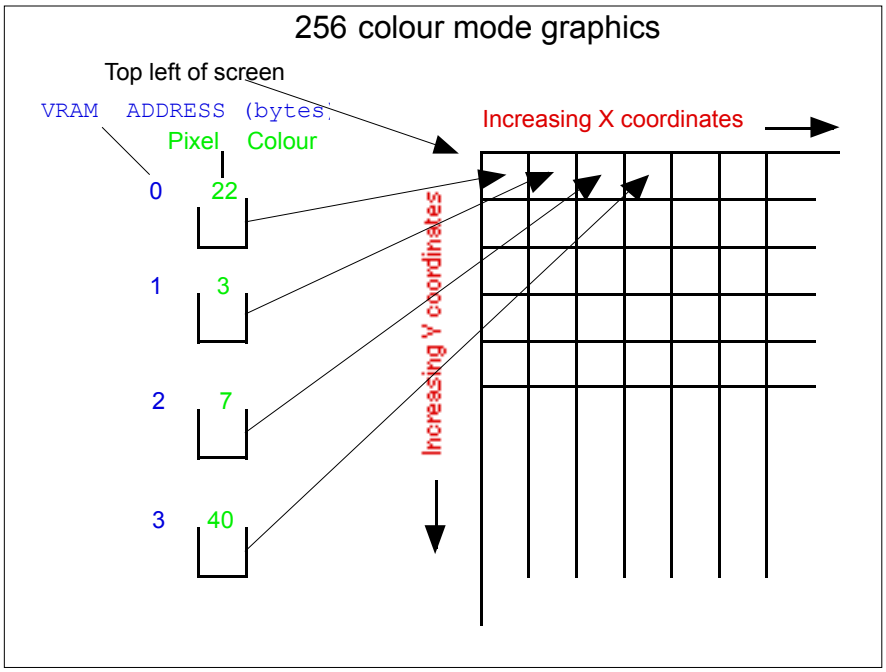
If the screen is in 1 bit mode, or black and white, then each dot on the screen is either off (black) or on (white). Therefore each pixel on the screen can be defined with just 1 bit. As we know, memory is laid out in bytes, so each byte of VRAM can hold 8 pixels in black and white mode.

For a screen resolution of 640 by 480, each screen line requires 640 pixels divided by 8 bits in a byte, or 80 bytes.
As there are 480 lines, the total memory required is 80 bytes times 480 lines or 38400 bytes.

If we now increase the number of colours to 16 - to define 16 colours requires 4 bits. So each line of 640 pixels requires 320 bytes (640 pixels times half a byte (4 bits)). The full screen takes 320 bytes per line, times 480 lines, which equals 153600 bytes. Quite a large jump from 38400 bytes in B/W mode, which explains why your Mac runs fastest in B/W!

Now lets move to 256 colour mode - how many bits does it take to define 1 pixel? To define 256 colours we need 8 bits, or 1 byte for each individual pixel. Therefore each screen line is 640 bytes, and the full screen is 640 bytes per line times 480, which is 307200 bytes - a lot of memory in anybody’s books!

We will deal with 256 colour mode, as this is rapidly becoming the standard for Mac games. Ideally, in your programs, you should check the current mode, and run to that.




In the above diagram, you should be able to see that the colour of the first pixel on the screen (top left corner) is defined by the first byte of VRAM. The second byte defines the colour of the second pixel and so on right up to pixel 639 (pixels are numbered from 0).

You would expect the first pixel of the second line to be defined by byte 640, as each line is 640 pixels wide - this may or may not be the case depending on the Mac. To find the physical length (in bytes) of each screen line, we need to examine the system variable "screenrow" at address $0106. This word tells us the physical length of each screen line - in most cases, where the Mac is connected to a 14 or 15 inch monitor, this will be 1024 if you have 512K of VRAM. The pixels in line 0 (the top line) of the screen start at VRAM address 0, the pixels for line 1 of the screen start at VRAM address 1024, line 3 at 2048 and so on.

**Palettes or CLUTS.**

When we say colour 3, what exactly do we mean? How does the Mac know what colour 3 is?
VRAM is accessed by two chips. One is the processor when it wants to set pixels in the VRAM, the other is the video chip. Approximately 65 times a second, the video chip has to read every single byte in VRAM to display the picture on the monitor. In 256 colour mode, each byte defines a specific colour.

As the video chip gets a byte from VRAM, it uses the colour number to index what's called the colour lookup table, or CLUT. For example it gets a byte from VRAM that contains 3. The video chip then looks up location 3 in the CLUT.

Each CLUT location is four words. You can see this in the devlist above as rgb. These four words define what colour the colour really is. We're not really interested in the first word, but the next three make up the red, green and blue values of the colour. The larger the word, the more of that colour there is.
As an aside, if each red grean and blue component is a word, the Mac can display a possible 2 to the power of 48 colours.

Commodore brag about the Amiga's possible 16.7 million colours - wowee - 2 to the power of 48 is (roughly) 128 million, million colours! (how come Apple never advertise these facts?)

As any colour can be defined by mixing red, green and blue, the video chip sends the three words as three voltages to your monitor tube. Inside the tube are three guns, one for the red signal, one for the green and one for the blue. The guns send out streams of electrons - the higher the voltage applied to the gun, the more electrons the gun shoots out.
The electrons travel down the tube, until they hit the screen. The screen is coated with three different types of phosphorus, one for red, one for green and one for blue. The electrons from the red gun hit only the red phosphorus, the same for the green and blue.

The phosphorus is laid onto the screen as tiny dots. When the electrons hit a dot, the chemical properties of phosphorus enable it to glow. The more electrons, the brighter the glow.

Phew, don't worry if you didn't follow that, its enough that you understand the colours are defined in the colour look up table, so when the video chip gets a byte from VRAM, it gets the colour definition out of the CLUT and sends it to the monitor.

Now the crucial test - we want to set the pixel at 300x,20y to colour 3. How do we calculate the VRAM address of the byte?

Start by calculating the start address of the y coordinate, in this case 20.
We need to length (in bytes) of each line from Screenrow, then we can multiply this by the y coordinate to get the start address of this line.
If Screenrow contain 1024, then the start address of the Y coordinate line is 1024 times 20, which is 20480.

This is the offset in VRAM of the start of line 20. Because in 256 colour mode, each byte is a pixel, all we need to do is add the x coordinate to this address, then move the colour (as a byte) into the new VRAM address.

20480 plus 300 (the x coordinate of the pixel) = 20780.

The address 20780 is the VRAM address of the byte that defines the colour of the pixel at coordinates 300,20. If we move 3 as a byte into this VRAM address, this pixels' colour will become 3.

All we need to know now is where the VRAM lives in the 68000's addressing space - i.e. what address the VRAM lives at. Once we know this, we can add it to the VRAM address to give us the absolute address that we must set to 3.

To find the VRAM address, we have to examine the device list for the video controller.

```
Displaying GDevice at 800050BC
 800050BC  gdRefNum           FFCF
 800050BE  gdID               0000
 800050C0  gdType             0000
 800050C2  gdITable           0000211C -> 000D0D10 ->
 800050C6  gdResPref          0004
 800050C8  gdSearchProc       NIL
 800050CC  gdCompProc         NIL
 800050D0  gdFlags            BC01
 800050D2  gdPMap             00002110 -> 8000510C ->
 8000510C    baseAddr         50F40000 is a bad pointer


 80005110    rowBytes         8400
 80005112    bounds           #0 #0 #480 #640
```

This is the start of the device list. As stated above, the address of this list can be found in the system variable “devicelist”. This points to a long that contains the address of this list.

If you look down this list, you’ll see a variable called “gdPMap” - this is a pointer to a pointer that holds the start of VRAM. The offset to this long pointer is 22, so by pointing to the device list with an address register, then moving 22(address register), we get the pointer to VRAM. The code goes like this:

```
DEVICELIST: EQU   $08A8 THE ADDRESS OF DEVICELIST
GET_VRAM_ADDR:    MOVE.L      DEVICELIST,A0
            MOVE.L      (A0),A0     POINT TO DEVICELIST
            MOVE.L      A0,A1 COPY ADDRESS OF DEVICE LIST
                MOVE.L              22(A1),A1         CONTENTS OF GDPMAP
                MOVE.L              (A1),A1           VRAM BASE ADDRESS IN A1
                MOVE.L              (A1),A1           VRAM ABSOLUTE ADDRESS NOW IN A1
```

Using this code gives us the address of the device list in a0, and the address of the VRAM in a1.

Whenever we want to set a pixel in VRAM, all we need do is calculate the VRAM address, as above, then add the base address of the VRAM to get the absolute address of the pixel.

Lets have a look at a real example. Example 7 in the guide examples folder contains the following program:

```
*****************************************************************************
*EXAMPLE 7 - SETTING PIXELS                                                 *
*AUTHOR    - STUART BALL JULY 94                                            *
*SCREEN MUST BE SET TO 256 COLOUR MODE FOR THIS EXAMPLE                     *
*****************************************************************************
SCREENROW:      EQU                 $0106             WIDTH OF EACH SCREEN LINE
DEVICELIST:     EQU                 $08A8             THE ADDRESS OF DEVICELIST

GET_VRAM_ADDR:
      MOVE.L      DEVICELIST,A0
      MOVE.L      (A0),A0     POINT TO DEVICELIST
      MOVE.L      A0,A1 COPY ADDRESS OF DEVICE LIST
      MOVE.L      22(A1),A1   CONTENTS OF GDPMAP
      MOVE.L      (A1),A1     BASE ADDRESS IN A1
      MOVE.L      (A1),A1     VRAM ABSOLUTE ADDRESS IN A1
**SET THE PIXEL AT 300,20 TO COLOUR 3
      MOVE.W      #500,D0     X COORDINATE OF PIXEL
      MOVE.W      #200,D1     Y COORDINATE
      MOVE.W      #20,D2      COLOUR
      BSR   PLOT  PLOT THIS PIXEL
      RTS
*****************end of program*************************************************

**PLOT IS A SUBROUTINE THAT'S PLOTS A PIXEL ON THE SCREEN IN THE COLOUR HELD IN D2
**NEEDS D0.W = X COORDINATE
**      D1.W = Y COORDINATE
**      D2.B = THE COLOUR
**      A1   = VRAM BASE ADDRESS


**ALL REGISTERS REMAIN UNAFFECTED
PLOT: MOVEM.L     D0-D3/A1,-(SP)    SAVE THE REGISTERS WE USE
      MOVE.W      SCREENROW,D3      GET PHYSICAL LENGTH OF EACH SCREEN LINE
```

```
MULS  D3,D1 Y COORDINATE TIMES THE LENGTH OF EACH LINE
ADD.L D0,D1 VRAM ADDRESS OF THIS PIXEL
ADD.L D1,A1 ABSOLUTE ADDRESS OF THIS PIXEL
MOVE.B      D2,(A1)     SET THE PIXEL TO THE COLOUR IN D2
MOVEM.L     (SP)+,D0-D3/A1    RESTORE THE REGS TO HOW THEY WERE
RTS         END OF SUBROUTINE PLOT
```

In this example, we use a subroutine called “plot” to plot a pixel on the screen, with the colour in d2. It follows the principles detailed above to calculate the VRAM address of the pixel and then moves D2 as a byte into this address.

This example sets the pixel at 500,200 to the colour 20, which on my machine, in the default set up is a pinky colour.

If you run it in Fantasm, by telling Fantasm to assemble to Mac ram, you should see a single pink spot in the middle right of the assembler window when the program is run - not very impressive I know, but it illustrates a very important subroutine - once you know how to set a pixel to a colour, its easy to get a pixels colour. This basic pixel plot routine can be used for drawing lines, circles, any shape you can think of in fact, with a little imagination.

The other plus is that the plot routine can be speeded up quite a bit by getting rid of the MULU instruction, and replacing it by a shift if “screenrow” is a multiple of 2 - 1024 is 2 to the power of 10, so the muls can be replaced by a shift instruction, which is faster.

Now we know how to plot a pixel, how about drawing a line?
Because the “plot” is a subroutine, and it saves the registers it uses, it can be called many times, to plot lots of pixels. If you imagine a line on a sheet of paper, it is just a line - one continuous sweep of the pen. On the Mac's screen however, a line is a series of pixels - the line is made up of many pixels, giving the impression of a continuous line.

How can the plot routine be speeded up? Well the biggest calculation involved is finding the address of the pixel in VRAM. We take the y coordinate and multiply it by the physical length of each line, then add the VRAM base address to it. What would be faster would be to calculate the physical VRAM address of every line on the screen in the init routine,and store these addresses in a table. All we have to do to get the vram address of the y coordinate is read it from the table, where the y coordinate is the index into the table. Give it a try.

While we discuss the next section, have a think about using the plot routine to draw a horizontal line across the middle of the screen.

**Goodbye windows.....**

The time has come to reduce the Mac's screen to its barest bones!
In a game, windows, menus and the system icons can be distracting to the player. We want the full screen for our playing area. How can we clear the screen of everything?

Now that we have total control over the VRAM, its just a matter of clearing the VRAM - simple as that.

However, as Mac's have all different sized screens, we must check the size of the screen - how many pixels vertically. This information can be gleaned from the device list
At offset 22 is gdPMap. This is a pointer to the “graphics device Pixel Map”. The first long is the physical address of the VRAM, the next word is the physical size of each line, then there are

4 words defining the screen rectangle. Unless you are driving a really complicated monitor, the first two coordinates (the top left of the screen) will be 0,0. The next two define the bottom right, in this case 480 y, 640x. This is where we can get the number of lines from - gdPMap+10 is the word we want.

We know the physical size of each line (in bytes) from the system variable "screenrow", or you could get it from the device list at gdPMap. Now its just a matter of clearing the number of lines.

In this example the physical length of each line is read from SCREEN_ROW, and the number of lines read from the device_list.

We can use two loops to clear the screen - the first clearing bytes until the physical screen length of the line has have been cleared, the other loop counting the lines, until all of them have been cleared.

```
*****************************************************************************
*EXAMPLE 8 - CLEARING THE SCREEN                                            *
*AUTHOR    - STUART BALL JULY 94                                            *
*SCREEN MUST BE SET TO 256 COLOUR MODE FOR THIS EXAMPLE                     *
*****************************************************************************
SCREENROW:      EQU         $0106                   WIDTH OF EACH SCREEN LINE
DEVICELIST:     EQU         $08A8                   THE ADDRESS OF DEVICELIST
gdPMap_offset:  EQU         22
EXAMPLE8:       MOVEM.L     A5-A6,-(SP)             SAVE MAC WORLD
                BSR         INIT                    INITIALISE OUR VARIABLES
                BSR         CLS                     CLEAR THE SCREEN
                MOVEM.L     (SP)+,A5-A6             RESTORE MAC WORLD
                RTS
************************END OF EXAMPLE 8*************************************

**INIT SETS UP OUR VARIABLES
INIT:           LEA         MY_VARS(PC),A6          A6 IS OUR VARIABLE POINTER
                MOVE.L      DEVICELIST,A0
                MOVE.L      (A0),A0                 POINT TO DEVICELIST
                MOVE.L      A0,A1                   COPY ADDRESS OF DEVICE LIST
                MOVE.L      gdPMap_offset(A1),A1    A1=POINTER TO GDPMAP
                MOVE.L      (A1),A1                 gdPMap IN A1
                MOVE.W      10(A1),NUMB_OF_LINES(A6)SAVE NUMBER OF LINES
                MOVE.L      (A1),VRAM_ADDR(A6)      SAVE VRAM ABSOLUTE ADDRESS
                RTS

**CLS CLEARS THE MAC SCREEN TOTALLY
**PRESERVES ALL REGISTERS
CLS:            MOVEM.L     D0-D2/A1,-(SP)          SAVE REGS

                CLR.L       D0
                CLR.L       D1
                CLR.L       D2

                MOVE.L      VRAM_ADDR(A6),A1        VRAM ADDRESS IN A1


                MOVE.W      NUMB_OF_LINES(A6),D0    NUMBER OF LINES ON SCREEN
                SUBQ.W      #1,D0                   DBcc ALWAYS LOOPS ONE MORE THAN THE COUNT
                MOVE.W      SCREENROW,D1            WIDTH IN BYTES OF EACH LINE
```

```
                LSR.W       #2,D1                   DIVIDE BY FOUR AS WERE CLEARING LONGS
                SUBQ.W      #1,D1                   DBRA ALWAYS LOOPS 1 MORE THAN THE COUNT!
                MOVE.W      D1,D2                   SAVE SIZE OF LINE(IN BYTES)

NEXT_LINE:
CLEAR_LINE:     CLR.L       (A1)+                   CLEAR THIS BYTE OF VRAM AND POINT TO NEXT
                DBRA        D1,CLEAR_LINE           CLEAR THIS LINE
                MOVE.W      D2,D1                   RESET BYTES/LINE COUNTER
                DBRA        D0,NEXT_LINE            CLEAR NEXT LINE

                MOVEM.L     (SP)+,D0-D2/A1          RESTORE REGS
                RTS

*********************************VARIABLES FOLLOW*********************************
MY_VARS:        DS.B        20                      SPACE FOR OUR VARIABLES

*****************************OFFSETS INTO VARIABLES*****************************
NUMB_OF_LINES:  EQU         0                       .W NUMBER OF PHYSICAL LINES ON SCREEN
VRAM_ADDR:      EQU         2                       .L ABSOLUTE ADDRESS OF VRAM
```

This example is a little more complicated as its been written in position independent code, and saves a5 and a6 before running, and restores them when its finished. The main program calls two subroutines. The first, INIT, sets up the variables we need - NUMB_OF_LINES is the physical number of lines on the screen, and VRAM_ADDR is the physical address of the VRAM.

The subroutine CLS clears the screen to colour 0 using the method we described above. Lets look at it in more detail.

```
CLS:            MOVEM.L     D0-D2/A1,-(SP)          SAVE REGS

                CLR.L       D0
                CLR.L       D1
                CLR.L       D2
```

These first four lines save the registers we're going to use, and then initialises d0 to d2 by clearing them.

```
                MOVE.L      VRAM_ADDR(A6),A1        VRAM ADDRESS IN A1
                MOVE.W      NUMB_OF_LINES(A6),D0    NUMBER OF LINES ON SCREEN
                SUBQ.W      #1,D0                   DBcc ALWAYS LOOPS ONE MORE THAN THE COUNT
                MOVE.W      SCREENROW,D1            WIDTH IN BYTES OF EACH LINE INTO D1
                LSR.W       #2,D1                   DIVIDE BY FOUR AS WERE CLEARING LONGS
                SUBQ.W      #1,D1                   DBRA ALWAYS LOOPS 1 MORE THAN THE COUNT!
                MOVE.W      D1,D2                   SAVE SIZE OF LINE(IN LONGS)
```

The first thing we do here is get the VRAM physical address into a1, and the number of lines into d0. We have to subtract 1 from the number of lines, because we'll be using the decrement and branch instruction for the loops. The DBcc instruction will always loop one more time than the count, so if we wanted to branch around a loop 10 times, 9 would have to be loaded into the controlling data register - be wary of this.

Next we get the physical number of bytes on each line with the move.w screenrow,d1 instruction. This moves the number of bytes in each screen line into d1. D1 is used as a counter to count the bytes on each line as they are cleared, however....

Clearing the screen byte by byte is quite a slow way to do it - in 256 colour mode on a 14 inch

screen there are over 490,000 bytes to be cleared. Instead we clear longs, which reduces the number of clears by a factor of 4, as a long is 4 bytes. If we are clearing 4 bytes in one go, then the number of time we have to clear 4 bytes is reduced by a factor of 4. Hence the LSR.W instruction. Shifting a number right by 1 is the same as dividing it by 2, so if we shift it right twice, we divide it by 4 - this is far quicker than using the divide instruction. Similarly, shifting a number to the left multiplies it by 2.

We save this count in d2, so we can quickly reload the line counter (d1) after we've cleared a line.

```
NEXT_LINE:
CLEAR_LINE:     CLR.L       (A1)+                   CLEAR THIS BYTE OF VRAM AND POINT TO NEXT
                DBRA        D1,CLEAR_LINE           CLEAR THIS LINE
                MOVE.W      D2,D1                   RESET BYTES/LINE COUNTER
                DBRA        D0,NEXT_LINE            CLEAR NEXT LINE
```

This is the heart of the cls routine. There are two loops here. One "nested" inside the other. CLR.L (A1)+ clears the long word at a1 (VRAM), then adds 4 to a1 to point to the next long word of VRAM - this simultaneously clears four bytes, or pixels at once, and automatically increments a1 to point to the next four pixels.

The next line, DBRA D1,CLEAR_LINE decrements d1 by 1, and if d1 isn't zero, branches to clear_line, where another long word of VRAM will be cleared. These two instructions are the inner loop of the cls routine. We carry on branching back to clear_line until one complete line of the screen has been cleared.

The next line move.w d2,d1 resets the byte counter. Then dbra d0,next_line decrements the line counter (d0), and if not zero branches to next line, which is the clr.l (a1)+ instruction.

This example really shows how compact and fast machine code can be - the instructions for these two loops fit into less than 20 bytes, yet clears the whole screen quicker than anything else possibly could!

At the moment the routine can only clear the screen to all zeros - can you modify it to clear the screen to any colour you want?

Instead of clearing the long word of VRAM, why not MOVE a long word of data?

```
Replace the line CLR.L      (A1)+
with             MOVE.L     #$01010101,(A1)+
```

This will set the four bytes to colour 1 - you could use any colour you wanted - for example

```
                 MOVE.L     #$FFFFFFFF,(A1)+
```

clears the screen to colour 255 - whatever it may be - experiment!

This routine can be speeded up quite a bit. At the moment we are using two loops - one to clear the line, the other to count the number of lines.

Note that we are clearing the whole physical length of the line - not just the visible portion of it.
There are two possible ways of speeding this routine up.
Firstly, why clear the bytes that arent visible? This is only useful if you are drawing graphics in the hidden area of VRAM, possibly to be scrolled onto the screen from the right.
If each line is only 640 bytes, why not just clear these visible bytes?

Secondly, if you do want to clear the whole VRAM, then why bother with screen lines?
Of course we need to know the physical length of the VRAM, but then, if we're clearing all the bytes on each line, its just one big block of memory. All we need to do is calculate the size of the VRAM, divide it by four and then starting at the start, clear that number of longs.
Even though this involves clearing the whole VRAM, it may be faster than just clearing the visible portion of the lines.

**Drawing lines.**

Before we got into clearing the screen, we were talking about pixels and how lines are made up from individual pixels.
Example 9 is a small program that clears the screen then draws a vertical line across the centre of the screen. It uses the CLS routine from example 8.

```
**DRAW_H_LINE DRAWS A HORIZONTAL LINE.
**NEEDS START X COORDINATE IN D0
**            Y COORDINATE IN D1
**            LINE COLOUR IN D2
**        END X COORDINATE IN D3
DRAW_H_LINE:    MOVE.L    VRAM_ADDR(A6),A1        FOR PLOT
DHL_LOOP:       BSR       PLOT                    PLOT THIS PIXEL
                ADD.W     #1,D0                   INC X COORDINATE
                CMP.W     D3,D0                   COMPARE WITH END COORDINATE
                BNE       DHL_LOOP                IF NOT THERE YET, DRAW NEXT PIXEL
                RTS
```

We copy the VRAM address into A1, as plot expects it.
Then we call plot to draw the first pixel at d0,d1 in d2's colour. We add 1 to the start coordinate in d0, compare it to the end x coordinate in d3, and branch if not equal to DHL_LOOP which calls plot again to plot the next pixel.

Example 10 expands on this them to draw a multi coloured box on the screen using a loop that sets the colour to 255, then calls draw_h_line. Then the colour is decremented by 1, and if not equal to zero, draw_h_line is called again. The routine is called draw_box.

I urge you to experiment with example 10. Note the use of short branches - this is faster than normal branches, as the offset is held in a byte within the instruction, rather than in a word following the instruction (this is called an extension word).

Example 10 can be speeded up quite a bit by redesigning the draw line routine. At the moment it plots each individual pixel, using plot, which sets a byte. Why not check if the pixel is on a long word boundary, that is the VRAM address is divisible by four, and setting 4 pixels simultaneously by moving a long. This would easily double the speed of the line drawing routine.

There are various ways of checking if the address is on a long word boundary.
If we take two addresses as an example - address 1 is $800050C0, and address 2 is $80007FFB
We take the VRAM address "and" and it with $00000003, if the result is zero, then its a long

word boundary.

$800050C0 anded with $00000003 is 0
$80007FFB anded with $00000003 is 3.
The code:

```
                move.l    a1,d0                   assume vram address is in a1
                andi.l    #3,d0                   and vram address with 3
                beq       long_boundary           branch to long_boundary if on a long boundary
```

I'll leave it as an exercise for the reader (they always say that don't they?)

The other exercise you should consider, is changing draw_h_line to draw vertical lines, and then, what about diagonal lines.....

If you are having trouble with any of this, register you're copy of Fantasm, then you can drop us a line - we'll gladly help. There's only so much we can include due to time constraints and trying to keep the size of the shareware distribution down.
This is only the tip of the iceberg - we couldn't possibly cover all you need to know to write a game - sprites, random numbers, sound, control, fast screen algorithms etc, however one last topic we will whet you're appetite with is scrolling.

**Scrolling.**

Now we have access to the Mac's screen directly, there's no end to the possibilities. As we can access the VRAM we can move the VRAM bytes about, in certain ways to produce scrolling. Suppose we want to scroll the middle section of the screen, down by 8 pixels. This is easily achieved using two address registers. We have to move the data, in such a way, so that it doesnt get destroyed by the move. To scroll the screen down, we have to start at the bottom and work our way up the screen, 1 line at a time. To scroll the screen down by 8 pixels, we have to set an address register to the end of the eighth line of the screen, and another to the end of the last line of the screen. We then copy longs from the first address register into the second address register, thus copying the line into the last line of the screen.
Then we set the first register to the end of the ninth line of the screen, and the second address register to the end of the last but one line of the screen and repeat the process for all the lines. it sounds more complicated than it is - have a look at example 11.

```
**SCROLL_DOWN SCROLLS THE SCREEN DOWN BY 8 PIXELS
**EXPECTS VRAM_ADDR,NUMB_OF_LINES AND LINE_LENGTH TO BE SET UP.
SCROLL_DOWN:    MOVEM.L      D2-D7/A0-A1,-(SP)      SAVE REGS

**FIRST GET THE DESTINATION ADDRESS IN A1
1               MOVE.L       VRAM_ADDR(A6),A0       START OF VRAM
2               MOVE.W       SCREENROW,D7           PHYSICAL LENGTH OF LINES
3               MOVE.W       NUMB_OF_LINES(A6),D6   LINES ON SCREEN
4               MULS         D7,D6                  D6 = LAST LINE ON SCREEN
5               CLR.L        D5                     FOR LINE_LENGTH
6               MOVE.W       LINE_LENGTH(A6),D5     divide d5 by 2 for 16 colour mode
7               ADD.L        D5,D6                  D6 = LAST PIXEL ON SCREEN (BOT RIGHT)
8               MOVE.L       D6,A1
9               ADD.L        A0,A1                  A1=PHYSICAL VRAM ADDR OF LAST PIXEL
10              MOVE.L       A1,A0                  ON SCREEN, AND COPY INTO A0

**NOW GET THE SOURCE ADDRESS IN A0
11              MOVE.L       D7,D3                  SCREENROW (ACTUAL WIDTH OF EACH LINE)


12              SUB.L        D5,D3                  DISTANCE IN BYTES FROM START OF LINE TO
                             THE END OF THE NEXT LINE
13              MULU         #DIST,D7               SCREENROW * SCROLL STEP
14              SUB.L        D7,A0                  A0 = LAST PIXEL OF SOURCE LINE

**NOW CALCULATE THE NUMBER OF LINES TO SCROLL
15              MOVE.W       NUMB_OF_LINES(A6),D2   CALCULATE NUMBER OF
16              EXT.L        D2                     LINES TO SCROLL
```

```
                SUB.L       #DIST+1,D2              DIST PLUS 1 FOR DBRA

**NOW SCROLL THE ENTIRE SCREEN
SCROLL:         MOVE.L      D5,D4                   LINE LENGTH IN BYTES
18              LSR.L       #2,D4                   DIVIDED BY 4 FOR LONGS
19              SUBQ.L      #1,D4
COPY_LINE:      MOVE.L      -(A0),-(A1)             MOVE THE SOURCE LINE TO DESTINATION
                DBRA        D4,COPY_LINE            UNTIL ALL LONGS HAVE BEEN MOVED

                SUB.L       D3,A0
                SUB.L       D3,A1                   POINT TO NEXT LINE
                DBRA        D2,SCROLL               NOW MOVE THIS LINE, UNTIL ALL ARE MOVED

                MOVEM.L     (SP)+,D2-D7/A0-A1       RESTORE REGS
                RTS
```

The subroutine scroll_down is the core of the program. It will scroll the whole screen down by the number of lines in DIST. If DIST is 1, then the screen will be scrolled down by 1 pixel. If DIST equals 8 then the screen will scroll down by 8 pixels.

Assume we have a 640 by 480 screen, and we want to scroll the screen down by one pixel. This means moving line 638 to 639, then 637 to 638, then 636 to 637 until we've moved line 0 to line 1. This is exactly the way example 11 works. First we find the address of the last pixel in line 638 - this is called the source address. Then we find the address of the last pixel of line 639 - this is the destination address. Then we move line 638 to 639 with the loop labelled copy_line. Both address registers are now pointing to the start of the lines. We want them pointing at the end of the next lines - ie source (a0) at line 637 and destination (a1) at 638. We could calculate the addresses using the same method as we did at the start of the subroutine, but because we know the physical length of each line, and the actual number of bytes in each line, if we subtract the number of bytes from the physical length, this gives us the distance in bytes, between the end of the line above, and the start of this line. This value is in d3, so all we need to do is subtract d3 from both the source and destination address registers, then move the line. We carry on doing this until all the lines have been moved (the count for the number of lines is in d2). The set up for the source and address registers is as follows:

The destination address is given by the formula
LINES * PL + LENGTH + VRAM

Where LINES is the physical number of lines displayed on screen
PL is the physical length of each line - i.e. 1024
LENGTH is the actual length of each screen line - i.e. 640 for 256 colour mode
VRAM is the base address of the VRAM

The source address is given by
The destination address - dist*PL

where dist is the number of pixels to scroll.

Lines 1 to 10 calculate the destination address A1
11 to 14 calculate the source address in A0

The reason for the calculations is to make the scroll routine easily modified for different colour modes. At line 6, where LENGTH is moved in d5, if d5 is now divided by 2, the routine will work in 16 colour mode. Divide by 4 for 4 colour mode, and 8 for black and white.
In the same way, if you want it to work in 16 bit colour, all you have to do is multiply d5 by 2.
As proof of the pudding, example 12 is modified to work in 16 colour mode - which is twice as fast, simply by dividing d5 by with a shift instruction.
Also note the use of the RS directives in example 11 and 12.

First try example 11 in 256 colour mode to prove it works, then try example 12 in 16 colour mode. Of course it would take a pretty powerful Mac to scroll the whole of a 256 colour screen fast enough for a game. Hence we have the 16 colour scroll in example 12. If this is still too slow, why not scroll only part of the screen - a simple modification - most scrolling games do this - part of the screen has a static picture, whilst the rest of the screen scrolls. To scroll the screen the other way - ie upwards you have to start at the top of the screen and work your way down. Left and right scrolling is the same principle, except instead of moving lines, you have to move columns.
Try it as an exercise - if you get stuck, and you've registered, a simple note will bring some code winging its way to you.

We've missed two important areas in scrolling.
The first is, as you can see if you run example 11 or 12, that as the screen scrolls, you have to print new graphics to be scrolled onto the screen. As the screen is scrolled down, the source line isn't destroyed - you need to print new graphics onto it. Ideally this should be done off screen, or during a time when the video chip isn't reading data out of VRAM, so the user doesnt see it. This is quite easily achieved on the Mac, using either "hidden" VRAM at the top, bottom and right of the screen, or by using the "VBL" (Vertical BLanking) interrupt, which we wont go into here, cause we haven't the space, the time, and you possibly haven't paid for this.

The second area we've missed is that of sprites, or graphics over layed onto the scrolling screen to give your game monsters, missiles, aliens or whatever! These again have to be done either off screen or by using the VBL, so the screen is scrolled and you sprites are over layed onto the screen *after* the scrolling has taken place, or believe me, it looks horrible!

**The end.**
Well, that's it folks, Its taken nearly as long to produce this beginners guide, as it did to write Fantasm. Hopefully you'll have learnt quite a lot by now, and be eager for more info, and there is a *LOT* more.

In this beginners series, we haven't covered sound, sprites, graphics (in any great detail - Lightsofts main work is in 3D graphics), program design in any great detail, or advanced 68000 instructions and hardware like MMU's and caches (beware the caches!) or any decent algorithms.

Its fair to say that a lot of programming, whether in machine code, or basic or C, is the

algorithms you employ, and being able to spot a quick speed up somewhere. As an example, in example 11 - the scroll routine, there's a very basic speed up that we've missed out deliberately. If you spotted it you are well on the way.
The offending code is lines 17 to 19 where for every line to be scrolled, we calculate the number of longs to be moved in D4. We should have calculated this prior to the loop, and stuffed it in a spare register, say d2, then all we'd have to do is move d2 to d4, instead of moving d5 to d4, dividing it by 2 then subtracting 1!

Try modifying it - it will yield a useful speed increase.

Its 0205 on Friday 15 July 94(yawn) - most of this text and examples have been written in the wee

hours after midnight over about four weeks from the middle of June. I apologise for any great errors or obvious omissions. I cannot guarantee the accuracy of anything in this text, but publish it in the belief that it is correct.

Here is my list of essential Mac programming tools:
Fantasm (obviously)
Edit II from ESCware, which I use to write the machine code source files.
Macsbug by Apple - Macintosh aware debugger - vital!
Popchar by Günther Blascheck - essential for finding the Ascii code of a weird character.
Progcalc() by Bob Meyers - handy for checking arithmetic and logical operations whilst coding.
Resedit © Apple - Really the only way to edit a create resources.
Syser701 by Dr. Pete Corless and Apple - useful on line reference of all error codes.

**Useful tips and other bits.**

1. *NEVER* program when drunk - its not worth it honestly.

2. Always check your code for speed after its working. The way I personally program is scribble the algorithm down, code it, get it working, then optimise it for speed.

3. Always keep plenty of backups - hard disks are not fallible, specially when programming in machine code - one little slip is all it takes. I have personally screwed up my LCII so badly that I was forced to pull the battery from the motherboard and leave it for 2 hours before it would boot up again!

4. The important one - don't get disheartened because something will not work - *NOTHING* is impossible. On one of our programs, it took six months to get one subroutine working correctly. A sense of humour, headphones and some good music in the background are essential.

5. Don't try programming whilst the sun is up! Its true, its impossible to produce a really wicked piece of code unless the sun has gone down, and you have to get up to go to work in four hours.

6. The realisation that there are better programmers than you is very important to the learning process. They may come across as brash, confident and cocky, but in my personal experience its because they are right about what they are saying. Of course, if it turns out they are all mouth and no code.....

7. Sometimes, you may have to resort to some quite complicated mathematics. Don't be put off by it - you needn't understand the maths to be able to use it! If you need an equation(s), just use it, don't worry about the theory behind it. Libraries are excellent sources of all kinds of information, specially real world mathematics.

8. Share your code - that way everybody gets to learn. Of course if you have a specialised algorithm that increases the speed of Mac scrolling by a factor of 4 - guard it with your life - its probably worth a lot of money!

**Chow....**
Ok, that's it, I'd love to carry on with this course - random numbers, graphics, sound - the Mac is a powerful computer, that has not been exploited on the games front yet - I believe there is a big opportunity for some enterprising programmer to write a truly good game for the Mac that doesnt involve 500 megabytes of movie on CD! Take Elite II - 800K and its huge!
That's why Fantasm was produced - there isn't anything like it for the Mac. If there is interest then Fantasm will be expanded, along with a very fast debugger, which is about half complete - one thing you can't give Macsbug credit for is its speed.

I'd like to thank Cath my wife, and appreciation for the following:

Timothy Ungless for testing.

Pink Floyd, Tasmin Archer, Chris Rea, Dave Letterman, Startrek TNG, The X files.

This document was produced with Clarisworks V1.0BV3

Stuart Ball 130894

Revision 1.01