The Atari ST M68000 tutorial part 4 – of the ways of addressing memory

So, finally; addressing! Before writing this, I read the first two chapters in Steve William's Programming the 68000, and it was very good reading. He explained some things that I did not really go into, and above all, he has extensive explanations of addressing, before going into any code. Well well, all I can say is that I have a bit more pragmatic approach, this is after all a tutorial, and the aim is for you to learn through doing. Theory will usually only be covered in connection with practice. With this in mind, I urge you to get your hands on some M68K book and study it, because it goes through things more in theory than I do. You will probably find some of these books on some dusty bookshelf in your local library.

You should have a pretty good idea what memory is, and also what the memory registers do. Memory is simply put the computers warehouse for storing numbers. Numbers are stored in binary format, that is, only 1's and 0's (refer to tutorial 2 for information on this). There are also standard formats for how many 1's and 0's are stored. The M68K has three; byte, word and longword. A byte is 8 1's and 0's, a word is 16 and a longword is 32. So the quantities have the following storage capacities.

Name	1's and 0's	Capacity
byte	8	2^8 = 256
word	16	2^16 = 65536
longword	32	2^32 = 4294967296

So, move.w means move a value, with word size. If we move a value that is bigger than word size (value > 65536), the value will be truncated (cut off), since we tell the computer to only move 16 1's and 0's. The term "1's and 0's" is called a bit. So, a byte contains 8 bits, never ever never confuse bits with bytes. The capacity is also somewhat of a misnomer. Since computer start counting at 0, the maximum value containable in a byte is 255. Also, in order to have negative numbers, the range of a byte is usually -128 to +127. Arithmetic that does not deal with negative numbers is called unsigned, and when negative numbers are involved, it's called signed arithmetic.

Now, we can move on to addressing modes I think. Each address is a reference or pointer to a place in memory. An address does not point to specific bits, but rather to bytes, meaning that address $2 does not point to the second bit in memory, but rather points to the second byte, which begins with the 9th bit in memory.

Address register	Contains
a0	$2

Memory	Contains
$1	%10101010
$2	%01010101
$3	%10101010

If we moved the byte that address register a0 pointed to, we would get %01010101, but if we moved the word that a0 pointed to, we would get %

0101010110101010. If we simply moved the value of a0, we would get $2. There is a great difference to getting the value an address register points to, and the value in the address register. When we want something the register points to, we use parentheses around the address register, like this (a0). So, (a0) means the value of the memory place that a0 points to, in this case % 01010101, and simply a0 (without parentheses) means the value in a0 itself, in this case $2.

You can also put '+' and '-' characters before or after the address, meaning that you wish to increase or decrease the address registers value, either before or after the operation has been performed (called post or pre increment or decrement). Therefore, move.b (a0)-,d0 means put the value of byte size that a0 points to in d0, then decrease a0 with one. The value that the address will be manipulated with, is dependent on the memory chunk size, if it's byte, then 1, if word 2 and longword 4, in order to keep up with the changes. This is ideal for moving large areas of memory, you put your move instruction with post increment in a big loop. Christ, it feel like I've explained this very poorly, but don't worry, it will become clear with time I think, and especially after this example.

Memory	Contains
$1	%00000001
$2	%00000010
$3	%00000011
$4	%00000100
$5	%00000101
$6	%00000110
$7	%00000111
$8	%00001000

a1	$1
a2	$2
a3	$3
a4	$4

Command		Effect
move.b	(a1),d0	d0 = %00000001
move.w	(a1),d1	d1 = %0000000100000010
move.b	(a3)+,d2	d2 = %00000011, a3 = $4
move.b	(a3),d2	d2 = %00000100
move.l	#$3,a3	a3 = 3

make a3 point to $3

move.b	$1,d0	d0 = %00000001

put the value of $1 in d0

move.w	-(a3),d3	a3 = 1, d3 = %0000000100000010
move.l	#$1,a1	a1 = 1
move.l	#$5,a2	a2 = 5

loop 4 times

```
        move.b  (a1)+,(a2)+
```

end loop

Result	
$1	%00000001
$2	%00000010
$3	%00000011
$4	%00000100
$5	%00000001
$6	%00000010
$7	%00000011
$8	%00000100

As you might have noticed, if you studied the example thoroughly, when you want an address register to point to a place, you use # when giving the address, but when you want the content of the address of a memory location, you don't use #. Thus, if you want a0 to point to $100, you use move.l #$100,a0. But if you want the value of memory address $100 to be put into d0, you use move.l $100,d0. So, with a '#', you have a value, but without '#', you have a pointer.

I think that should cover it nicely. These memory addressing modes are our main concern, however, there are some more. You can use two address registers in order to get index (place of pointer), or you can add either a data register or a fixed value to an address. These are pretty self explanatory and you'll see when they come by. For example move.b (a0,a1),d0 means move the memory value pointed to by a0 + a1 into d0.

Now, in order to get an even firmer grip on traps, which seem to be hard to grasp, I'll explain a bit more about the stack pointer. The trap part is the one I've had to edit the most because it was unclear. The stack pointer works like any other address register, the only difference is that some functions have the stack pointer as default address pointer, for example, traps. The idea of the stack is something that you can put data in, the last data entered is the first data that comes out, like spring loaded platforms for plates in cafeterias. When you put data on the stack, it's called push, and when you retrieve it again, you pop. So if you push the numbers one and two onto the stack, and then pop two items, you will get two and one (last in, first out; LIFO).

When pushing (putting items on the stack), you address it using pre-decrement addressing mode, and when popping, post-increment. This means that if the stack from the beginning points to $100, it will point to lesser and lesser values as you push data, and will increase in value as you pop.

```
                        a7 = $100
move.w    #10,-(a7)     push 10 onto the stack
                        a7 = $98
move.w    #8,-(a7)      push 8 onto the stack
                        a7 = $96
move.b    #1,-(a7)      push 1 onto the stack
                        a7 = $95, uneven address, be careful
move.b    (a7)+,d0      pop stack into d0
                        a7 = $96
```

```
move.w    (a7)+,d1    pop stack into d1
                      a7 = $98
```

As you see, the stack clears itself up when using push and pop instructions. However, when we use traps, the BIOS, XBIOS or GEMDOS won't clear up the stack for us, so in order for the stack to keep the correct address, we have to add a certain number at the end of the trap call. If we don't do this, the stack will not point to the correct address, and when you start pushing and popping, everything will be out of alignment. Also, of interest to note, is that when you go into super mode, the user stack is replaced by the super stack, so it needs to be backed up in order to be restored later, when we switch back into user mode again.

When we already are into talking about memory and addressing, I thought I'd cover how to make your own variables and what the text to the leftmost bit really is. As you must have noted, all instructions are one tab in, so to speak, while the name of subroutines and variables are at the leftmost in the text, well, there you have it :) Whet happens when you put text to the leftmost in your source code, is that you tell the assembler that that memory position, is also known as the text you entered. Every line of source code has its memory value, which of course is some hex value, so instead of trying to figure out that hex value, you tell the assembler that "henceforth, this line shall be known as [whatever you write]". Also, any text written after the instruction, is treated as comments and are passed over by the assembler. You may also use a '*' to denote comments. Comments beginning with a '*' can be inserted everywhere. Disclaimer; the "actual memory position" values are purely for example and have nothing to do with real life (or computers).

```
Actual memory position          Label       Commands
        Comments

$0                  first_line  move.w      #10,d0      easy as pie
$2                              move.w      #$0,a0
$4                  bra         a0                      moves to $0
$6                  bra         first_line              moves to $0
$8
$a                  * a nice comment
$c                  exit        clr         -(a7)       never reached
$e                  trap        #1                      clean exit
```

The command bra, for Branch, is used to jump to different memory positions, what it does then is to alter the value of the program counter (PC). You remember, the address register that holds the position of the next instruction to be executed. Branching will be covered extensively in the next tutorial. Since variables are just chunks of data at a certain memory position, they are defined in almost the same way. You have a name for the variable, and then you say either dc.N, where N is either b, w or l for byte, word or longword, or ds.N. DC stands for Define Constant, and DS is Define Storage.

DC is a variable, while DS is a large storage area of memory. The number after DC is the initial value of the variable, and the number after DS is

how many variables of the same type you want. DS is used for creating big memory spaces that you want to put stuff into later, like a bitmap or so, more on this in another tutorial. The DC area should be denoted by a “section data”, and the DS section should be denoted by a “section bss” (Block Storage Segment). Section data comes first, and section bss next, these areas are to be put last in the code.

```
            section data
temp        dc.l        0           a longword sized chunk of memory, given
value 0

            section bss
storage     ds.w        4           four words after one-another
storage2    ds.l        2           two longwords after one-another,
                                    since one longword is two word
                                    storage and storage2 have the same size
```

I didn’t come up with any creative way to use our new-found knowledge of addressing modes, so I just made some changes to the program we already have. For example, an unnecessary putting of the background colour memory in a0 instead of just accessing it directly, and moving the temp storage from d0 to the stack instead.

```
            jsr         initialise          jump to initialise

            move.w      $ffff8240,-(a7)     push old colour to stack
            move.l      #$ffff8240,a0       a0 points to colour 0
            move.w      #$700,(a0)          put $700 where a0 points

            move.w      #7,-(a7)            wait for a keypress
            trap        #1                  call gemdos
            addq.l      #2,a7               clear up stack

            move.w      (a7)+,(a0)          pop from stack

            jsr         restore             jump to restore

            clr.l       -(a7)               clean exit
            trap        #1                  call gemdos


initialise
* set supervisor
            clr.l       -(a7)               clear stack
            move.w      #32,-(a7)           prepare for user mode
            trap        #1                  call gemdos
            addq.l      #6,a7               clean up stack
            move.l      d0,old_stack                backup old stack
pointer
```

```
* end set supervisor

            rts

restore
* set user mode again
            move.l      old_stack,-(a7)         restore old stack pointer
            move.w      #32,-(a7)               back to user mode
            trap        #1                      call gemdos
            addq.l      #6,a7                   clear stack
* end set user

            rts


            section data

old_stack   dc.l        0
```

That was that, I hope you know enough about addressing now to push on. In the next tutorial we will probably cover the graphics memory a bit, and what you can do with it. This means we'll finally get some action people! I'm starting to get bored of only changing the colour of the background, aren't you? We have covered most of the basic theory I think, which means that in the future there will be more practical coding, like techniques for scrollers, moving sprites, making rasters and stuff like that.

perihelion of poSTmortem 2002-05-06

"Target that explosion and fire."

- Star Trek VI, the Undiscovered Country

Last edited 2002-06-14