The Atari ST M68000 tutorial part 12 – of controlling the puppets

Yep, here we go again, this time I think we'll have a nice little tutorial on our hands, not that big. It only concerns the workings of the joystick. It could've involved the mouse as well, but to be honest I haven't gotten the workings of the mouse down yet. The code will build heavily on the previous tutorial, since we are going to move a sprite around with the joystick, but you don't need to understand the sprite parts of the code to understand the workings of the joystick. If you don't know what a joystick is, or if you don't recognise the little sprite ship used in the sample source, you are not allowed to read further. Please stop this instant and browse the web for more generally related Atari information.

A while back, I thought the ST was so much cooler than your average PC, because with the ST, you just have to plug in a joystick and it works. With a PC, you have to install drivers and shit, and configure the exact joystick and generally mess around lots and perhaps even then it won't work or the program you want to run doesn't support your joystick. All in all inferior construction, or so I thought. Actually, with the ST, you also need to set up your own joystick driver. In fact, since you usually don't have a hard drive and the OS (operating system) doesn't have drivers for the joystick, every program needs it's own drivers for the joystick. Writing the joystick driver isn't at all difficult, but you have to have some working knowledge to do it.

There is a little 6301 processor inside the Atari ST, which takes care of the keyboard, the mouse and the joystick. It even has a real time clock. This cute little chip is sometimes referred to as the IKBD, for Intelligent KeyBoarD. It might be fun to know that the IKBD has 4K (4096 bytes) of ROM memory, and 128 bytes of RAM. ROM stands for Read Only Memory, and as it says, it's memory that can't be altered, RAM is Random Access Memory and it is that which we usually mean by memory. The 128 bytes of RAM on the IKBD are only used as a temporal storage area. The reason for having a separate chip altogether taking care of the keyboard, mouse and joystick is that those actions won't burden the main processor (the 68000, the one we've been programming so far in these tutorials). Instead, we can poll the IKBD as we choose, or tell it to report stuff in any way we choose, and just let the IKBD worry about the details.

Our mission therefore is clear: we must find a way to make the IKBD report the status of the joystick, and also find a way to read that status in some way. When that is accomplished, we can use the sprite routine from the previous tutorial as it is, with only a change in the move_sprite subroutine. The new subroutine will update the X and Y coordinates in accordance with the joystick status instead of just moving it about.

Trap function 25 of the XBIOS will allow us to send commands to the IKBD. However, unlike other trap calls, the input data is a pointer to a string of data. The text file IKBD.TXT may seem very sketchy and difficult to understand, but it does contain a list of all the possible commands that you can send to the IKBD, taking a look inside it, we see function $14. IKBD command $14 will report joystick status every time the joystick is changed. All well and good, this is how we set it up.

```
		move.l	#joy_on,-(a7)		pointer to IKBD instructions
```

```
            move.w      #0,-(a7)                instruction length - 1
            move.w      #25,-(a7)               send instruction to IKBD
            trap        #14
            addq.l      #8,a7

joy_on      dc.b        $14
```

The first parameter is a pointer to the address which contains the commands, the second parameter is the length in byte of the command list minus one, in this case zero. Then the function number, a trap calling XBIOS and a normal stack clean up. Sure, so now the joystick reports information, but where does the information go? Well, actually we need to write our own routine to read the joystick information.

Every time the joystick sends information, there is a jump to an address with instructions of what to do with this data, compare this with the timers from tutorial 9. Also, as with the timers, we will hook up our own routine to read the joystick. With trap function 34 of the XBIOS, the IKBD returns a list of all its vectors. The address of the IKBD vectors is put in d0. The joystick report vector is at offset 24, so by putting our own joystick routine at the address pointed to by d0 +24, we have effectively hooked up our own joystick routine.

```
            move.w      #34,-(a7)
            trap        #14
            addq.l      #2,a7                   return IKBD vector table

            move.l      d0,ikbd_vec                         store IKBD vectors
address
            move.l      d0,a0                   a0 points to IKBD vectors
            move.l      24(a0),old_joy          backup old joystick vector

            move.l      #read_joy,24(a0)        input our joystick vector

read_joy
            nop                                 so far, we don't know what to do
            rts                                 note, rts, not rte

ikbd_vec    dc.l                                old IKBD vector storage
old_joy     dc.l                                old joy vector storage
```

Straightforward, first get the address of the IKBD vectors. Store it for future restoration. Then put the address in a0 so that a0 points to the IKBD vectors, backup the old joystick vector which is found at offset 24, and input our own joystick routine. By the way, the mouse vector is at offset 16. With the help of this and the information given on the other IKBD commands in the IKBD.TXT file, you should be able to setup your own mouse routine as well.

The joystick routine ends with an rts, nothing else, and may not take more than 1/100 of a second (half a VBL, more than enough time really). What happens now is that each time the joystick status is changed, the ST will jump to our joystick routine. Once there, a0 will point to three bytes in memory which contain the status of the joysticks.

The first of these bytes is a header telling us which joystick it was that did something. The byte will contain $FE if joystick 0 did something, and $FF if it was joystick 1 (meaning the last bit represents either joystick 0 or joystick 1). Remember, joystick 0 is the joystick port shared with the mouse, and joystick 1 is the port exclusively for joysticks. The next two bytes contain the actual information for the joysticks. The first one holds status for joystick 0, and the other one for joystick 1. The data has this structure

F000RLDU
76543210
(F = fire, R = right, L = left, D = down, U = up)

So if bit 7 is set, the fire button was pressed, if bit 0 is set, the joystick is moved up, if bit 0, 2 and 7 are set, the joystick is moved up-right while the fire button is being pressed. Real simple. Here’s a joystick routine that will simply store the joystick data in memory, two different variables could have been used instead of course (but this is good practice on addressing modes).

```
read_joy
* executes every time joystick information is changed
            move.b      1(a0),joy               store joy 0 data
            move.b      2(a0),joy+1                     store joy 1 data
            rts

joy         ds.b        2                       storage for joystick data
```

That’s it! Well, almost. We must restore our poor system, for one thing, it would be good to turn the mouse back on :) When we turn on the joystick, the mouse is turned off. In order to turn it on, we send command $08 to the IKBD, to put the mouse in relative report mode, which would probably be the default mode for the mouse then. While we’re at it, might be good to restore the joystick vector as well. For the curious lot out there, mus is Swedish for mouse, and it’s a suitable short form for mouse as well.

```
            move.l      #mus_on,-(a7)           pointer to IKBD instruction
            move.w      #0,-(a7)                length of instruction - 1
            move.w      #25,-(a7)               send instruction to IKBD
            trap        #14
            addq.l      #8,a7

            move.l      ikbd_vec,a0                     a0 points to old IKBD
vectors
            move.l      old_joy,24(a0)          restore joystick vector

mus_on      dc.b        $08
dc.l        ikbd_vec                            IKBD vector storage
dc.l        old_joy                             old joy vector storage
```

Two other commands of the IKBD that might be good to know about are $1a, which turns off the joystick, and $12 which turns off the mouse. Let's say we want to be on the really safe side and not only turn on joystick reporting but also turn off mouse reporting, it would look thusly

```
            move.l      #joy_on,-(a7)           pointer to IKBD instructions
            move.w      #1,-(a7)                instruction length - 1
            move.w      #25,-(a7)               send instruction to IKBD
            trap        #14
            addq.l      #8,a7

joy_on      dc.b        $14,$12
```

Note how the extra parameters are just appended to the command list, and the update of the instruction length parameter to reflect the new command list length. Here comes the source of the program, hold on!

```
            jsr         initialise

* pre-shifting sprite
            move.l      #spr_dat,a0                         original sprite data
            add.l       #34,a0                  skip palette
            move.l      #sprite,a1              storage of pre-shifted sprite

            move.l      #32-1,d0                32 scan lines per sprite
first_sprite
            move.l      (a0)+,(a1)+                         move from original to
pre-shifted
            move.l      (a0)+,(a1)+
            move.l      (a0)+,(a1)+
            move.l      (a0)+,(a1)+                         32 pixels moved
            add.l       #8,a1                   jump over end words
            add.l       #144,a0                 jump to next scan line
            dbf         d0,first_sprite
* the picture sprite has been copied to first position in pre-shift

            move.l      #sprite,a0              point to beginning of storage
area
            move.l      #sprite,a1              point to beginning of storage
area
            add.l       #768,a1                 point to next sprite position

            move.l      #15-1,d1                15 sprite positions left
positions
            move.l      #32-1,d2                32 scan lines per sprite
line
            move.l      #4-1,d3                 4 bit planes
plane
            move.w      (a0),d0                 move one word
```

```
            roxr        #1,d0               pre- shift
            move.w      d0,(a1)             put it in place


            move.w      8(a0),d0            move one word
            roxr        #1,d0               pre- shift
            move.w      d0,8(a1)            put it in place

            move.w      16(a0),d0           move one word
            roxr        #1,d0               pre- shift
            move.w      d0,16(a1)           put it in place

            add.l       #2,a0               next bit plane, also clears X flag
            add.l       #2,a1               next bit plane

            dbf         d3,plane

            add.l       #16,a1              next scan line
            add.l       #16,a0              next scan line

            dbf         d2,line

            dbf         d1,positions
* pre- shift of sprite done, all 16 sprite possitions saved in sprite


* pre- shifting mask
            move.l      #spr_dat,a0
            add.l       #34+160*32,a0       skip palette and sprite
            move.l      #mask,a1            load up mask part

            move.l      #32- 1,d0           32 scan lines per sprite
first_mask
            move.l      (a0)+,(a1)          move from original to pre-
shifted
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      #$ffffffff,(a1)+    fill last two words...
            move.l      #$ffffffff,(a1)+    ... with all 1's

            add.l       #144,a0             jump to next scan line
            dbf         d0,first_mask
* the picture mask has been copied to first position in pre- shift
```

```
		move.l	#mask,a0		point to beginning of storage
area
		move.l	#mask,a1		point to beginning of storage
area
		add.l	#768,a1			point to next mask position

		move.l	#15-1,d1		15 sprite positions left
positions_mask
		move.l	#32-1,d2		32 scan lines per sprite
line_mask
		move.l	#4-1,d3			4 bit planes
plane_mask
		move.w	(a0),d0			move one word
		roxr	#1,d0			pre-shift
		or.w	#%1000000000000000,d0		make sure most
significant bit set
		move.w	d0,(a1)			put it in place


		move.w	8(a0),d0		move one word
		roxr	#1,d0			pre-shift
		move.w	d0,8(a1)		put it in place

		move.w	16(a0),d0		move one word
		roxr	#1,d0			pre-shift
		move.w	d0,16(a1)		put it in place

		add.l	#2,a1			next bit plane
		add.l	#2,a0			next plane, clears X flag (bad)

		dbf	d3,plane_mask

		add.l	#16,a1			next scan line
		add.l	#16,a0			next scan line

		dbf	d2,line_mask

		dbf	d1,positions_mask
* pre-shift of mask done, all 16 sprite possitions saved in mask


		movem.l	bg+2,d0-d7
		movem.l	d0-d7,$ff8240

		move.l	#bg+34,a0		pixel part of background
		move.l	$44e,a1			put screen memory in a1
		move.l	#7999,d0		8000 longwords to a screen
pic_loop
		move.l	(a0)+,(a1)+				move one longword
to screen
```

```
		dbf		d0,pic_loop				background painted

		jsr		save_background		something in restore buffer

** joy code
		move.w	#34,-(a7)
		trap		#14
		addq.l	#2,a7				return IKBD vector table

		move.l	d0,ikbd_vec				store IKBD vectors
address
		move.l	d0,a0				a0 points to IKBD vectors
		move.l	24(a0),old_joy		backup old joystick vector

		move.l	#read_joy,24(a0)		input my joystick vector

		move.l	#joy_on,-(a7)		pointer to IKBD instructions
		move.w	#0,-(a7)			instruction length - 1
		move.w	#25,-(a7)			send instruction to IKBD
		trap		#14
		addq.l	#8,a7
** end joystick init

		move.l	$70,old_70				backup $70
		move.l	#main,$70			put in main routine

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

		move.l	old_70,$70				restore old $70

** joy code
		move.l	#mus_on,-(a7)		pointer to IKBD instruction
		move.w	#0,-(a7)			length of instruction - 1
		move.w	#25,-(a7)			send instruction to IKBD
		trap		#14
		addq.l	#8,a7

		move.l	ikbd_vec,a0				a0 points to old IKBD
vectors
		move.l	old_joy,24(a0)		restore joystick vector
** end shut down


		jsr		restore

		clr.l		-(a7)
		trap		#1				exit
```

```
main
            movem.l     d0-d7/a0-a6,-(a7)       backup registers

            jsr         restore_background
            jsr         move_sprite
            jsr         save_background
            jsr         apply_mask
            jsr         put_sprite

            movem.l     (a7)+,d0-d7/a0-a6       restore registers

            rte

move_sprite
* updates x and y coordinates according to joystick 1
* if fire button pressed, add 1 to colour 0
            move.b      joy+1,d0                check joystick 1

            cmp         #128,d0                 fire
            blt         no_fire
            add.w       #$001,$ff8240
            and.b       #%01111111,d0           clear fire bit

no_fire

            cmp.b       #1,d0                   up
            beq         up
            cmp.b       #2,d0                   down
            beq         down
            cmp.b       #4,d0                   left
            beq         left
            cmp.b       #8,d0                   right
            beq         right
            cmp.b       #9,d0                   up-right
            beq         up_right
            cmp.b       #10,d0                  down-right
            beq         down_right
            cmp.b       #6,d0                   down-left
            beq         down_left
            cmp.b       #5,d0                   up-left
            beq         up_left
            bra         done
up
            sub.w       #1,y_coord
            bra         done
down
            add.w       #1,y_coord
            bra         done
left
            sub.w       #1,x_coord
```

```
		bra	done
right
		add.w	#1,x_coord
		bra	done
up_right
		sub.w	#1,y_coord
		add.w	#1,x_coord
		bra	done
down_right
		add.w	#1,y_coord
		add.w	#1,x_coord
		bra	done
down_left
		add.w	#1,y_coord
		sub.w	#1,x_coord
		bra	done
up_left
		sub.w	#1,y_coord
		sub.w	#1,x_coord
		bra	done
done

* avoid going outside screen
		cmp	#319- 32,x_coord
		blt	x_right_ok
		move.w	#319- 32,x_coord
x_right_ok

		cmp	#0,x_coord
		bgt	x_left_ok
		move.w	#0,x_coord
x_left_ok

		cmp	#199- 32,y_coord
		blt	y_low_ok
		move.w	#199- 32,y_coord
y_low_ok

		cmp	#0,y_coord
		bgt	y_high_ok
		move.w	#0,y_coord
y_high_ok
		rts


read_joy
* executes every time joystick information is changed
		move.b	1(a0),joy		store joy 0 data
		move.b	2(a0),joy+1			store joy 1 data
		rts
```

```
apply_mask
* applies the mask to the background
            jsr         get_coordinates
            move.l      #mask,a0
            mulu        #768,d0                 multiply position with size
            add.l       d0,a0                   add value to mask pointer

            move.l      #32- 1,d7               mask is 32 scan lines
maskloop
            rept        6                       mask is 6*4 bytes width
            move.l      (a0)+,d0                mask data in d0
            move.l      (a1),d1                 background data in d1
            and.l       d0,d1                   and mask and picture data
            move.l      d1,(a1)+                move masked picture data to
background
            endr
            add.l       #136,a1                 next scan line
            dbf         d7,maskloop

            rts


put_sprite
* paints the sprite to the screen
            jsr         get_coordinates
            move.l      #sprite,a0
            mulu        #768,d0                 multiply position with size
            add.l       d0,a0                   add value to sprite pointer

            move.l      #32- 1,d7               sprite is 32 scan lines
bgloop
            rept        6                       sprite is 6*4 bytes width
            move.l      (a0)+,d0                sprite data in d0
            move.l      (a1),d1                 background data in d1
            or.l        d0,d1                   or sprite and background data
            move.l      d1,(a1)+                move ored sprite data to
background
            endr
            add.l       #136,a1
            dbf         d7,bgloop

            rts


save_background
* saves the background into bgsave
            jsr         get_coordinates
            move.l      #bgsave,a0
```

```
            move.l      #32- 1,d7               sprite is 32 scan lines
bgsaveloop
            rept        6                       sprite is 6*4 bytes width
            move.l      (a1)+,(a0)+                         copy background to
save buffer
            endr
            add.l       #136,a1                 next scan line
            dbf         d7,bgsaveloop

            rts


restore_background
* restores the background using data from bgsave
            jsr         get_coordinates
            move.l      #bgsave,a0

            move.l      #32- 1,d7               sprite is 32 scan lines
bgrestoreloop
            rept        6                       sprite is 6*4 bytes width
            move.l      (a0)+,(a1)+                         copy save buffer to
background
            endr
            add.l       #136,a1                 next scan line
            dbf         d7,bgrestoreloop

            rts


get_coordinates
* makes a1 point to correct place on screen
* sprite position in d0.b
            move.l      $44e,a1                 screen memory in a1
            move.w      y_coord,d0                          put y coordinate in
d0
            mulu        #160,d0                 160 bytes to a scan line
            add.l       d0,a1                   add to screen pointer
            move.w      x_coord,d0                          put x coordinate in
d0
            divu.w      #16,d0                  number of clusters in low, bit in
high
            clr.l       d1                      clear d1
            move.w      d0,d1                   move cluster part to d1
            mulu.w      #8,d1                   8 bytes to a cluster
            add.l       d1,a1                   add cluster part to screen
memory
            clr.w       d0                      clear out the cluster value
            swap        d0                      bit to alter in low part of d0

            rts
```

```
		include		initlib.s


		section	data
x_coord		dc.w		150
y_coord		dc.w		80

spr_dat		incbin		SHIP.PI1
bg		incbin		XENON.PI1
old_70		dc.l		0

joy_on				dc.b		$14
mus_on				dc.b		$08
ikbd_vec			dc.l		0
old_joy				dc.l		0


		section	bss
sprite		ds.l		3072		32/2+8*32 bytes * 16 positions / 4 for long
mask		ds.l		3072		same as above
bgsave		ds.l		192		32/2+8*32 bytes / 4 for long
joy		ds.b		2
```

Yup, another long source code. There are big similarities between the sprite tutorial though, since we're basically doing the same thing. The new things are of course the joystick on and off, which are located between the "* joy code" comments, after the pre- shiftings. Nothing to say there that hasn't been said before. Same with the joystick routine. The move_sprite routine is all new and deserves attention.

It begins by moving the joystick data to d0. In this case, I only check joystick 1. First I begin by checking for the fire button, this is done by seeing if d0 contains a number larger than or equal to 128. If the fire button is pressed, the 8th bit (bit 7, start counting from 0 and from the rightmost bit) in the joystick status byte is set which means that the byte will hold a value equal to or higher than 128, since %10000000 = 128. Then I clear out the fire bit so that it won't bother me anymore.

Next I check for joystick movement. This is done by using the same method as above. For example, if the joystick is down- left, then bit 1 and 2 are set, meaning the byte will hold value %00000110 = 6. This is the reason for clearing out the fire bit above. If it hadn't been cleared, the number would be either 6 or 128 + 6 = 134 for down- right. So just run through all 8 directional checks to see if any bits are set, if they are not, I just branch right away to done. If this branch hadn't been there, the program would just continue and execute the code associated with joystick up if the joystick wasn't moved at all. An early bug that caused me some confusion.

After the coordinates have been changed accordingly, I also check to see that the sprite isn't out of bounds, since this could cause a crash and be generally stupid in all kinds of ways. So just check if the coordinates are right, and if they’re not, reset them to the closest correct value. If you want a speedier ship, just increase the speed accordingly, adding more than one to the coordinates, and also remember to include this in the boundary check, just as the sprite.

Some of you will probably notice that the ship itself is not 32 scan lines, although I treat the sprite as such. This has the effect of the ship never reaching all the way down the screen, since there is some black space worth of sprite data. This could be easily fixed of course, but I didn’t. Also, two ships moving might be nice, at first I considered having both the Xenon 2 ship and the Xenon 1 ship side by side, controlled by two joysticks, but I decided to keep it simple. However, there should be no big trouble incorporating that, and changing the fire button perhaps to morph the Xenon 1 ship.

Having two sprites is no harder than having one sprite, the only thing you have to think about is the order of painting the sprites, the ones painted first will be painted over by the ones that come next. Yet another cool thing is to change the look of the sprite as you move it, like in the real Xenon game, they have the ship tilted sideways and generate rocket fire when it moves, all you need is a flag to know which state the ship is in and change the sprite address accordingly.

This means having a sprite picture with not just one ship, but the ship tilted in directions and with rocket flames, all in all lots of pictures. All of these sprites will of course fit in one degas picture, so all you need is the correct offset into this picture depending on what “mode” the sprite is in. Compare this to the way we address the sprite mask, only in this case it’s a different sprite (or different look of the sprite, depending on how you see it).

Now you have the tools needed to create a game, or even a demo for that matter: now go to it! Even though there is still much to learn, the basics have been covered, all but one thing: music and sound. It is my hope that this will come soon. But you don’t have to worry about that for now, code away and the music will be easily incorporated at a later stage.

Usually, you just hook up the music in your VBL routine. On the Dead Hackers Society page, http://dhs.nu, there are two chip editors (at least) with instructions on how to play the generated music in assembler; Edsynth and the XLR8. Go take a look at them if you’re curious, there should be no trouble understanding the code.

perihelion of poSTmortem, 2002-07-13

"I love the smell of napalm in the morning... it smells like victory"

- Apocalypse Now