The Atari ST M68000 tutorial part 9 – of revealing the unseen and expanding our consciousness without the use of illegal drugs

It's been a while since the last tutorial, almost a month actually, sorry for that. I've had a rough class in school, but that's no excuse since I found lots of time to play computer games. I just haven't felt up to it. Now, summer holidays are on and I plan on coding some for myself, besides the tutorials, but since I need the knowledge myself, you can look forward to a tutorial on sprites and how to handle the joystick (with that, one could make a nice shoot- em- up game, yay). This tutorial however, will, as promised some while back, cover timings. To have some practical example to work with, I'll show you how to do the neat trick of killing the upper and lower border.

But now for something completely different; Boolean algebra. Boolean algebra states that the world is neatly and nicely built up of true or false, black or white, good or evil, 1 or 0. The last bit there applies to us as computer programmers. Boolean algebra is all about bit manipulation. There are a few so called logical operands, that you can use to compare two bits to each other, and get the result true or false (1 or 0) from the equation. The ones I will cover here are and, or and exclusive or. In each case, there are two bits involved, resulting in four different combinations of those bits, this is to hard to put in words, see below for how it works.

AND

Bit 1	Bit 2	Result
1	1	1
0	0	0
1	0	0
0	1	0

OR

Bit 1	Bit 2	Result
1	1	1
0	0	0
1	0	1
0	1	1

EOR (XOR in some other languages)

Bit 1	Bit 2	Result
1	1	0
0	0	0
1	0	1
0	1	1

For an and operation to be true, both operands need to be true (in programming lingo, that means that the result of an and operation is 1 if both bits are 1). For an or operation to be true, either one or both of the operands must be true. For an exclusive or operation to be true, either one, but not both, of the operands must be true.

These kinds of operations become extremely important when doing stuff to the screen memory later on. For example, imagine you have a

screen filled with colour (all 1's in the screen memory), and you want to clear out just that one bit in a certain place. You then prepare a so called mask, and and it in. A mask really is a quantity, that is to be applied in a logical operation on another quantity, in order to produce the result you want, that is one hard and stupid way of explaining it. Example again, in this example, we want to clear the most significant bit and keep the others intact.

```
            Mask
            %01111111

            Memory
            %11111111

            and mask,memory     (pseudo code)
            %01111111
and         %11111111

result      %01111111
```

When performing and here, you just compare bits one after another, in the most significant bit, the and operation becomes false, thus the result is 0, and in all other cases, it's true. So by having this mask, and and:ing it with the screen memory, we have a good way of clearing away bits, we could create a raster by using a %10101010 mask.

Each operation, that is and, or and exclusive or, is good for different things. As we have seen, and is good for clearing bits. Exclusive or is good for many things, but the most obvious one is flipping bits, if you exclusive or a bit with 1, that bit will always "flip" (go from 1 to 0, or 0 to 1). Or is good when you want to set some bits, no matter what value they had before, it's called setting a bit when you make it 1, or true. So and clears, or sets and exclusive or flips, that really covers most things that need to be done. Of course, you can most likely come up with devious plots to do different things than the ones we've gone through here.

Now, onto timings! When an exception occurs, normal program execution halts and the ST looks at a certain vector (memory position) depending on the kind of exception, and then executes what it finds there. What this means is that when an exception occurs (exceptions are "special events") the ST looks for an address pointer at a given address, and jumps there. For example, when an address error occurs, there is an address error exception. The address at $00c, is the address error exception address, so every time there is an address error, the ST will jump to the address found at $00c. This address, we can change ourselves.

```
            into supervisor mode
            move.l      $00c,-(a7)              backup address error vector
            move.l      #address_error,$00c     put our own routine there
            make address error occur                    for example, an
uneven address call
            move.l      (a7)+,$00c              restore address error vector
```

```
                into user mode
                exit

address_error
* our own address error routine, replacing the normal address error routine
                output error text, or do something else, freedom of choice
                rte                                     return from exception
```

In normal cases, when there is an address error, there will be three bombs on the screen, but with the little program above, we can change what happens when an address error occurs. We could make the address error routine do anything, like changing background colour; quite fun, every time there is an address error, instead of three bombs, the background colour changes. The program above won't really work, some things are missing, you will replace the bombs with some effect of yours, but the ST will probably hang in all sorts of ways, it's just provided as a demonstration. As a side note, whenever an exception occurs, the status of the ST is saved at $384 and a bit forward, you can read exactly about that in ST Internals pp. 235- 237. The ST Internals is a great book by Abacus Software, that describes much of the hardware of the ST.

The ST has several timing pulses, that generate exceptions, this means that we can control these timing pulses and make them work for us. I'll explain the most simple one, the $70 vector. Every VBL, an exception occurs and the ST jumps to the address stored at $70. So instead of using the old way we've been using with doing a VBL check at the start of our main routine, we can put our main routine in the $70 vector, because it will start every VBL! All exceptions must end with a rte command, ReTurnException, compare this to the rts command. Here's a little pseudo code on the usage of the $70 vector.

```
                into super mode
                move.l      $70,old_70                  backup $70
                move.l      #main,$70
                wait key press
                move.l      old_70,$70                  restore $70
                out of super mode
                end program

main
                do stuff
                rte

                section data
dc.l            old_70
```

The thing here which might seem a bit strange is the wait key press and then just a clean exit. Well, the thing is that once we hook up the $70 vector, the main routine will be executed every VBL, so while the ST waits for a key to be pressed, the main routine will execute. In a bigger program, you can start off by hooking up, say a music routine on the $70 vector, then load in lots of stuff from disk, meanwhile, the music will play, then after loading is

finished, you change the $70 vector to the real program so to speak. Endless possibilities :)

Oh, btw, the routine may not take more than 1/50th of a second to perform, because if it does, the ST will call the routine again, while you are still executing it and that won't work. Use the background colouring method from the last tutorial to see how much time your routine takes. Also, you must backup all your registers and restore them at start and finish of the $70 routine, otherwise your computer might crash for some strange reasons. Here's how to do that really simple, by pushing and popping them on and off the stack.

```
vbl
        movem.l   d0-d7/a0-a6,-(a7)     backup registers
        ...                             do stuff
        movem.l   (a7)+,d0-d7/a0-a6     restore registers
        rte                             exit vbl routine
```

Btw, using the $70 vector for your main instruction is slightly faster than the technique we used before. There is a little chip in the ST that is called MFP, for Multi Functional Peripheral, it can do lots of cool stuff, but right now we're interested in its timers, there are four timers that control timing pulses, and we will be interested in looking at one of them; Timer B. This is the complete list of the MFP registers, all are 8 bits.

Address	Register
$fffa01	Parallel port
$fffa03	Active Edge register
$fffa05	Data direction
$fffa07	Interrupt enable A
$fffa09	Interrupt enable B
$fffa0b	Interrupt pending A
$fffa0d	Interrupt pending B
$fffa0f	Interrupt in-service A
$fffa11	Interrupt in-service B
$fffa13	Interrupt mask A
$fffa15	Interrupt mask B
$fffa17	Vector register
$fffa19	Timer A control
$fffa1b	Timer B control
$fffa1d	Timer C & D control
$fffa1f	Timer A data
$fffa21	Timer B data
$fffa23	Timer C data
$fffa25	Timer D data
$fffa27	Sync character
$fffa29	USART character
$fffa2b	Receiver status
$fffa2d	Transmitter status
$fffa2f	USART data

These are the vectors

$134	Timer A vector
$120	Timer B vector

To make things difficult fore some strange reason, Atari decided that the names given to the MFP registers would be misnomers, at least I think they are. As I said, there are four timers. The timers share some registers, here's how that's broken down.

Timer A

All of

$fffa19	Timer A control
$fffa1f	Timer A data

Bit 5 of

$fffa07	Interrupt enable A
$fffa0f	Interrupt in-service A
$fffa13	Interrupt mask A

Timer B

All of

$fffa1b	Timer B control
$fffa21	Timer B data

Bit 0 of

$fffa07	Interrupt enable A
$fffa0f	Interrupt in-service A
$fffa13	Interrupt mask A

Timer C

Bit 5 of

$fffa09	Interrupt enable B
$fffa11	Interrupt in-service B
$fffa15	Interrupt mask B

So you see, timer A and B share some registers, and only use one bit in those shared registers. OK, that's a long list, but we don't have to worry about to many of those address. We'll only be using enable A, mask A, mask B, Timer B control, Timer B data and two vectors; $70 and $120, if that's of any comfort. Right now, you are probably wondering your ass off, that's ok, I did too the first time I read about this (in the tutorials by James Ingram).

If you wonder about the MFP, and exactly where it is physically in the ST, it's not necessary to know. You access the timer addresses just as you would any other address. The ST has many small chips that do stuff, like controlling the joystick, the sound and so on. The only thing you need to know to handle them is where they are in memory, every device is "mapped"

to memory, so just think about the ST as one big list of memory positions, by changing the memory, you change the way the chips inside the ST work.

It really is due time to do something practical with all of this. Timers A and B can be in one of many modes, controlled by Control A and Control B respectively. For Timer B, the most interesting one is #8, event count mode. When Timer B is in event count mode, it will interrupt for every Nth scan line , where N is the number put in Timer B data (thus 2 means every second scan line, 1 means every scan line). So if we put Timer B in event count mode, put number 1 in Timer B data, then the instructions found at $120 will be executed on every scan line, very much like $70 will be executed every VBL. For this reason, Timer B is also called HBL, Horizontal BLank.

Now this is interesting and useful, finally. In order to turn timer B on, we must set bit 5 in both Enable A and Mask A. To manipulate certain bits we use the commands bset, for Bit SET and bclr for Bit CLeaR. Here's how we actually do to make the ST jump to a certain address every scan line.

```
        clr.b     $fffffa1b          disable timer b
        move.l    #timer_b,$120      move in my timer b address
        bset      #0,$fffffa07               turn on timer b in enable a
        bset      #0,$fffffa13               turn on timer b in mask a
        move.b    #1,$fffffa21               number of counts, every scan line
        move.b    #8,$fffffa1b               set timer b to event count mode
```

Now the address at #timer_b will be jumped to every scan line. What really fires away Timer B is the activation of the Timer B Control ($fffffa1b) when we put it in event count mode. Whenever we exit a Timer B exception, we must tell the ST a bit more specifically than when we exit form a $70 exception. We have to clear the 0 bit in in-service A, like this.

```
        bclr      #0,$fffffa0f               tell ST interrupt is done
        rte                          return from exception
```

You must also back up all registers you plan to use in the interrupt, or you'll once again get a crash. So finally, we know how to use Timer B at least, and we have the power to know exactly at what scan line we're at (do we really understand this?). It might be very frustrating with all those addresses and how they work and so, actually, it's not so much to understand, rather just accept. When we put certain values into these registers, stuff will happen, memorize the addresses to make life easier, and just go about your work.

So how do we kill borders? This also is somewhat "just do it and realize it works". In order to kill the top and bottom border, you change from PAL to NTFS exactly on the correct scan line, then wait some for the effect to kick in and then back again. For killing the top border, it's the first scan line, for killing the bottom border, it's the last scan line.

For killing the top border, you just wait some, about 15000 clock cycles, which will put the electron beam on the first scan line and then toggle PAL/NTFS, for killing the bottom border we check when we're on the last scan line, and toggle PAL/NTFS.

Did someone say toggle and check for scan line? Yes someone did (that was me), and haven't we just learned how to do just these things; an exclusive or and Timer B will do the trick! Now we just need one more thing; how to change between PAL and NTFS, it's probably in memory somewhere, so whip out the Memory.txt and do a search.

The synchronization mode is controlled by bit 1 at address $ff820a. If this bit is 1, the system is in PAL (50Hz) mode, and if it's 0 the system is in NTFS (60Hz) mode. Even though this will work and kill the borders, there will be lots of flickering due to Timer C and other interrupts interfering. The reason for the flicker, is that the interrupts will interfere with our time critical calculations. To disable Timer C, just clear bit 5 of Mask B, to disable all interrupts, we have to mess around some with the status register.

The status register is made up of 16 bits, the first 8 bits being the user bits and the next 8 the system bits. The user bits are so called flags, and record the result of the latest operation. The system bits control interrupts, a trace bit and the supervisor bit.

Bit	Name
0	Carry flag
1	Overflow flag
2	Zero flag
3	Negative flag
4	eXtended flag
8	Interrupt
9	Interrupt
10	Interrupt
13	Supervisor bit
15	Trace bit

The carry flag is set when the result of an arithmetic operation is too big to fit, this is the same as the little memory tag used by humans when adding or multiplying with pen and paper. Say we want to add a number and put it in d0.b, and the result is %100000000 = 256, 9 bits won't fit in d0.b, so d0.b will contain all zeros and the carry flag will be set. Also set when a borrow occurs in a subtraction. The overflow flag is set when the result of an arithmetic operation is too high to fit in the destination, like the add example above. The zero flag is set when the result of an operation is zero (the strange, mystic workings of computers, which seldom make sense to the human mind ...) . The negative flag is set when the result is negative. The extended flag is as the carry flag in arithmetic operations, otherwise it can serve special functions given for each instruction.

Note in all the flags the difference between arithmetic operations and other operations. The trace flag is set when the computer is in trace mode, as it is when debugging, performing only one instruction at a time.

Depending on how the interrupt bits are set, the ST will accept different interrupt levels. In our case, the only interesting interrupt level is

when all bits are set, because then all interrupts are disabled. So, we want to set bit 8, 9 and 10, but not touch any of the other bits. An or operation has the power to set some bits, and leave all other alone. By or:ing the status register with %0000011100000000, we make sure that bits 8 – 10 are set, and that all other bits are left as they were. In order not to have to write that cumbersome number each time, we instead use $0700, which is the same number. Of course, the status register must also be backed up. I'm tired of all theory, so I'll just drop all source code in your face right now and go through it.

```
            jsr         initialise

            movem.l     picture+2,d0-d7     put picture palette in d0-d7
            movem.l     d0-d7,$ff8240       move palette from d0-d7

            move.l      #screen,d0          put screen1 address in d0
            clr.b       d0                  put on 256 byte boundary

            move.l      d0,a0               a0 points to screen memory

            clr.b       $ff820d             clear STe extra bit
            lsr.l       #8,d0
            move.b      d0,$ff8203                  put in mid screen
address byte
            lsr.w       #8,d0
            move.b      d0,$ff8201                  put in high screen
address byte

            move.l      #picture+34,a1      a1 points to picture

            move.l      #11199,d0           320*280 / 8 - 1
loop
            move.l      (a1)+,(a0)+                 move one longword
to screen
            dbf         d0,loop

            move.l      #backup,a0                  get ready with backup
space
            move.b      $fffa07,(a0)+               backup enable a
            move.b      $fffa13,(a0)+               backup mask a
            move.b      $fffa15,(a0)+               backup mask b
            move.b      $fffa1b,(a0)+               backup timer b
control
            move.b      $fffa21,(a0)+               backup timer b data
            add.l       #1,a0               make address even
            move.l      $120,(a0)+          backup vector $120 (timer b)
            move.l      $70,(a0)+           backup vector $70 (vbl)

            bclr        #5,$fffa15          disable timer c
            clr.b       $fffa1b             disable timer b
            move.l      #timer_b,$120       move in my timer b address
```

```
        bset    #0,$fffa07          turn on timer b in enable a
        bset    #0,$fffa13          turn on timer b in mask a

        move.l  #vbl,$70

        move.w  #7,-(a7)            wait keypress
        trap    #1
        addq.w  #2,a7

        move.l  #backup,a0
        move.b  (a0)+,$fffa07               restore enable a
        move.b  (a0)+,$fffa13               restore mask a
        move.b  (a0)+,$fffa15               restore mask b
        move.b  (a0)+,$fffa1b               restore timer b
control
        move.b  (a0)+,$fffa21               restore timer b data
        add.l   #1,a0               make address even
        move.l  (a0)+,$120          restore vector $120 (timer b)
        move.l  (a0)+,$70           restore vector $70 (vbl)

        jsr     restore

        clr.l   -(a7)
        trap    #1

vbl
        move.w  sr,-(a7)            backup status register
        or.w    #$0700,sr           disable interrupts
        movem.l d0-d7/a0-a6,-(a7)   backup registers

        move.w  #1064,d0
pause
        nop
        dbf     d0,pause            about 15000 cycles pause

        eor.b   #2,$ff820a          toggle PAL/NTSF
        rept    8
        nop                         wait a bit ...
        endr                        ... for effect to kick in
        eor.b   #2,$ff820a          toggle PAL/NTFS back again

        clr.b   $fffa1b             disable timer b
        move.b  #228,$fffa21                number of counts
        move.b  #8,$fffa1b          set timer b to event count mode

        movem.l (a7)+,d0-d7/a0-a6   restore registers
        move.w  (a7)+,sr            restore status register
        rte                         finnished interrupt
```

```
timer_b
            movem.l   d0/a0,- (a7)                  backup registers
            move.l    #$fffa21,a0                   timer b counter
address
            move.b    (a0),d0           get timer b count value
            pause_b
            cmp.b     (a0),d0           wait for it to change
            beq       pause_b           EXACTLY on next line now!

            eor.b     #2,$ff820a        toggle PAL/NTSF
            rept      8
            nop                         wait a bit ...
            endr                        ... for effect to kick in
            eor.b     #2,$ff820a        toggle PAL/NTFS back again

            movem.l   (a7)+,d0/a0                   restore registers
            bclr      #0,$fffa0f        tell ST interrupt is done
            rte                         exit interrupt


            include initlib.s


            section data

picture     incbin    kenshin.pi1


            section bss

            ds.b      256
screen      ds.l      11200

backup      ds.b      14
```

Phew, that was some. Nice and gentle walkthrough. First, just as usual, just initialise screen and so on. The picture is 320*280 pixels, instead of the normal 320*200. For compatibility reasons, I did it in Degas format, so you'll have no problem looking at it in Degas, but you'll not see the last 80 scan lines. With the borders killed, my guess is that we'll se about 270 or so scan lines, a bit depending on monitor, perhaps a bit less.

After the picture is loaded into the screen, I back up all the registers used, it's essential to return to the state before the program was run. As you see, the backup is a little storage area of 14 bytes that is loaded into a0, and then data is moved in. It only backs up 13 bytes of data, but it starts off by backing up 5 bytes of data, putting it on an uneven address, that means that the two addresses which are then backed up, will be on uneven addresses, which is bad. So after the five bytes, I add one to a0 in order to put it on an even address, so the storage area needs to be 14 bytes in order to handle the extra empty byte.

Then, disable Timer C, and Timer B. I only disable Timer C and do nothing more with it, with Timer C on, there would be disturbances due to the critical timing of the border killing. Put the correct address in the Timer B vector, and then enable Timer B by setting the correct bits in Enable A and Mask A. Next, kickstart the main routine (here called vbl) and just wait for a key press. After the key press, everything is restored and a clean exit performed.

The VBL routine starts off by backing up the status register and disabling all interrupts, then it continues by waiting. By my calculation, we are waiting for exactly 15074 clock cycles. Nop, NoOPeration, is a command that does exactly nothing but take 4 clock cycles. Backing up the status register is a move instruction, that takes 12 clock cycles, and an or instruction on memory takes 8 clock cycles if it's word sized. A movem from registers to a pre-decremented memory position takes 8 clock cycles, plus 10 per register moved since we use long-word size, and each dbf takes 10 clock cycles. This should add up to 12 + 8 + 8 + 10 * 15 + (10 + 4) * 1064 = 15074 clock cycles. Since I just took this method from James Ingram's tutorials, I haven't really experimented with it and don't know exactly how far you can stretch it (that is, what happens if you delay by say 15070 clock cycles instead).

Now comes the part that actually does anything, first I toggle the second bit at $ff820a, by an exclusive or operation, then wait a bit and toggle back. The rept, endr commands is a way to tell the assembler that the lines between these two commands should be repeated for so many times. This has no effect on the program when actually running, it's as though I'd written nop eight times in a row, but this is easier to read. Thus, I wait for 8 * 4 = 32 clock cycles between the synchronization changes.

After the top border has been killed, it's time to prepare to kill the bottom border. First it should be disabled, so it's not jumped to while I set it up, then the number of counts, in this case 228. If I'd only been interested in killing the bottom border, and not the top, this value would've been 199. Lastly, Timer B is started by putting the value 8 in $fffffa1b, meaning that Timer B goes into event count mode. Now, the value in $fffffa21 will decrement by one for each scan line. The vbl routine is then finished by restoring the registers and status register.

On to Timer B, first off, backup the registers that are used in the routine, to avoid bombs and other unpleasantries. I arrive in Timer B somewhere on 228:th scan line, and I want to be on the 229:th line when I kill the border. Timer B data changes exactly on the start of every scan line, so by checking for a change in that register, I'll know exactly when the change comes and I'm exactly at the beginning on the 229:th scan line and kill off the border; khazam! (note: if the top border is not killed, the numbers are 199:th and 200:th respectively)

The check for change in the register might be a bit tricky at first glance; I put the value of the register in d0, then I compare d0 with the value of the register, if those are equal, I branch back a step and do the process over. This is repeated until the value in Timer B changes, and d0 and Timer B will no longer hold the same value. Neat. Arriving on the 229:th scan line now, I just do as before; toggle PAL/NTFS, and finish off that border as well. I restore the backed up registers, tell the ST the interrupt is over and make a clean exit. All done; no top or bottom border.

It feels like this tutorial has been a lot of fact blurping, and painfully little understanding. Well, I guess you have to endure some things. Now that the borders are gone, we have gained some more pixels to work with obviously. From my gazing-hard-at-the-monitor-trying-to-see technique, I assume that the top border is 29 scan lines, and that the total visual spectra goes up to 320*270 pixels, meaning the bottom border is 41 scan lines.

There are lots of good ways to make use of Timer B, for instance, one can change the palette on every scan line, this means that you aren't limited to 16 colours a screen, but can with ease have 16 colours per scan line. In a game, it would be nice to have a status bar in the lower border, or upper for that matter, to leave the 320*200 "main area" uncluttered with such stuff. It would also be able to have that status bar in a different palette, making it very smooth. Another thing is the possibility to change resolution mid-screen, by doing this, you can have a medium resolution star filed in the upper part of the screen (star fields require few colours), and then change resolution to low and have, say a nice mountain formation on the bottom, which require more colours. Creativity is up to you!

Again, thanks to all people who support and encourage me. I got a mail from Bruno Padinha, who sent me the entire tutorial formatted very nicely. I've received mail from more people than I could have dreamed of, thank you all! Also, big thanks go out to all good people at #atariscne on IRC, who help me with various coding stuff.

perihelion of poSTmortem, 2002-06-01

"In strategy it is important to see distant things as if they were close and to take a distanced view of close things. It is important in strategy to know the enemy's sword and not to be distracted by insignificant movements of his sword. You must study this. The gaze is the same for single combat and for large-scale strategy."

- Book of Five Rings, by Miyamoto Musashi

Last edited 2002-07-06