•Beginners Mac Assembly•

 Chapter 3 

Instruction classes.

The instructions fall into specific classes:

- •Integer Arithmetic
- •Data movement
- •Logical operations
- •Shifts and rotates
- •Bit manipulations
- •Program control
- •Binary Coded Decimal
- •System control

The arithmetic instructions.

ADD - Adds two operands leaving the result in the destination operand. One of the operands must be a data register.

ADDA - Adds an operand to an address register.

ADDI - Adds an immediate value (real number) to an operand.

ADDQ - Adds a number in the range 0-8 to an operand.

ADDX - Allows the use of multiprecision additions - numbers of any length can be added.

CLR - Clears an operand

CMP - Compares two operands and sets the condition code flags accordingly.

CMPA - Compares an operand to an address register and sets the condition code flags.

CMPI - Compares a real number to an operand and sets the condition flags.

CMPM - Compares the contents of two memory locations using the post increment addressing mode.

DIVS - Divides the destination operand by the source operand using signed arithmetic.

DIVU - Divides the destination operand by the source operand using unsigned arithemtic.

EXT - Sign extend a byte to a word, or a word to a long.

MULS - Multiplies a destination operand by the source operand using signed arithmetic.

MULU - Multiplies the destination operand by the source operand using unsigned arithmetic.

NEG - Negate a number - make positive numbers negative and vice versa.

SUB - Subtract the source operand from the destination operand leaving the result in the destination operand.

SUBA - Used to subtract the source operand from an address register.

SUBI - Subtracts a real number from the destination operand.

SUBQ - Subtracts a number in the range 0-8 from a destination operand.

SUBX - Multiprecision subtract.

TAS - Test And Set - Test a byte and sets the high order bit.

TST - Tests an operand - compares it to zero.

Data movement instructions.

EXG - Exchange the contents of two registers.

LEA - Load Effective Address. Calculate a memory address and store it in an address register.

LINK - Allocates a stack frame.

MOVE - Move the source operand into the destination operand.

MOVEM- Transfers multiple register to and from memory.

MOVEP- Transfers data to/from an 8 bit peripheral.

MOVEQ- Loads a data register with a number in the range of +- 128.

PEA - Same as LEA, but pushes the address onto the stack.

SWAP - Swaps the words of a data register. The high word becomes the low and the low the high.

UNLK - Deallocates a stack frame.

Logical operations.

AND - Performs an AND operation with the operands.

ANDI - Perform and AND operation with a real number and an operand.

OR - Same as and except an OR operation is performed.

ORI - OR a real number with an operand.

EOR/EORI - Exclusive or.

NOT - Invert an operand.

Shifts and rotates.

ASL and ASR - Arithmetic shift left or right.

LSL and LSR - Logical shift left or right.

ROL and ROR - Rotate left and right.

ROXL and ROXR - Rotate with the carry bit left and right.

Bit manipulation.

BTST - tests a single bit.

BSET - set a single bit.

BCLR - clear a single bit.

BCHG - change a single bit.

Program control.

Bcc - Branch if a condition (cc) is met.

DBcc - Decrement and branch if a condition is met.

Scc - Set if the condition is met.

BSR - Branch to subroutine

JSR - Jump to subroutine

RTS - return from subroutine.

JMP - Jump to an absolute memory location.

RTR - Restores the program counter and condition codes from the stack.

System control instructions.

Most of these are privileged instructions - meaning the processor must be in supervisor mode to use them.

MOVE USP - Move an operand to the user stack pointer.

RESET - Reset external peripherals.

RTE - Return from an exception.

STOP - Stop processing until an exception occurs.

CHK - Check an operand against boundaries - used to prevent serious software errors.

TRAPV - Trap on overflow - used to prevent serious software errors

TRAP - 16 instructions that provide a method for a user program to call a supervisor mode program.

We are not going to detail every instruction. The information is easily available from Motorola, or any computer book shop. What we will do is explain some of the more obscure instructions starting with the logical operations.

And, Or and Exclusive Or.

Logical operations are very simple. An "and" operation simply says if both A and B are equal to a logical 1, then set the result. Logical operations work at a bit level, that is, for you to decide what the outcome of a logical operation will be, you have to understand the data.

Suppose we "and" 1 and 9, with the instruction ANDI.W #1,d0, and d0 contained 9. If we look at the number in binary, 1 is 0001 and 9 is 1001. When these two numbers are anded, the processor looks at the numbers like this:

```
      3210 <- Bit number
      0001 <- 1 in binary
      1001 <- 9 in binary.
```

First it will look at both bit 3's. It says I have a 1 and a 0, so I don't have two 1's. Therefore the result is zero. Then it looks at bit 2's, which are both zero, so the result is zero. Bits 1 are both zero, so the result is zero. Bits 0 are both 1. It says if I have a 1 "and" a 1 then the result is 1, so bit zero result is 1.

The result of 9 and 1 is 1.
The and operation can be summarised by saying "only if both bits are set will the result be set"

Examine the following examples, to see if you can spot the main use of the "and" instruction.

What is the result of $FA anded with $0F?
$FA = 11111010
$0F = 00001111
AND = 00001010 = $0A

What is the result of $F1 anded with $0F?
$F1 = 11110001
$0F = 00001111
AND = 00000001 = $01

What is the result of $1220 anded with $00FF
$1220 = 0001001000100000
$00FF = 0000000011111111
AND   = 0000000000100000 = $0020

The and instruction is mostly used to mask off wanted data in a register. By setting bits in the word or byte you want to keep, the other bits will be discarded.
For example if you had a routine that returns the ASCII value of a key pressed on the keyboard, and it returned the key in d0. The key can be specified in a byte, but there may be data from earlier processing in the upper three bytes of d0 - so to ensure you don't create errors further in the program, the byte can be masked off with the and instruction as follows:

```
      ANDI.L      #$FF,D0
```

Irrespective of how much garbage is in the upper 24 bits of d0, after this instruction all that will be left in d0 is the byte defining the key press, because the size of the instruction was

long, so all 32 bits of the register will have been anded.

The OR instruction works like this:
If either or both of the bits are 1, then the result bit is a 1. The other way of looking at it is "If both bits are a zero then the result is a zero, otherwise its a 1".
Example - OR 1 with 2
1 = 0001
2 = 0010
or= 0011 = decimal 3

The Exclusive OR instruction works like:
"If one bit is a 1 and one bit is 0 then the result is 1, otherwise the result is 0".

Example - EOR 1 with 15
1 = 0001
15 = 1111
EOR= 1110 = decimal 14

EOR 1 with 0
1 = 0001
0 = 0000
EOR= 0001, so the result is 1.
If we EOR the result with 1 we get a 0. This is a neat way of toggling a bit, every time a loop executes.

Initially the bit is set to 1. Each time round the loop, the bit is EOR'd with 1. Every time the loop executes. if the bit is a 1 its set to a 0, and if its a 0 its set to a 1.
The instruction EOR #1,my_bit will toggle bit zero of my_bit every time it executes.
What's the use of this? Suppose you want to flash something, say an alien spaceship on the screen between red and yellow. You test my_bit, if its a one you set the spaceships colour to red, if its a zero, you set the colour to yellow.

Program flow control.

If you remember back to the previous examples, what happened when you ran them. Well it was all over rather quickly wasn't it. If you look closely at example 3 running, you should see a window, with the number 60 in the top left corner of the window. Then the window was cleared and the program ended. All in a matter of half a second. It would be nice if your Mac waited for you to examine the results of the program before clearing the window from the screen. What you may want is to be able to tell the computer when it should clear the window?
Enter stage left example 4.

```
*******************************************************
*FILENAME:      EXAMPLE4.S                              *
*DESCRIPTION:  PROGRAM TO ADD 3 NUMBERS TOGETHER    *
*       AND PRINT THE RESULT IN A WINDOW.              *
*       WRITTEN IN POSITION INDEPENDENT CODE           *
*******************************************************

EXAMPLE4:       LEA             EX4_VARS(PC),A6
                BSR             INIT_CURSOR                     *part of io_lib.s
                LEA             WINDOW_TITLE(PC),A0  *Open a window with this title
                BSR             OPEN_WINDOW                     *Draw window (part of io_lib.s)
                MOVE.L A0,WINDOW_HANDLE(A6)         *OPEN_WINDOW RETURNS A HANDLE TO THE WINDOW

                MOVE.L #10,D0                   *ADD 10 AND 20 AND 30
                ADD.L           #20,D0
```

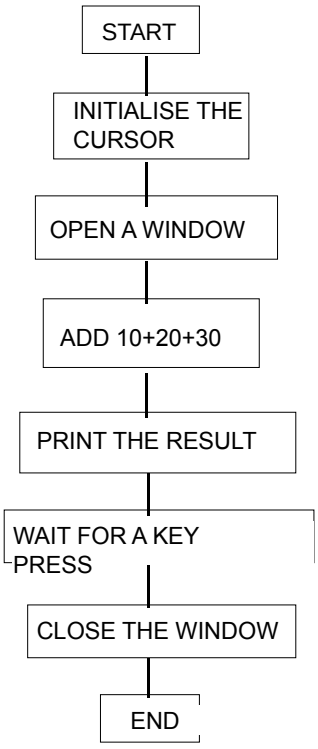
```
                ADD.L           #30,D0
                BSR             PRINTNUM                *part of io_lib.s - PRINTS D0

                BSR             WAIT_KEY                *WAIT FOR A KEY TO BE PRESSED
                MOVE.L WINDOW_HANDLE(A6),A0     *GET THE HANDLE FOR THE WINDOW
                BSR             CLOSE_WINDOW            *CLOSE THE WINDOW
                RTS
                INCLUDE         IO_LIB.S
****VARIABLES
WINDOW_HANDLE:      EQU     0       *WINDOW HANDLE .L
EX4_VARS:           DS.B    4       *SPACE FOR 1 LONG
WINDOW_TITLE:DC.B   "EXAMPLE 4",0
                EVEN
**********************END OF EXAMPLE4********************
```

Note the change - after the number is printed, we call a subroutine that waits for the user to press a key. The subroutine “Wait_key” actually returns the key pressed in D0 as the lower byte - that is bits 0-7. Armed with this knowledge its time to find out about comparing data and controlling program flow depending on the outcome of the comparison.

Here’s a tough question - how can we modify example 4 so that it quits only when the “Q” key is pressed on the keyboard? Any ideas - have a go a writing the program, but don’t worry if you don’t have a clue. Remember, “WAIT_KEY” returns the key pressed in the lower byte of d0.

A note on program design - the interlude.



This is the “flowchart” of example 4. A flow chart is a simple way of visually describing the “flow” of a program. It’s written in plain English, with each individual step of the program being enclosed in a box. The boxes are not mandatory, you don’t need to use them.

Its what's in the boxes that is important. Irrespective of the actual computer language you are using, by describing the program in plain English first, you have programmed it.

After describing the program in this way, you should try to play computer. This means mentally running through the program, analysing what the computer is doing at this step, and then verifying it is correct. This is very important. Normally, when you have a specific problem to code, you need an "algorithm" that explains how a problem can be solved by breaking it into smaller, easier steps. The algorithm may already be known, in which case its just a matter of converting the algorithm to something the machine can understand, or you may have to develop the algorithm yourself, which is by far the hardest part.

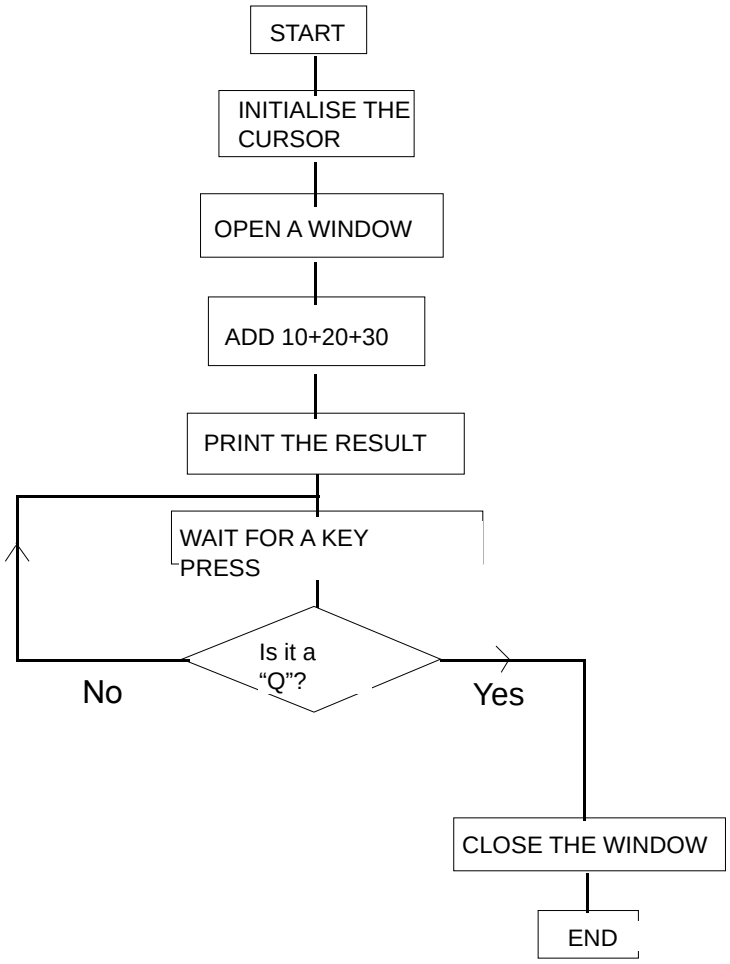
Once the algorithm is described in readable English, either on a nice flow chart, or the back of a cornflake packet, you can then convert it into a language the computer can understand. I cannot stress how important it is to design the algorithm(s) before you even walk into the same room as the computer.

Sometimes the solution to a problem can be blindingly obvious, in which case you may think you can just sit down and program it (we do it all the time), but then somebody, someday, will type something in, or click the mouse, in just the wrong place, and your program will crash because you didn't analyse the problem sufficiently before coding it.

You will find, that as you program more, and become more experienced at it, the amount of scribbling you have to do gets less and less - personally I use these little yellow post-it pads. I can scribble the program down on a couple of these, then stick them over the desk where they don't get in the way of mice and things.

Back to the practical side of things....

We want to modify example 4 to quit only when the "Q" key is pressed. Examine this flow chart of the modified program.



Now, we have a new box - a diamond. Diamonds are used to indicate decisions. In this case its "was the key press a Q"? If it wasn't, then the line goes from the decision box, back to get a key press. If the letter was a "Q", then the line goes to "Close the window" and "End".
We need a way to compare the contents of d0 with the letter "Q". As was mentioned earlier characters are represented in the computer by numbers. A byte can represent the whole character set, so its logical to assume that when I say "wait_key" returns the key in d0, what I mean is that wait_key returns a number in the lower byte of d0 that represents the character of the key pressed.
The numbers are defined by the ASCII code. Any useful computer book will have the ASCII character set in one of its appendices. They wont be reproduced here, but in one of the indices. If you look up the code of the letter "Q", you'll find it is 81, or $51.

So we check d0 after we have "called" get_key. If the number isn't $51 (its not the letter Q) then we BRANCH back to the instruction that calls wait_key. If the number in D0 is $51, then we end the program.

The idea of an assembler is to make life as easy as possible for the programmer. To this end, we don't have to compare d0 to $51, we just use the letter Q, but we enclose it in speech marks - "Q". When the assembler sees this, it looks up the ASCII code for the letter Q and substitutes it. The program "Example5.S" embodies these changes.

```
*************************************************************
*FILENAME:       EXAMPLE5.S                                  *
*DESCRIPTION:  PROGRAM TO ADD 3 NUMBERS TOGETHER    *
*          AND PRINT THE RESULT IN A WINDOW.                 *
*          The program then waits for the Q key              *
*          WRITTEN IN POSITION INDEPENDENT CODE              *
*************************************************************

EXAMPLE5:       LEA             EX5_VARS(PC),A6
                BSR             INIT_CURSOR    *part of io_lib.s
                LEA             WINDOW_TITLE(PC),A0
                BSR             OPEN_WINDOW *PART OF IO_LIB.S
                MOVE.L A0,WINDOW_HANDLE(A6)         *OPEN_WINDOW RETURNS A HANDLE TO THE WINDOW

                MOVE.L #10,D0  *ADD 10 AND 20 AND 30
                ADD.L           #20,D0
                ADD.L           #30,D0

                BSR             PRINTNUM        *part of io_lib.s - PRINTS D0
wait_for_q:     BSR             WAIT_KEY        *WAIT FOR A KEY TO BE PRESSED
                cmpi.b   #"Q",d0
                bne             wait_for_q
                MOVE.L WINDOW_HANDLE(A6),A0         *GET THE HANDLE FOR THE WINDOW
                BSR             CLOSE_WINDOW        *CLOSE THE WINDOW
                RTS
                INCLUDE         IO_LIB.S
****VARIABLES
WINDOW_HANDLE:      EQU     0       *WINDOW HANDLE .L
EX5_VARS:           DS.B    4       *SPACE FOR 1 LONG
WINDOW_TITLE:DC.B   "EXAMPLE 5",0
                EVEN
************************END OF EXAMPLE5*********************
```

Because we need to branch to the line that calls wait_key, we need a label. The label "wait_for_q" should be as descriptive as possible. You have up to 32 characters to use, so there's no excuse for a labels such as WFQ, except laziness (I admit it!). Note the colon after the label, when its defined, but no colon when the label is being "called".

Bcc

The instruction that actually decides whether to branch back to wait_for_q, or to carry on with the rest of the program is BEQ - branch if equal. This tests the condition code register. In this case its looking at the zero flag. It says "Branch if the zero flag is set". How is the zero flag affected? The instruction before the branch is a compare. In this case it is comparing the ASCII code for the letter "Q", with the lower byte in d0. How does it compare them? It subtracts the source operand from the destination operand. It discards the result, but makes a note in the flags. Thus is d0.b contains $51 and we compare it with $51, the result will be zero, so the zero flag will be set.

There is a limitation in this example. The program will only quit if the key pressed is an upper case Q - nothing will happen for a lower case q. Can you modify example 5 to quit on either an upper case or lower case q?

If the only test we could perform was "if zero", things would be pretty difficult. Fortunately there are thirteen other tests we can make. Here are all the possible branch instructions:

BCC - branch if the carry bit is clear (a zero)
BCS - branch if the carry bit is set (a one)
BEQ - branch if equal.
BGE - Branch if greater than or equal.
BGT - Branch if greater than.
BHI - Branch if higher than. This is the same as BGT except its used on unsigned numbers.
BLE - Branch if less than or equal.
BLS - Branch if lower than or the same. The same as BLE except used for unsigned numbers.
BLT - Branch if less than.
BMI - Branch if minus.
BNE - Branch if not equal.
BPL - Branch if plus.
BVC - Branch if the V bit is clear. Means branch if there was no overflow.
BVS - Branch if the V bit is set - there was overflow.
BRA - Always branch.

There are two other instructions that use these flags in the same way.

Scc

The first is the SET instruction. This is used to set a byte IF a certain condition is met. For example, if you compared a word in memory to a data register, and if they were different you wanted to set a flag to let another part of the program know this fact the following code could be used:

```
      CMP.W       (a0),d0
      SNE         flag(a6)
```

The CMP compares the word at the address pointed to by a0 with the word in the lower 16 bits of d0, and sets the condition code bits accordingly.
The next instruction, SNE, uses these bits to determine the result of the compare. If the word in memory didn't match the word in d0, then the set instruction will set the byte at flag(a6) to all 1's. If we wanted to set the flag IF the words were equal, we would use the SEQ form of the instruction.

DBcc

The other instruction that uses these bits is the decrement and branch instruction. This takes the form of

```
            DBcc  data reg,address
```

This instruction is used for looping. That is, repeating a piece of code many times, until either the condition is met, or a maximum count is exceeded. The count is contained in the lower word of a data register. This means the maximum number of times a loop can be repeated using this instruction is 65536. Every time this instruction is executed, the data register will be decremented by 1. When the count gets to -1, the loop will stop, and execution will continue with the next instruction. The other condition where the loop may stop, is if the condition is true.

```
            moveq       #8,d0
            moveq       #10,d1
loop:       subq.l      #1,d0
            dbpl        d1,loop
```

In the above code, d0 is loaded with 8, and d1 is loaded with 10. Then in the loop, 1 is subtracted from d0. The DBPL instruction examines the CC register. If the result of the subtraction of 1 from d0 is positive, then d1 will be decremented by 1. If d1 is greater than -1

then the instruction will branch to loop.

This instruction therefore allows us to loop around a piece of code until either the condition is met, or the count runs out.

The simplest use for this instruction is in creating a delay:

```
            move.l      #1000,d0
delay:      dbra        d0,delay
```

We move 1000 into d0. Then we decrement and branch to itself until d0 = -1. The outcome of this is the DBRA instruction in this piece of code will be executed 1001 times. Its important to note that making a loop with the DBcc instruction will cause the loop to execute 1 more time than the number you put in the register. If you wanted this loop to execute 1000 times, then d0 would have to be loaded with 999!

Because this instruction is a branch instruction, the offset of the label must fit into a signed word, that is must be in the range of + or - 32767 bytes. If its outside this range, the assembler will give an error.

Do it yourself.

Up until now, we've been writing very simple programs, but using subroutines to open windows,get key presses and print numbers. Now's the time to learn how to do it yourself.

As you've seen, machine code instructions only perform very simple operations - printing a number or drawing a window is obviously quite a complicated task. How is it done? One of two ways, either write it yourself, or use the in built software to do it for you.

Firstly we'll look at the built in software method.
Every single computer ever sold, whether it be a ZX81 or the latest Mac has what is called an operating system (OS). On the Mac, the OS is called system 7 (currently). The OS is provided by the manufacturer to provide a standard set of tools for doing things on that computer range. Things like windows, sound, talking to disk drives, the screen and most other peripherals. Generally speaking, the more powerful the computer, the bigger the operating system. The OS is built up from thousands of small subroutines, each one performing a different task. For example, there are routines to draw a line, or print a character, or read a file from disk - on a Mac running system 7, there are over 4000 of these routines that programmers can call, from within their programs. For example, our example programs call a subroutine "printnum". The printnumb subroutine calls system 7 to draw the characters that make up each number.

So, we have our program - maybe ten or eleven lines of code. This calls a subroutine that may be 40 lines of code. The subroutine calls a routine from system 7 that is literally thousands of lines of code, just to print a character on the screen. If we had those lines printed out, then our example program to Add 3 numbers and print the results on the screen would be a lot of paper, as it is, our program is 11 lines long.

How then, do we know what the OS routines are, and how do we access them?
Apple have published them. They're all detailed in books like "Inside Macintosh". However its unlikely that you'll ever need to know all 4000 of them. As a matter of fact, only 50 or 60 are needed to write most applications.

All the routines in system 7 are given names. Whether you stick with the names or not is entirely up to you. Lets have a look at some:

```
SYSBEEP:        EQU     $A9C8
NEWWINDOW:      EQU     $A913
DISPOSEWINDOW:          EQU     $A914
TEXTFONT:       EQU     $A887
TEXTSIZE:       EQU     $A88A
```

These are a couple I've dragged out of io_lib.s.
They are given labels, so we can use the names instead of the numbers. As everything in machine code they are just numbers. For example sysbeep equates to $A9C8. So?

Well have a look at this:

```
SYSBEEP:    EQU   $A9C8

START:      MOVE.W      #1,-(SP)
            DC.W        SYSBEEP
            RTS
```

Would you believe this makes your Mac go beep? Try it, load up fantasm, assemble example6.s into ram and run it.

Its an incredibly simple program, but illustrates a very powerful fact.
The operating system calls on a Mac are called "TRAPS". They take advantage of a 68000 feature that allow the user to define their own instructions. In this case, the instruction is "sysbeep". To use a trap you need to push any parameters it needs onto the stack, then just define the word for that trap as a constant in your program.

```
START:      MOVE.W      #1,-(SP)
```

This line pushes the parameter onto the stack. In this case, sysbeep needs 1 parameter - the duration of the beep. Parameters can be bytes, words or longs.

```
            DC.W  SYSBEEP
```

This line inserts $A9C8 into the program at this point, How does it know which number to insert? Because $A9C8 was EQUated to SYSBEEP in the first line of the program. When the processor gets to this piece of the code, it sees the instruction A9C8. The processor does not understand this instruction, so it gets an address from a special area of memory, called the "System vectors". These live at the bottom of memory. In this address is a pointer to the Mac operating system that tell the processor how to execute this instruction - the instruction is simulated by software. (For those in the know - the Mac OS is called via the line a vector).The processor jumps to the software, executes the instructions that make a beep sound, pops the parameter off the stack, and when finished returns to your program.

Nearly all of the OS calls operate like this. A few require the parameter(s) to be in registers and not on the stack.

Examine io_lib.s. It's just a source file, you can have a look either in Fantasm's editor, or any other editor. The best way to learn how to program is by examining other peoples code.
If you register you're copy of Fantasm you get the most useful traps explained in a document.

Resources.

Before we can continue, there is one more Mac specific topic we have to cover - resources.
A term synonymous with Macintosh, resources are a way of including all the bits and pieces that go to make up a program in one file. For example a program may need sounds, graphics, windows, text -

all classified as resources. On any other computer, these resources live in

separate files, in different folders - on a Mac they all live in one file, hidden from the user.

A Macintosh program on disk has two parts to it - the data fork and the resource fork. In the resource fork is the code and anything else the program needs to run. In the data fork can be anything, but is not normally used.

Resources can be thought of as folders. The individual resource items, be they sounds, graphics or whatever, are identified by an unique ID number, which is a signed word.
All an applications sounds will live in the "sound" resource, and each individual sound will have a number associated with it.
As the resources are a subpart of a file, if the file is open, then resources can be read. So the default resource file is the current application. Other resource files can be opened via OS calls, such as _OpenResFile ($a997), then individual resources loaded into memory as needed.

When a resource is loaded, with one of the _Getresource calls, a handle will be returned. You should save this handle, as its the only way of referencing the resource in future. When you want to use the resource, you have to pass its handle (normally on the stack), so the OS knows which resource you are referencing.

Normally resources are loaded into memory and locked with another OS call. They have to be either used almost immediately, or locked for future use. A resource is loaded into the system "temporary" memory as a default. Temporary memory means that if the system needs the space occupied by the resource, it will just take it. If however the resource is locked, then the system can't delete it. The Hlock call is $a029. This is a registered routine - it needs the handle in a0.
A useful list of system calls is provided on registration for Fantasm.

Apple have thoughtfully provided an editor for these resources, so they can be easily created and manipulated. Resedit is used to create these resources, which can then be used by your programs by calling the OS to load them into memory as and when you need them.

Fantasm will produce the application file, with the code as two resources. Any other resources you're program needs have to be put in the file with the Resedit. Once a file is created, and the resources have been inserted, Fantasm will not delete them on subsequent assemblies - it will only alter the code resources.

In the source folder of fantasm, is another folder called "resource demo". In here are a resedit sound file called "HI" and a source file called res_example.s

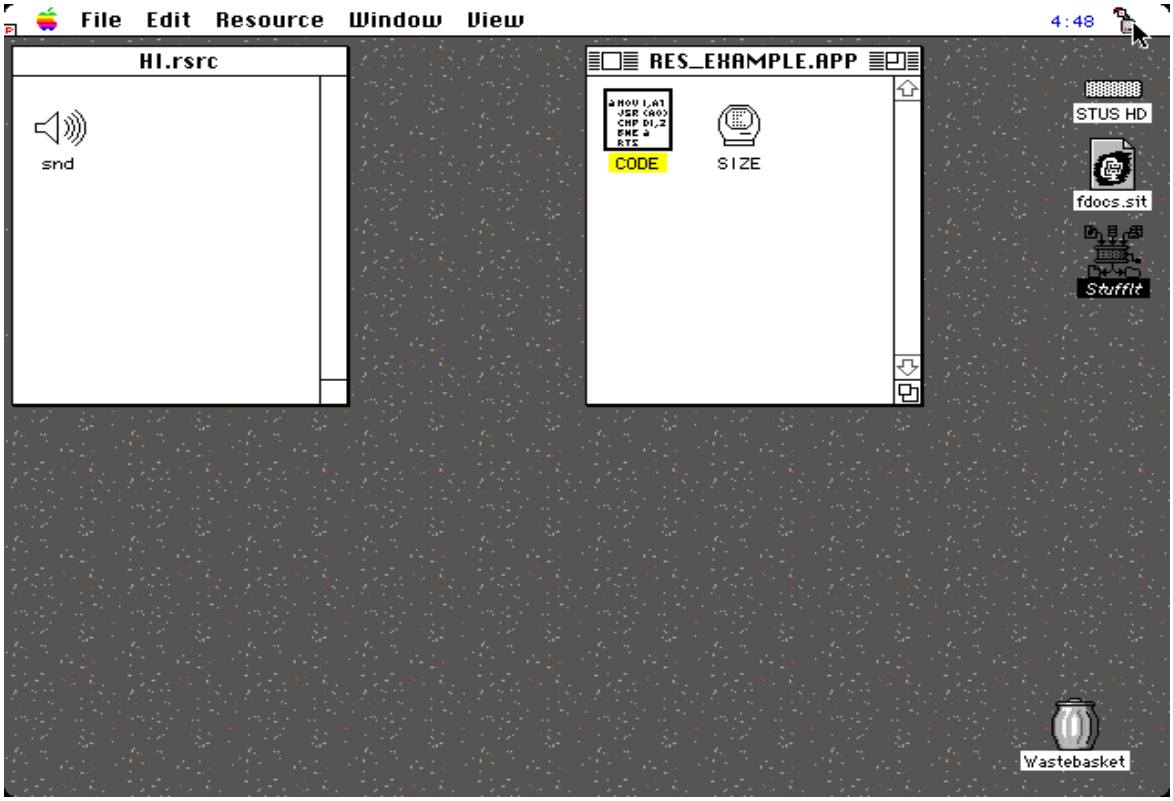
We'll run through an example of how to use resources in an application.

Load Fantasm, and click on "Output Mac Application" in the project menu. Now set the project name to "res_example.s". Assemble the file by clicking on assemble in the Fantasm menu, then quit Fantasm.

In the resource demo folder will be the application that Fantasm created from res_example.s.
If you double click it now, not a lot will happen. It will run, but wont make a sound because the resource it needs isn't in the application.

Now load Resedit. If you haven't got a copy, either contact Lightsoft, who can supply it for a handling charge, or it is commonly available from any bulletin board, or PD library.

Once Resedit has loaded open the file “HI.rsrc”. Now open “res_example.app”, so you should have two open windows as below:



Now click on the sound resource in HI.rsrc and copy it into res_example.app, using the copy and paste from the edit menu. Close both files and double click res_example.app from the desktop.

To summarise, we created the application with Fantasm as res_example.app. Then we used resedit to copy a resource from HI.rsrc into res_example.app.

```
*********************************************************************
*THIS EXAMPLE PLAYS SOUND RESOURCE NUMBER 128                       *
*********************************************************************
GET1RESOURCE:       EQU     $A81F
SNDPLAY:            EQU     $A805

START:      MOVEM.L     A5-A6,-(SP)          SAVE MAC WORLD      line 1

            CLR.L       -(SP)                SPACE FOR HANDLE    line 2
            MOVE.L #$736E6420,-(SP)          ASCII FOR "snd "line 3
            MOVE.W      #128,-(SP)           RESOURCE NUMBER     line 4
            DC.W        GET1RESOURCE         GET THE RESOURCE    line 5
            MOVE.L (SP)+,D0                  RETURNS A SOUND HANDLE line 6

            BSR         DO_SOUND             PLAY IT             line 7
            MOVEM.L     (SP)+,A5-A6          RESTORE REGISTERS   line 8
            RTS                              BYE                 line 9
```

```
***PLAY_SOUND PLAY THE SOUND WHO'S HANDLE IS IN D0
DO_SOUND:       CLR.W       -(SP)               SNDPLAY RETURNS A WORD      line 10
                CLR.L       -(SP)               CLEAR THE SOUND CHANNEL     line 11
                MOVE.L D0,-(SP)      HANDLE OF SOUND            line 12
                CLR.W       -(SP)               DO IT NOW                   line 13
                DC.W        SNDPLAY             PLAY SOUND                  line 14
                MOVE.W      (SP)+,D0            GET ERROR                   line 15
                RTS                             WE DON'T CHECK THE ERROR IN THIS EXAMPLE
```

The first two lines equate SYSTEM 7 traps to their names, thus making the program easier to read.

The program proper starts at the label START. The first line in any program that can run on its own - ie off disk as opposed to mac ram from Fantasm, should be to save A5. This is necessary because the Mac task switches - this means you can change programs with the click of the mouse. To be able to do this there are certain practices that must be observed. You'll see the full initialisation procedure later on. The bare minimum initialisation consists of saving A5 so it can be restored when your program quits. You really shouldn't need to do this, but in my experience, if you don't restore A5, it can lead to all sorts of nasty hiccups.

The next five lines are concerned with loading the sound resource into memory. Some traps return a value, either a byte, word or long on the stack. To do this, the trap needs the space reserved on the stack by your program - line 2 does this by clearing a long on the stack.
The trap we are using, GET1RESOURCE, needs two parameters - 1 the name of the resource and 2 what number the resource is. Line 3 moves the ASCII codes for "snd " (that's "snd space") onto the stack using a long word, and line 4 moves the resource number, as a word on to the stack - in this case 128.

Now that the parameters have been put on the stack, we can call GET1RESOURCE by using the dc.w directive which inserts the word $A81F into the code. GET1RESOURCE returns a handle, which has exactly the same format and meaning as a window handle, on the stack. At this stage, the sound is loaded into memory, and the Mac has assigned a handle to it. if we want to play the sound, all the Mac needs is the handle on the stack and call the SNDPLAY trap and it will play that sound, as detailed in the next paragraph.

We move the sounds handle into d0 and then branch to a subroutine that will play the sound. The subroutine needs the handle of the sound in d0. The routine goes through the same procedure as before - it clears space on the stack for the return code - in this case it will be an error code (which we don't check here), then it puts the parameters on the stack and calls the routine - in this case SNDPLAY. When SNDPLAY has finished we move the error code off the stack and return to the main routine. Then its a case of restoring A5 off the stack and returning, effectively ending the program.

We could have dispensed with the subroutine totally, and written the Do_SOUND routine in the main code, however the idea of programming, and specially machine code programming is to keep the code as readable as possible. The ideal way to write the above program would be to take the code that loads the sound into memory, out of the main code and use a subroutine for that as well:

```
GET1RESOURCE:       EQU     $A81F
SNDPLAY:            EQU     $A805

START:      MOVEM.L     A5-A6,-(SP)         SAVE MAC WORLD      line 1
            BSR         LOAD_SOUND          LOAD IN OUR SOUND   line 2
            BSR         DO_SOUND            PLAY IT             line 3
            MOVEM.L     (SP)+,A5-A6         RESTORE REGISTERS   line 4
            RTS                             BYE                 line 5
```

Now its far easier to see what's going on. We save the important registers, load a sound, play it, restore the registers and leave.
This is called the main loop. The reason its called a loop is because normally a program will initialise everything, then continuously loop around a main loop waiting for things to happen. The more readable the main loop is, the easier it is to follow what is happening.
Here is Fantasm's main loop:

```
**MAIN LOOP
LOOP:       BSR         EVENTS      *check for something to happen
            TST.W       D0          *events returns in d0
            BEQ.S       LOOP        *wait for quit from menu - -1 if quit
```

By breaking large projects down into much smaller subroutines, things become a lot more understandable - especially if a fault develops in some code that you wrote six months ago. You will not remember how it works (guaranteed!).

In the next chapter we'll examine a complete macintosh application written in machine code.