•Beginners Mac Assembly•

 Chapter 4 

In this chapter we'll examine a simple Macintosh application - a basic text editor, that combines menus, windows, resources etc. The source code is in the Fantasm folder as "edit example", and has been split up into twenty sections so we may discuss the structure of the program. You'll also find the assembled code as an APP. Double click it now to see the end result.

To build the application you will need Resedit to copy the resources into the basic application that Fantasm produces. I would also recommend you install Macsbug.

Macsbug.

Macsbug is a Macintosh debugger written by Apple. It is placed in your system folder, along with its preferences. When you reboot your Mac, it should tell you that the debugger is installed.

After Macsbug is installed, you can drop into it at any time by pressing the interrupt button on you're keyboard - you may have to press the Apple key as well on some models.

When you are writing machine code, you can call up the debugger at a specific point by using the directive "debug" without the quotes. When your program reaches this point, it will stop and control will be passed to Macsbug:

```
fred:   move.w #10,d0
        debug
        add.w       d0,d1
```

In this example, after the line fred, Macsbug would gain control, and you will be able to step each instruction from the keyboard in slow motion as it were. Macsbug will display all the registers at the left of the screen, with your code at the bottom, and previous lines in the centre of the screen.

Macsbug works by redirecting the traps to all the error conditions that bring up alert boxes with things like "The application Unknown has quit because of an error of type 1", which will happens a lot in machine code programming!

Instead of the alert boxes, you will get Macsbug, with the current instruction and the state of all the registers.

Typing help in Macsbug will bring a fairly comprehensive help facility - for more on Macsbug read the documentation that comes with it.

NOTE - LC475's and Performa 475's need software FPU v3.0 to work with Macsbug

Editor example.

```
**********************SECTION 1*****************************************************
            INCLUDE     EDIT_EXAMPLE_EQUS INCLUDE OUR EQUATES
EDIT_EXAMPLE:
**INIT MAC**
            LEA   QDGLOB,A0   QUICKDRAW WORK SPACE
            LEA   202(A0),A0
            MOVE.L      A0,-(SP)          QDGLOB(202)
            DC.W  INITGRAF    INIT GRAPHICS
            DC.W  INITFONTS
            MOVEQ #-1,D0
            DC.W  FLUSHEVENTS
            DC.W  INITWINDOWS
            DC.W  INITMENUS
            CLR.L -(SP)
            DC.W  INITDIALOGS
            DC.W  TEINIT            INIT TEXT EDITOR
            DC.W  INITCURSOR
**END OF MAC INIT**
```

This first section is concerned with initialising the Mac. The first trap initialises the graphics. It needs the address of an array that will be used internally by the Mac for this applications graphics. We call the array qdglob, which stands for quick draw global. Next we initialise the fonts with the initfonts call, then clear out any outstanding i/o operations with flushevents. Flushevents needs the event to clear in d0, so by putting -1 in d0 we're saying clear all events. Then the windows, menus, dialog boxes, text editor and cursor are initialised with the relevant traps.

```
**********************SECTION 2*****************************************************
      MOVEM.L     A5-A6,-(SP) SAVE MAC WORLD
      LEA   VARS,A6           A6 IS OUR VARIABLES
      BSR   MAKERESMENU       GET MENU FROM RESOURCE FORK AND DRAW THEM

      BSR   CLEAR_PB          CLEAR PARAMETER BLOCK FOR FILE IO
      CLR.W VOLPTR(A6)        EDITOR VOLUME POINTER

      LEA   NO_W_TITLE,A0           DRAW WINDOW WITH DEFAULT TITLE
      BSR   DRAW_WINDOW
```

Section 2 carries on with the initialisation, but here we're concerned with the application rather than the Mac. First A5 and A6 are saved on the stack. When the program finished they are moved off the stack, thus restoring the registers to their original values.
A6 is used as a pointer to our variables, with the variables being accessed as offsets from A6. The next line branches to a subroutine that gets our menus from the resource file and puts up the menu bar.

BSR CLEAR_PB clears the parameter block used for file operations. The parameter block is 100 bytes, more details are given in the section concerned with the file i/o (input/output).

Next we initialise the volume pointer to zero. The volume pointer is a quick way of accessing a folder. Rather than have to mess about passing huge strings (folder names) to file routines, the

Mac will assign a unique number to a folder, so we can use this instead when we want to access a file.
The last two lines of section 2 open a window with the default title on the screen by loading the address of the title into A0 and calling draw_window.

```
********************SECTION 3*****************************************************
MAIN_LOOP:  BSR   EVENTS      MAIN LOOP STARTS HERE
            TST.W D0          EVENTS RETURNS -1 IF QUIT ELSE 0
            BEQ.S MAIN_LOOP   END HERE WITH SHORT BRANCH
```

Section 3 is the main loop. All it does is continually call events until events returns anything but 0 in d0.

```
********************SECTION 4*****************************************************
TERMINATE:  BSR   SHUTWINDOW        CLOSE WINDOW
            MOVEM.L     (SP)+,A5-A6 RESTORE REGS
            RTS                     BYEEEEEEEEEE
```

Section 4 terminates the program, when events returns any but zero, by closing the window, restoring A5 and a6 of the stack, and returning.

```
********************SECTION 5*****************************************************
EVENTS:     DC.W  SYSTEMTASK        LET THE MAC GET AN EYE IN
            TST.L TEHND(A6)         TEXT EDITOR HANDLE
            BEQ   NO_EDITOR         IF ZERO NO WINDOW OPEN
            MOVE.L      TEHND(A6),-(SP)
            DC.W  TEIDLE
NO_EDITOR:  CLR.B -(SP)             TRAP RETURNS A BYTE
            MOVE.W      #$FFFF,-(SP)      RETURN ANY EVENT
            PEA   SEVENTREC(PC),-(SP)     EVENT BUFFER ON STACK AS PARAM
            DC.W  GETNEXTEVENT
            MOVE.B      (SP)+,D0    THE EVENT IN D0
            EXT.W D0
            EXT.L D0
            TST.L D0
            BEQ   END_EVENTS        NOTHING HAPPENED
**1ST WORD OF SEVENTREC=EVENT
            LEA   SEVENTREC,A0      EVENT BUFFER
            MOVE.W      (A0),D0     THE EVENT IN D0
            CMPI.W      #1,D0       IF ITS 1 THEN D0 MOUSE PROCESSING
            BNE.S NOT_MOUSE
            BSR   DO_MOUSE
            RTS
```

Section 5 is events. The first line calls the OS routine systemtask. Remember that your program has total control of the machine, so if you want your Mac to be able to program switch and extensions to run in the background, then system task must be called periodically to let the Mac perform its regular housekeeping routines.
Next we need to check if the text editor has a window open, and if it has let it do its housekeeping by calling the OS routine “teidle”.
After taking care of the Mac, we check if the user has pressed a key, or clicked the mouse

by calling the OS routine “getnextevent”. This needs the address of an array it will use to pass

the information back, we call this array “seventrec” - system event record. Getnext event returns a byte value on the stack - if zero then nothing has happened and we branch to end_events which is the end of the routine. If the byte isn’t zero, we examine the event record and act accordingly. if the first word of the record is a 1, we goto do_mouse which handles the mouse control.

Section 5a handles the keyboard, if a 3 or a 5 is in d0 from events. A 6 means update and 8 is activate our window.

```
*****************SECTION 5B*************************************************
KEYDOWN:    LEA   SEVENTREC,A0
      MOVE.W      14(A0),D1   MODIFIERS (NORMAL KEY OR MENU CLICK)
      CLR.L D0
      MOVE.B      5(A0),D0    THE KEY THAT WAS PRESSED
      ANDI.W      #256,D1
      BEQ.S NORMAL_KEY  KEYBOARD KEY
**HERE ITS A MENU CLICK

      CLR.L -(SP) MENUKEY RETURNS A LONG
      EXT.W D0
      MOVE.W      D0,-(SP)    THE KEY
      DC.W  MENUKEY     GET MENU SELECTION
      MOVE.L      (SP)+,D0    IN D0
      BSR   DO_MENU     PROCESS MENU SELECTION (IN MENUS.S)
      RTS
```

Section 5b handles the key processing. The Mac will also return a key if a menu item has an apple key assigned to it, so for example “open” in the file menu can be activated by typing [command] O. We check for this by moving the word at 14 in the event record into d1 and anding it with 256. If its a zero its a normal key, if not its a menu key.
Note that it would be easier to test byte 15 in the event record, but as this is an example....
If byte 15 is 0, we call the OS routine “menukey”. This will return the handle of the menu, and the item number as a long on the stack. The upper word is the handle, and the lower word is the item. We move this long into d0 and call do_menu to process it, then return to the main loop.

```
******************SECTION 5C*************************************************
NORMAL_KEY: BSR   PUTACHAR          PRINT THE CHARACTER
      MOVE.W      #1,DIRTY(A6)      SET DIRTY FLAG - THE FILE IS MODIFIED
      MOVE.L      TEHND(A6),A0      GET TEXT EDITOR HANDLE
      MOVE.L      (A0),A0           POINT TO TERECORD
**CHECK FOR VERY ROUGH SCROLLING
      MOVE.W      16(A0),D0         GET CURSOR LINE NUMBER
      CMPI.W      #5,D0
      BGT   NOT_OFF_TOP             IF THE CURSOR Y IS > 5 ITS NOT OFF TOP


**HERE CURSOR HAS GONE OFF THE TOP OF THE WINDOW - WE NEED SCROLL THE WINDOW DOWN
      MOVE.L      SCROLLHND(A6),-(SP)     SCROLL BAR HANDLE
      MOVE.L      CURTOP(A6),D1     THE LAST TOP POSITION
```

```
      SUBI.L      #PAGELINES-1,D1   MINUS THE NUMBER OF LINES PER PAGE
      MOVE.W      D1,-(SP)          ONTO STACK
      DC.W        SETCTLVALUE             SET THE SCROLL BAR VALUE
      BSR         SCROLL                  SCROLL WINDOW
      BRA         CURSOR_ON_PAGE
**CHECK IF THE CURSOR HAS GONE OFF THE BOTTOM OF THE WINDOW
NOT_OFF_TOP:
      CMPI.W      #379,D0
      BLE         CURSOR_ON_PAGE    NO IT HASN'T

**CURSOR OFF BOTTOM OF WINDOW
      MOVE.L      SCROLLHND(A6),-(SP)
      MOVE.L      CURTOP(A6),D1
      ADDI.L      #PAGELINES-1,D1
      MOVE.W      D1,-(SP)
      DC.W        SETCTLVALUE       FLICK DOWN ONE PAGE
      BSR         SCROLL

**NOW LET SCROLL BAR KNOW HOW MANY LINES THERE ARE
CURSOR_ON_PAGE:
      MOVE.L      SCROLLHND(A6),-(SP)
      MOVE.W      NUMBLINES(A6),-(SP)
      DC.W        SETCTLMAX         SET SCROLL BAR MAX NUMBER OF LINES
      CLR.L       D0                RETURN 0
      RTS
```

Section 5c - normal_key, processes keyboard presses. First the character is printed in the editor window by calling putachar, which prints the character in the editor window. Then we set a flag called dirty. This tells our program that this file has been modified, so when we quit, or close the window, or open another file, the program can remind us that this file has been modified, and give us the option of saving it. Dirty is a word - this is a terrible waste of memory as a single bit would do just as well!

Having typed a character into the window, we need to check if the window needs to be scrolled There are literally an unlimited number of ways to do this, in this example we opt for the simplest (and most unreliable) of checking the current cursors coordinates, and compare them to the windows maximum and minimum y coordinates. If the cursor is outside this range, then the window either needs to be scrolled up or down. The current cursor coordinates can be found in the text editor record, which is pointed to by the text editor handle.

The text editor record looks like this:

```
struct TERec {
   Rect destRect;
   Rect viewRect;
   Rect selRect;
   short lineHeight;
   short fontAscent;
   Point selPoint;
   short selStart;
   short selEnd;
   short active;


   WordBreakProcPtr wordBreak;
   ClikLoopProcPtr clikLoop;
   long clickTime;
```

```
   short clickLoc;
   long caretTime;
   short caretState;
   short just;
   short teLength;
   Handle hText;
   short recalBack;
   short recalLines;
   short clikStuff;
   short crOnly;
   short txFont;
   Style txFace;                                    /*txFace is unpacked byte*/
   char filler;
   short txMode;
   short txSize;
   GrafPtr inPort;
   ProcPtr highHook;
   ProcPtr caretHook;
   short nLines;
   short lineStarts[16001];
};
```

Isn't it lovely? This is a C definition of the text editor record. How do we read this? Well in C "char" means a byte, "short" is a word, ptr is a long. The Mac also defines other types as follows:
Rect is four words defining start x,y and end x,y of a rectangle.
Point is a long with x coordinate in upper word and y coordinate in lower word.
Handle is a long.

Armed with this information, we can calculate the offset to any entry in the record. In this case we want the cursor coordinates, which we can get from "selrect" in the record, which starts at byte 16 of the record. How byte 16? The first entry in the record is destrect, which is 4 words, or 8 bytes. The next entry is viewrect, which is another 8 bytes. Then we have selrect, so selrect lives at terecord+16. We set a0 to point to the terecord, then we can get the cursor position by moving 16(a0) into a data register.

If you register your copy of Fantasm, you will get a list of the more important types of (current) Mac record and traps, and what parameters each trap requires. There are records for windows, files, resources, menus, in fact just about every type of Mac object has a record associated with it. Of course, if you want to program games, which will not be using Mac interfaces, then you wont need them anyway.

Section 6 updates our window by calling all the relevant OS routines - very boring!

Section 7 activates our window, if needs be, again by calling a few os routines - yawn.

Section 8 is a handy routine, that will create a Mac window, with scroll bar. Again its just a matter of calling OS routines, with the correct parameters - use it as a black box routine. If you want a smaller or bigger window alter the coordinates.
Shutwindow does just that - closes the window, shuts down the text editor and clears some handles.

Section 9 is a routine that will display an alert on screen. The alert must be in the resource fork, and do_alert needs the resource ID of the alert in d0.

Section 10 -do_mouse gets the mouse coordinates, relative to our window. If the mouse is in the menu bar it calls do_menu. If in the scroll bar, then the relevant scrolling action is performed, depending upon whether it was the up arrow, down arrow, the scroll gadget, or a click somewhere in the shaded area of the scroll bar.

Section 11 contains some utility routines. The first two generate the dialog boxes that put up the Mac file selectors for loading and saving files - very handy. One thing to note here is that a large proportion of the Mac OS was written in Pascal, so if you want to pass a string to the OS it must be Pascal format. Pascal format strings weren't designed for humans (Pascal was written by a Frenchman - Blaise Pascal - sorry Blaise but its true!), so we use C type strings, which are terminated in a 0 byte, and convert them to Pascal with C_to_P_str and vica versa.

Setscrollmax tells the scroll bar how many lines of text there are in the text editor, so the scroll bar can display the gadgets correctly (am I allowed to use the word gadgets, or do Commodore have a monopoly on it?)

Section 12 - doscroll handles the scroll bar. It calls the OS routine "testcontrol" to find out if any of the gadgets have been clicked, if so it calls scroll, then goes round again until the mouse button is released.

Section 13 contains two string convertors written in a different way - these need the address of the strings passed on the stack. This illustrates another method of passing parameters to subroutines. Normally parameters (data a routine needs, or returns) are passed in registers because it is the quickest way to do it. However, sometimes you may not have enough registers to do this, so you can push the params on the stack, and the subroutine can pop them off, or read them directly from the stack. These two routines read the parameters off the stack by linking a6 to the stack pointer, and reading the parameters as offsets from a6.
Putting parameters on the stack is the standard way that C compilers pass data between procedures (subroutines), so knowing how its done is handy.

The other routine in section 13 is a little routine that will change a windows title. It needs a3 pointing to a the window title as a c string.

Section 14 firstly includes the rest of the routines needed by including file_io.s and menus.s, then defines our variables and data.

Section 15 contains the routines that will read a file into the text editor. It needs a0 pointing to the filename, and the volume pointer in d0. To read a file, we have to open it, read it into memory, then close it. Files must be closed before the program quits. If they are not, data could be lost, the file could be damaged, or permanently lost!
Generally its good practice to always close a file as soon as you've read it - then it doesn't matter if the Mac crashes for whatever reason - you've closed the file and it cant be corrupted (in theory anyway!)

The mac's in built text editor is limited to 32K - files bigger than that will generate an error. We have to check the size of the file. This is done quite simply by examining the files' parameter block. The Mac has two ways of dealing with files, either parameter block file operations, or hiracheal filing system operations. The first is the simplest, which we are using here.

Before each call to the OS file routines, the parameter block must be set up correctly for that call. The routine will alter the parameter block depending on the type of operation. For example if we read a file, a location in the parameter block will reflect the number of bytes read. When we write a file, we can set the files creator by setting four bytes in the parameter block etc

The parameter block is made up of a header, followed by a block that is specific to that type of operation. Examples of important file operations include open, close, read, write, getinfo, setinfo, create, seteof, geteof.

The header looks like:

```
#define ParamBlockHeader \
   QElemPtr qLink;                  /*queue link in header*/\
   short qType;                     /*type byte for safety check*/\
   short ioTrap;                    /*FS: the Trap*/\
   Ptr ioCmdAddr;                   /*FS: address to dispatch to*/\
   ProcPtr ioCompletion;            /*completion routine addr (0 for synch calls)*/\
   OSErr ioResult;                  /*result code*/\
   StringPtr ioNamePtr;             /*ptr to Vol:FileName string*/\
   short ioVRefNum;                 /*volume refnum (DrvNum for Eject and MountVol)*/
```

The only two we're interested in are ioNamePtr, which is where the address of the filename string has to go, and ioVrefNum, where the volume pointer goes.

Following the header is the rest of the parameter block, which will vary depending on the operation. For most operations (read, write, rename,geteof,seteof,open and setfiletype it looks like this:

```
struct IOParam {
   ParamBlockHeader
   short ioRefNum;                  /*refNum for I/O operation*/
   char ioVersNum;                  /*version number*/
   char ioPermssn;                  /*Open: permissions (byte)*/
   Ptr ioMisc;                      /*Rename: new name (GetEOF,SetEOF: logical end of file)
(Open: optional ptr to buffer) (SetFileType: new type)*/
   Ptr ioBuffer;                    /*data buffer Ptr*/
   long ioReqCount;                 /*requested byte count; also = ioNewDirID*/
   long ioActCount;                 /*actual byte count completed*/
   short ioPosMode;                 /*initial file positioning*/
   long ioPosOffset;                /*file position offset*/
};
```

Examine file_io.s and try to relate this parameter block layout to what's happening in open, read, write etc.
Here's a rough interpretation of the various offsets in the parameter block for simple file operations, as used in the example editor:

```
18 (long)=address of the filename
22 (word)=volref
26 (byte)=version for create, else 0 otherwise
27 (byte)=2 if write,1 if read
28 (long)=use hd buffer. 0 for open, after open, size of file in here
32 (long)=creator i.e. "text" for create. Address of data buffer for read/write
36 (long)=length of buffer for write  or size of file if read (after open)
40 (long)=actual number of bytes read.


44 (word)=1.w for write, 0 for read
46 (word)=offset (normally zero)
```

Note in section 15 the subroutine my_pbio - this is just to provide a convenient and tidy way of calling the various os file routines. Again it needs the parameters on the stack, first the

routine trap number, and then the address of the parameter block. In a real program, this method would not be used, as its slow, but it gives an example of using the stack. Note that because my_pbio doesn't pop the data of the stack, when my_pbio finishes (returns), the calling routine has to add 8 to the stack pointer to correct for the two parameters that were pushed onto the stack as longs.

Before we go any further, I suggest you experiment with the example editor. Writing applications like this for the Mac is generally a matter of stringing together a load of operating system routines in the right order, and passing the correct parameters. Its not something I'd personally call fun, but people do seem to get a lot of enjoyment out of writing filing system programs (catalogue programs, editors etc), and general Mac applications.

However, to stand any chance of writing a killer Mac application, you do need a complete list of all the Mac OS calls, and the data structures involved - you can get them from Apple at a price, or the Internet, or from us by registering. We cant include them in the distribution of Fantasm because of the size.

This chapter has necessarily glossed over the intricacies of a complete application. The best way to learn is not by reading about something, but by trying it for your self. This example editor is provided only as a reference piece.

Of course, we'll gladly supply more information to people that register.

In chapter 5, we'll be examining writing programs without the OS, and Macintosh machine code graphics.