The Atari ST M68000 tutorial part 14 – of using the gramophone

Wow, it really was a long time since the last tutorial. I've had more and more to do in school and other things have popped up, maybe I just needed a break too. Now I really feel up to writing again, thanks to some encouragement on the Atari forum (http://www.atari-forum.com).

This here tutorial will be the follow up of the previous one, in which I promised to tell you how to play the .ym files of the ST-sound format from Arnaud Carré. It will be quite easy and a bit of a soft start actually. The focus lies not so much on the code, but how to find and apply knowledge.

Like I always say, I am no musician, neither am I an artist, so therefore, I need to rip stuff or have it made for me. I have loads of .ym files on my PC, which can be played by using a plugin for Winamp. Wouldn't it be nice to be able to use this wealth of music? Yes it would, I wonder how that can be achieved, here's how.

In order to use the files, we need information on the file format. See tutorial 6 for a quick refresh on files if that's needed. Load up a good search engine in the browser, I used Google (http://www.google.com). Now we want to find info on the .ym file format, so a search string of "ym file format" would seem appropriate. Would you look at that, the first find seems good, taking us to http://leonard.oxg.free.fr/ymformat.html. Quickly browsing the side, we judge it seems to hold what we need. We also discover the file format is freeware, so there's no need to worry about the cops.

Hum hum, there seem to be different versions of the file format, didn't know that … hum hum, this information only applies to YM6, the latest version. "So YM6 is just a register dump file", this is an important key, it tells us how the file format works. It seems that a .ym file is simply a dump of the data used to play a song, but that's not enough, we need to know how the data is organized. Reading on … Ah, .ym files are packed using LHA, so that's why they are so small. Using the freeware UltimateZip (http://www.ultimatezip.com), a .ym file can be unpacked, or any other LHA packer, but UltimateZip is my choice of program.

Reading ever further down the page … ah, here it comes. The .ym file contains 16 bytes of data for each frame, interleaved. Sure, the sound chip has 16 registers, so by just putting the data into the registers of the sound chip, music should be played. Lastly, there's some info on the file header. Some files have headers that tell of important information for the rest of the file, here for example, it's nice to know how long a song actually is. There's some talk about didigrums and so, that will not be covered in this tutorial and you are welcome to explore it yourself.

So, now we have all the information we need, we just have to structure it and go through it. Load up the included .ym file JAMBLV1.YM in your favourite hexeditor. It's also possible to put it in an otherwise empty source file, assemble it and go into the debugger like this

```
nop
incbin jamblv1.ym
```

It seems that every program starts with two bytes of data that would overwrite the data in jamblv1.ym, that's what the nop is there for. By

hitting tab once to get into the memory window, you can use the arrow keys to scroll up and down in the jamblv1.ym file. Now we'll traverse the file and see if it corresponds to the information we have on what the file should look like. It starts with the values $59, $4d and $21, which identifies the file as a YM6 file. When interpreted as ASCII (numbers to letters), these numbers become the letters Y, M and !. Next follows a test string, "LeOnArD!", all good so far.

After the initial check- things comes the interesting information, a long (4 bytes) that tells us the number of frames in the file. In this case, it's a value of $0000bea, which corresponds to 3050 in decimal. Note that I wrote out the leading two bytes that for now only contain zeros, but they are important to count otherwise you'll get lost. What does this mean exactly? Well, frame of music is just like a frame of graphics, the ST usually operates at 50 Hertz which equals 50 frames per second. So we divide 3050 by 50 and get the value 61, indicating the tune should be 1:01 long. Load it up in Winamp to test, yep, seems to be right.

Next comes four bytes of song attributes, that I have no idea what it is, but zero seems to be a safe value, and two bytes of digidrums, which are also zero. Some files have a song attribute of one, and they seem to work fine to. You'll have to experiment with these yourself if you find songs that should use digidrums, or mail LeOnArD! Another uninteresting value, $001e8480, or 2000000, which seems to indicate this is indeed an Atari tune. Then two bytes, telling us the tune is operating at a frequency of 50 Hz. Lastly an additional six bytes of zero data.

Right, you with me so far? It's just a question of slowly going through the file and check that everything is in order and corresponds to the information we have. Of course it is in order, otherwise the file wouldn't work in Winamp, but I want to make sure for myself. Now comes some text again, according to Leonard's page, these are the song name, author name and song comment.

The data is in null terminated string format. This means the strings can be variable in length, and ends with the value zero. Quite true, after each little string, we can see zeroes shining through. After these strings, the real sound data begins, also of unknown length. However, since we know that there are 3050 frames of data, and each frame holds 16 bytes of sound data, there are 3050 * 16 = 48800 bytes of data here, this calculation also seems correct since this is roughly the file size. At the end, there are also four bytes forming the string "End!".

So what do we really need here? Two things, the number of frames, to know how long the music file is, so we know when to terminate play, or loop the song, and the start address of the music data. We know the address of the number of frames, so that's easy to just store in a variable. Getting to the music data is trickier, since we don't know exactly where it is. Sure, we can hexedit the file and then hardcode the address into the program, but a more general way of finding the music start data would be nice, so that we easily can play many different .ym files without having to check the start address of the sound data for each file.

What we want is to get to the end of the three text strings, because this is where the sound data begins (if you don't have any digidrums). To do this, we put ourselves at the beginning of the text field, which always start at the same place, then we check each byte for a zero, since this means the end

of a string, and do this three times. In so doing, we will have passed by all the three text strings, like so

```
            move.l      #ym_file,a0                 start of ym file
            move.l      12(a0),frames       store number of frames
            add.l       #34,a0              beginning of text

song_name
            cmp.b       #0,(a0)+            search for 0
            bne         song_name
comment
            cmp.b       #0,(a0)+            search for 0
            bne         comment
song_data
            cmp.b       #0,(a0)+            search for 0
            bne         song_data
            move.l      a0,music            skipped 3 zero, store address
```

Now we have the length of the tune in frames, and the start address for the sound data in music. What was that about interleaved data? The thing is, that many registers of the sound chip are all zero. In order to compress better, it would be nice to have all these zeros in one long row. Therefore, the data is not presented in the order it's supposed to be inserted in the sound chip, rather, the data is presented one full register after another. Thus, in our file, there is 3050 bytes of register 0 data, then 3050 bytes of register 1 data and so on.

When we put the sound data in the yammy, we have to add the number of frames for each input. In this way, we will first input data from register 0, then we skip the number of frames to reach the data for the next register and so on. Here's the entire code, the code for the VU bars has already been discussed and is only included here for fun, so there is very little new code

```
            jsr         initialise

            move.l      #palette,a0                 pointer to palette
            movem.l     (a0)+,d0-d7                 palette in d0-d7
            movem.l     d0-d7,$ff8240       apply palette

            move.l      #ym_file,a0                 start of ym file
            move.l      12(a0),frames       store number of frames

            add.l       #34,a0              beginning of text

song_name
            cmp.b       #0,(a0)+            search for 0
            bne         song_name
comment
            cmp.b       #0,(a0)+            search for 0
            bne         comment
```

```
song_data
            cmp.b     #0,(a0)+              search for 0
            bne       song_data

            move.l    a0,music              skipped 3 zero, store address

            move.l    $70,-(a7)             backup $70
            move.l    #main,$70             start main routine
            move.w    #7,-(a7)
            trap      #1
            addq.l    #2,a7                 wait keypress
            move.l    (a7)+,$70             restore $70

            jsr       restore

            clr.l     -(a7)
            trap      #1                    exit


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

            move.l    music,a0              pointer to current music data
            moveq.l   #0,d0                 first yammy register
play
            move.b    d0,$ff8800                    write to register
            move.b    (a0),$ff8802                  write music data
            add.l     frames,a0             jump to next register in data
            addq.b    #1,d0                 next register
            cmp.b     #16,d0                see if last register
            bne       play                  if not, write next one

            addq.l    #1,music              next set of registers
            addq.l    #1,play_time                  1/50th second play
time

            move.l    frames,d0
            move.l    play_time,d1
            cmp.l     d0,d1                 see if at end of music file
            bne       no_loop
            sub.l     d0,music              beginning of music data
            move.l    #0,play_time                  reset play time
no_loop
            jsr       vu_bars               paint the vu bars

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


* put in VU bars
```

```
vu_bars
            move.l      $44e,a0                 get screen address
            add.l       #160*199-(15*2)*160,a0          bottom area of screen
            move.l      #bar,a1                 point to bar colours

            rept        15                      15 max volume
            movem.l     (a1)+,d0-d1                     VU bar colour in d1-
d2
            movem.l     d0-d1,(a0)              first VU bar
            addq.l      #8,a0                   next VU bar
            movem.l     d0-d1,(a0)              second VU bar
            addq.l      #8,a0                   next VU bar
            movem.l     d0-d1,(a0)              third VU bar
            add.w       #320-16,a0                      two lines down, two
bars left
            endr

* delete VU bars depending on volume
            move.l      $44e,a0                 get screen address
            add.l       #160*199-(15*2)*160,a0          bottom area of screen

            moveq.l     #0,d0                   clear d0
            move.b      #8,$ff8800                      chanenl a volume
            move.b      $ff8800,d0                      put volume in d0
            jsr         del_bar

            moveq.l     #0,d0                   clear d0
            move.b      #9,$ff8800                      channel b volume
            move.b      $ff8800,d0                      put volume in d0
            add.l       #8,a0                   next VU bar
            jsr         del_bar

            moveq.l     #0,d0                   clear d0
            move.b      #10,$ff8800                     channel c volume
            move.b      $ff8800,d0                      put volume in d0
            add.l       #8,a0                   next VU bar
            jsr         del_bar

            rts


del_bar
* screen address of top line in a0
* volume in d0, gets detroyed
            move.l      a0,-(a7)                backup a0
            move.l      a1,-(a7)                backup a1
            and.b       #%1111,d0               keep only lowest 4 bits

            move.l      #delete,a1              beginning of delete blocks
            mulu        #12,d0                  length of one delete block
```

```
            add.l       d0,a1               skip some delete instructions
            jmp         (a1)                jump to correct delete position

delete
            rept        15
            clr.l       (a0)                clear two bit planes
            clr.l       4(a0)               clear two bit planes
            add.l       #320,a0             hop two lines down
            endr

            move.l      (a7)+,a1            restore a1
            move.l      (a7)+,a0            restore a0
            rts


            include     initlib.s


            section data
music                   dc.l        0       address of music data
frames                  dc.l        0       how many frames of music data
play_time   dc.l        0                   how many VBL's has elapsed

ym_file     incbin      jamblv1.ym

bar
* colour data for each line of VU bar
            dc.w        $00ff,$00ff,$00ff,$00ff
            dc.w        $0000,$00ff,$00ff,$00ff
            dc.w        $00ff,$0000,$00ff,$00ff
            dc.w        $0000,$0000,$00ff,$00ff
            dc.w        $00ff,$00ff,$0000,$00ff
            dc.w        $0000,$00ff,$0000,$00ff
            dc.w        $00ff,$0000,$0000,$00ff
            dc.w        $0000,$0000,$0000,$00ff
            dc.w        $00ff,$00ff,$00ff,$0000
            dc.w        $0000,$00ff,$00ff,$0000
            dc.w        $00ff,$0000,$00ff,$0000
            dc.w        $0000,$0000,$00ff,$0000
            dc.w        $00ff,$00ff,$0000,$0000
            dc.w        $0000,$00ff,$0000,$0000
            dc.w        $00ff,$0000,$0000,$0000
            dc.w        $00ff,$0000,$0000,$0000

palette
            dc.w        $000,$023,$023,$024,$024,$025,$026,$026
            dc.w        $027,$027,$227,$327,$427,$527,$627,$727
```

I start off with a normal setup, then read in the music data as described previously and start the main routine. The main routine here has

the actual routine for playing the tune, and the rest of the code is just VU bars.

First, make a0 point to the current music data, this is somewhere in the music file (on a number of frames boundary), then put the yammy register number in d0. The real routine for actually getting the sound data into the yammy is very compact. D0 holds the number of the register to manipulate, putting that in $ff8800 lets us manipulate the register in question, then I just put in the music data. After that, it's a question of adding the number of frames to the music pointer, in order to point to the next register. Increment d0 to point to the next register, and do this 16 times, one time for each register. If you don't remember about the sound chip, recheck tutorial 13.

Next I increment the music pointer, so that it points to the beginning of the next sound data set, and increase the number of played frames by one. The last part of the main routine checks to see if the number of played frames equals the number of frames, if this is so, I subtract the number of frames from the music pointer. This makes the music pointer point to the beginning of the music data again. The play time also needs to be reset of course, finally, a jump to the VU routine, just for the visual effect. Not to complex when you think about it, actually, I managed to get it right on the first compile ... almost, I had a slight offset error.

The routine should work for any and all YM6 version files without anything fancy (digidrums etc), and perhaps even with some fancy stuff. I don't really know. Unfortunately it will not play any other ym versions, you'll have to work that out yourself. In order to get any music you want from any Atari source, you can use SainT to record the music in .ym format, it's that simple.

With this routine, you could make yourself a .ym file player for the Atari. As the program is now, it's really crappy, there is no error reporting of any kind for starters. Perhaps some tunes really are in 60 Hertz, then they would play wrongly, or perhaps the file is something other than YM6 probably resulting in a crash. You should add some error reporting yourself.

One nice thing to do with this is to just hook up the music to the VBL, then drop out of the program (not waiting for a keypress nor restoring the VBL). The music will still be playing and you can go on coding. This is very unstable though, and doing this in the GEM desktop will probably get you an immediate crash, doing this in Devpac will probably get you a crash when you compile anything. It's just an idea to get you going.

perihelion of poSTmortem, 2003-02-22

"They fought like warrior poets. They fought like Scotsmen and won their freedom forever."

- Braveheart