The Atari ST MC68000 Assembly Language Tutorial

perihelion of poSTmortem

edited by bp

1st Edition (v0.84)

August 28th 2004

Table of Contents

0 Foreword

This document is a compilation and formatting of a set of tutorials created by perihelion with the intention to start people in the “art and science” of coding the Atari ST series of computers in assembly, one of the more popular programming languages especially for games, demos and other hardware-intensive applications. He welcomes any and all feedback at andreaswahlin@bredband.net .

For the sake of convenience, some very useful texts were added as appendixes, and due credit was given where possible. There are still some authors lacking for a few, so please, if you know who has written any of the un-credited appendixes, send an e-mail to bruno@atari.st .

While many people contributed with comments, suggestions, criticism and general incentive, some have been of special help, so perihelion would like to thank:

- Lars Lindblad, for suggesting it in the first place.
- Bruno Padinha (bp, http://stgameslist.atari.org), for critique and assistance extraordinaire.
- Marten Maartens (ST Graveyard, http://www.atarilegend.com/), for nice words and providing me with a canvas.
- Torsten Keltsch (mOdmate/Checkpoint, http://membres.lycos.fr/abrobecker/STart/mOd/mOd.html), for support in my early times.

1 On The Theory Behind Programming

"Conan, what is best in life?"

"To crush your enemies, see them driven before you, and to hear the lamentations of their women."

- Conan the Barbarian

Hi everybody! This is perihelion of poSTmortem (aka Andreas Wahlin) writing. Me and Aldebaran (aka Lars Lindblad) have just started this little project of ours: the demo group poSTmortem. As a first step towards actually doing something with the ST, we thought that writing a tutorial might be good. In this way, we can teach ourselves and you (whoever is listening).

This tutorial is aimed at people who, like ourselves, want to learn to code assembler for the Atari ST. What? You say, this is the year 2002, why on earth would you want to learn how to code, for one thing, assembler, and assembler for the Atari? You must be crazy. Well, you might think the Atari is dead, but we say it survived itself, it has risen again, it is: poSTmortem (totally lame, right?). You don't need any programming skills, although it might help since learning to code from assembly language is probably quite suicidal in a pedagogical view. I'll try to cover the basics of general programming and setting you up in this tutorial, and in part two we will do some rather simple program to get things started. I will however, assume that you have some basic skills in Atari management, like file copying and so on. Also, I will assume you have a real Atari, emulating might work fine, but nothing beats the real thing.

What is programming? Programming is the art and science of making the computer do what you want it to do. BTW, programming is also known as coding, I will use these two terms somewhat mixed probably. So, how do we do that? By telling the computer what to do, and then do it. Since you don't have shit for brains, you will know that when you double click on a .prg file (known as .exe files on a PC), the computer will do stuff, like running a game. So, what if we could create our own .prg files… Yes, we can do that, this is where you'll learn how!

Every computer has a memory, this is where the number on your Atari is derived from, 520 have half a Meg (short for Mega Byte) of ram, and 1040's have one Meg. Your usual PC these days have about 256 megs of ram, Bill Gates once said "nobody needs more than 640 K ram (about the memory amount of an Atari 520). If these numbers confuse you, do not worry, they aren't important right now. The memory is very temporary; it gets wiped out every time you turn off your Atari, unlike diskettes (thank God). When you run a program, the computer loads the program into memory, and then executes (follows the instructions given by the program).

Every area of the memory has an address, so you know where you are. You can think of these addresses as normal street addresses, but perhaps it would be better if you thought of them more as page indexes in a book. Every page is filled with information on how to act. So, let's assume that we have a monitor capable of displaying one of two colours, either black or white. The memory address $20 holds this colour. 1 means black, 0 white. If memory looks like this, we get a black screen.

Address	Value
...	
$19	12
$20	1
$21	67
...	
(each address can hold a number between 0 and 255)	

To understand a bit better about memory, because this is very important, we expand our little example. There is an area in the memory reserved for the user. This area is just for information storage, and does not affect the hardware. Let's say that the monitor is capable of displaying text as well, the text is also either black or white as previously stated, and the information on the text colour is to be found at $21. Further, the address $22 holds a pointer to the text to be displayed. Pointer? Argh! Well, it's really quite easy, a pointer is a reference to another part of the memory. Show first, talk later.

Memory	Value
…	
$19	12
$20	1
$21	0
$22	101
...	
$101	Hello World!
$200	DOH!
...	

In this example, the "user memory" begins after position $100. All address positions $0 - $100 do something with the hardware in some way, but after that, it's just storage space. With the above values, and given our premise, the text "Hello World!", will be displayed in white text on black background. Let's say we were to change the values to this instead.

Memory	Value
…	
$19	12
$20	1
$21	1
$22	200
...	
$101	Hello World!
$200	DOH!
...	
(changed values at $21 and $22)	

We would get the text "DOH!" written in black text on black background, not too clever. Therefore, a pointer is a reference to another place. Because $22 only can hold numbers 0 - 255, the text "Hello World!" would never fit, so instead we point to an area in the user memory, which can hold much more than just a number between 0 - 255. Are you beginning to grasp how computers work?

If you have an enquiring mind, and I hope you do, you'll probably wonder why the addresses $0 - $100 only hold numbers, while it seems that addresses $101 - ... can hold letters. The answer is, they actually can't hold any letters. In addition, it doesn't quite look like I've shown you either. This might get somewhat complicated, hold your hat and don't cry if you don't get it. Just read it, let it go, meditate a bit and you'll reach enlightenment.

The addresses $101 and $200 actually only hold numbers, but the computer has a decode key so that each number can be decoded to a letter. Let's say that

```
! = 0
A = 1
B = 2
C = 3
D = 4
…
```

and so on, then it really looks like this

Memory	Value	Meaning
$200	3	D
$201	14	O
$202	7	H
$203	0	!
...		

So you see, there are just numbers, also, as I said, each address can only hold a number between 0 - 255, meaning that each letter is held in one address. But but but but, why doesn't more stuff get displayed as text? Why does it stop at $203? The memory must continue after that surely. Let's look at this memory setup.

Memory	Value	Meaning
$200	4	D
$201	15	O
$202	8	H
$203	0	!
$204	15	O
$205	4	D
...		

The text on the screen would display as "DOH!OD" and probably much more (the rest of the memory in fact). Well, here we use a control number, let's say that the computer knows that when in reaches the number 255 the text ends there. If memory looks like this:

Memory	Value	Meaning
…		
$19	12	something
$20	1	background colour
$21	0	text colour
$22	200	pointer to text on screen
…		
$200	4	D
$201	15	O
$202	8	H
$203	0	!
$204	255	end of text
$205	4	D

the text “DOH!” would be displayed in white on black background. When the computer reaches $204, it sees the number 255, which means stop displaying text, so the letter (or rather, value) at $205 and following addresses will not be displayed. Like I said, this may be a bit advanced, don’t panic. We will get much more concrete in tutorial 2. I just want you to have a theoretical basis so you know what’s what and so you can refer back to this. Just let this sink into your unconscious, when the time is right and you have correct understanding, it will surface and you will get it.

Now, for the last theory lesson: how do you actually make something happen? As we know, there are .prg files that make stuff happen. With our above knowledge, we know that they affect memory. We can write down simple commands in a text file, and then have that text file translated into the .prg format, so that the computer will understand what we say. A program that can pull this off is known as a compiler, a compiler usually comes with a text editor, suited for programming needs. The text file you use to create a .prg file, is known as the source code. Let’s take another example, this time let’s assume we wrote this source code.

```
Put #1 at $20
Put #0 at $21
Put #200 at $22
Put #4 at $200
Put #15 at $201
Put #8 at $202
Put #0 at $203
Put #255 at $204
Initialise monitor
```

Now, as you can guess, # stands for value, a numerical value in our case, and $ stands for address. Now, if we compile this source code, that is, translate it to a format the computer understands, we will get a .prg file. When we double click on that file, the computer will do what it says above: the different values will be loaded into the different addresses, creating the memory profile given above. The last line “Initialize monitor” is for engaging the monitor. When the monitor is engaged, the Atari knows that it should look at $20, $21 and $22 to gather the data needed. So instead of “Initialize monitor”, perhaps we could’ve written

```
Activate $20
Activate $21
Activate $22
```

Because what we really want to do is to make the information on these addresses happen; we want the computer to process the information given. This is long and clumsy however, and the line “Initialize monitor”, or whatever you might call it, is far simpler.

The computer, internally, understands nothing but 1's and 0's, all text and numbers I have given above is for human understanding (more on binary understanding later). Also, none of the commands or memory addresses have any significance for the Atari, they are examples only.

OK, theory lesson over. Hope I haven't scared you away. In the next tutorial we will get into how to actually make a .prg file. It won't do much, but at least you will get to see your code in action.

2 Of The Workings Of Devpac 3 And The Realisation Of Some Code

"No immovable roots, no stirring of dust, only the tao; the way of nature. This is called tai chi."

- Tai Chi Master

Hello again. It's only been some days since the last tutorial, but I'm bored right now and I don't have any computer game I feel like playing. This weekend went great, as it was GothCon (game convention here in Göteborg, Sweden) which meant games were afoot. I met Aldebaran for the first time IRL and he demonstrated the game Illuminati, which I plan to play later on this evening. But you don't really want to listen to that, do you?

In the last tutorial, we went through some basic theory behind programming, and now we are going to put that theory into practice. We want to do something so awfully cool as to change the background colour to something else. First, we need to gather and learn how to use the tools. The tools in this case are only one; Devpac 3.00 by HiSoft. It can be acquired from Pompey Pirates CD #114, other places, or perhaps you can even buy it :)

Once we start up Devpac, we want to change some settings to make our lives easier. Go into options – control, under the setting format, select ST RAM, depending on your memory size, you may also wish to change the memory buffer. The only two interesting settings here are ST RAM and Atari Executable. If you choose Atari Executable, you will get a .prg file every time you assemble (also known as compile) your source code, so in order to test your source code, you will have to quit Devpac, and run the .prg file, then start Devpac again; enormously clumsy. Instead, we put the .prg file directly into the ST RAM (memory, RAM stands for Random Access Memory), from where it can be executed directly. So for now, set the setting to ST RAM. It's only when you are done coding something, that you will want to change this setting to Atari Executable in order to get a normal .prg file. Go into options – options, and uncheck the "Check absolutes for missing #". This eliminates lots of error reports than usually aren't error reports. In the options – environment you can set the environment variables, if you run from diskette, they should probably be something like PATH=a:\bin, INCDIR=a:\incdir. Go into options – resident and make sure both Assembler and Debugger are checked, this will also save you lots of time if you have the memory (since I run on 4 megs, I don't have to worry). That's about it I think, you can play around with settings on your own if you like, but these are essentials in my opinion.

OK, we have our environment (Devpac) correctly configured, now we want to do something. Remembering the past lesson, we soon realize that to change the background colour, we will have to know what controls the background colour. On Appendix B – Hardware Register Listing, by Dan Hollis, on page 132 – you'll find a listing of the ST's memory, great for reference, in it we find the following lines:

```
$FF8240|word |Video palette register 0                  |R/W
    :  |  :  |  :      :       :     :                   | :
$FF825E|word |Video palette register 15                 |R/W
```

As you may know, the ST is capable of displaying 16 colours at once, with a palette of 512 colours. Having a palette means that you don't have to stay with 16 fixed colours, by changing the palette you can, say have 16 hues of red, and then 16 hues of blue on the next screen. Imagine an artist that can only have 16 colours on each of his paintings, however, he

doesn't have to stick with the same 16 colours each time. OK, it seems that the first colour starts at $ff8240, this is indeed colour 0 and it's the background colour, so by merely changing the value here, we should change the background colour.

I talked about palettes just now, the ST is built on what you call RGB, Red Green Blue, colour. Every colour is made up of 8 levels of Red, Green and Blue, because the computer always counts from 0, the range is between 0 – 7. 8 * 8 * 8 = 512. So, the colour $700 would mean as red as you can get (maximum value of Red, zero value of Green and Blue). $770 would be yellow, $777 would be white and $000 would be black. Equipped with this knowledge we enter this single line as our source code

```
move.w    #$700,$ffff8240    red background color
```

Move is the command used for moving values around other values, in this case, we move the value #$700 into memory position $ffff8240. The # indicates that what comes after is an absolute value, and the $ means that the value is hexadecimal, instead of decimal (more on this later, accept for now). The .w after the move instruction means that the move instruction should move a word, indicating the size of the thing you want moved (more on this later also, accept for now). So, the line above means in clear (?) English; move a value of word size, an absolute value expressed as a hexadecimal value 700 into the memory address $ffff8240. This should be enough to change the background colour to red.

Now we want to assemble the source code, and get our executable. Short command for this in Devpac is alt + a. Now, a window will pop up, displaying some statistics, what we want to search for is especially the line "0 errors found". If there are errors found, the pointer will automatically move to the error so that you can correct it. By pressing alt + number, you can cycle through the available windows. So, if you do get an error, but you don't understand it, press for example alt + 2, which will take you to the second window, where the error report probably is, try to understand something, then hit alt + 1 to go back and edit your source code. ctrl + w will close a window. Okie dokie, the source code has compiled successfully, now we need to run it; hit alt + x (eXecute). Now you will be asked to pass parameters to the program, don't, just hit enter. Oh, the expectation, will the background change to red?

NO! We get two fucking bombs for all our effort. What the fuck? Is there anything wrong with the source code, no, it seems not. Is there anything wrong with the address? Double-checking the address value, no, $ffff8240 means colour 0, which is the background colour. Well, the ST can operate in two modes, user and super mode. In the user mode, we aren't allowed to access certain things in memory, for example the palette, the result if we try to do this is two bombs. So we need to go into super mode. Referring to a list of the so called trap functions of GEMDOS, ideally the ST Internals, we find out how to enter super mode, the code looks like this:

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

(ok, so the code looks somewhat different than in ST Internals, a good lesson for you that you can write differently, but still achieve the same)

Perhaps I shouldn't go too deeply into this, it will come in the next tutorial where I'll take up the different registers and talk about traps and so on. For now, you'll just have to accept it, but for you curious types, here's a short one. The GEMDOS has several special functions, which are accessed by the trap #1 command (calling trap #13 calls BIOS functions). The controlling value is put on the stack, which is address register a7 (you can also type sp, short for stack pointer, instead of a7). The move.l d0, old_stack is for backing up the old user stack, which gets replaced when we go into super mode.

This code obviously goes at the top of our source code, the first thing we want is to go into super mode, then we put red colour in palette register 0, lastly, we want to go back into user

mode and also add another two lines of “accept now, understand later” code, which will make a clean exit of the program. The total code looks like this:

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

                    move.w   #$700,$ffff8240    red background color

                    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

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

old_stack           dc.l     0
```

Running this program will successfully change the background colour to red, and then make a nice and clean exit, restoring the Atari to user mode once again. Problem is, we now have a red background. Not to good you might think. This can be easily remedied however, we have made a program that changes background colour, let’s use it! Change the value #$700 to #$777 and run the program again. The background colour will now be white as snow, like we’re used to.

In the next tutorial, I plan to cover hexadecimal, binary and decimal numbers, program flow and talk some about traps and registers.

3 Of Various Things Mystic And Important, Mainly Concerning The Art Of Understanding Digits And Performing Traps

"With your ability, if you learn to be fluid; to adapt. You'll always be unbeatable."

- Fist of Legend

Hello again! I've gotten some positive feedback on the first two tutorials, so I'm glad to begin writing this third one. As promised, I'll try to explain how computers think when it comes to numbers, the layout of the Atari hardware which will guide us to the workings of traps. I bet a very few understood anything about that.

So, now I'll try to explain what may have been a bit lofty in the previous two parts of the tutorial: how we express numbers. When you see three rocks, you count them, one, two, and three. We have speech in order to communicate our thoughts and emotions to other people, and we have writing in order to communicate speech in written form. We use numbers to communicate "counting". We have chosen the symbol '3' to express the amount you reach when counting one, two and three. However, this value, this "there are three things of something", can of course be written in different ways.

Our number system is based on base 10, meaning that we have ten different symbols to express values, one of them being no values (also known as zero), which leave us with the ability to count to nine. Once the number nine has been reached, we need to start using numbers over again, we don't have a symbol for the value ten, so we have to combine the numbers we have in some way in order to express this. What we do is to say that different positions are "worth" different. For example, in the expression 23, the number 2 is worth ten times as much as three. Do we see a connection here? We use base 10, each number is worth ten times as much as its predecessor. In the expression 123, 1 is worth ten times as much as 2, and one hundred times as much as 3. To calculate the value of the expression 123, we really use this formula:

$$\left(1\times10^{2}\right)+\left(2\times10^{1}\right)+\left(3\times10^{0}\right)$$

the generic formula looks like this

$$\sum value\times base^{position}$$

OK, you say hesitantly, perhaps I understand something of what you're trying to say; now what about computers? Computers use base 2, they are binary and can only count 1's and 0's. There is either current through a circuit, or there isn't. They only use two symbols to express values. As you can imagine, the result is very long expressions, like 10101010101111010001010 10. For presenting the value three, the computer uses the symbols 11. Using our generic formula above, we translate this to

$$\left(1\times2^{1}\right)+\left(1\times2^{0}\right)=3$$

For expressing the number five, we would get 101:

1		0	1		
1×2^2		+	0×2^1	+	1×2^0 = 5
4		+	0	+	1 = 5

Think of it as tumblers, when we have base 10, the tumbler starts at zero, makes nine "ticks", then the tumbler directly to the left makes one tick displaying one, the first tumbler reaches zero again, makes nine ticks, the left tumbler makes one tick, reaching two, the first tumbler starts at zero again. When the second tumbler is at nine, and the first tumbler is also at nine, the next tick will take both tumblers back to zero, and a third tumbler, the one directly to the left of the second tumbler, will go from zero to one, and so forth. If we have base 2, there are only two numbers on the tumblers, 1 and 0.

In order to facilitate things, we also have base 16, or hexadecimal numbers. The reason for this is that conversion between decimal (base 10) and binary (base 2) numbers are somewhat cumbersome, while conversion between hexadecimal, or hex for short (base 16) and binary is fairly easy. We use binary because the computer thinks in binary, we use hex because it's easier to convert binary into hex rather than decimal, and hex is much more manageable (less symbols required to express the same value), and of course we use decimal because we think in decimal.

If you've understood what I said above, you'll wonder how on Earth we use hexadecimal values, since we would need six more symbols for expressing the values ten, eleven, twelve, thirteen, fourteen and fifteen. We simply use letters from our alphabet. A = ten, B = eleven, C = twelve, D = thirteen, E = fourteen and F = fifteen. So to express the value fifteen in hex, we simply use 'F'. To express sixteen, we get 10 ($1 \times 16^1 + 0 \times 16^0$). Seventeen is 11 ($1 \times 16^1 + 1 \times 16^0$. AF would mean one hundred and seventy five ($10 \times 16^1 + 15 \times 16^0$) and finally F5A would be three thousand nine hundred and thirty ($15 \times 16^2 + 5 \times 16^1 + 10 \times 16^0$). If you don't get all this, this topic will probably be covered by every beginner's book and tutorial out there, go find an additional source of information.

In Devpac, values are expressed by putting '#' in front of a number, then further using '$' or '%' before numbers to indicate hexadecimal or binary numbers. Thus, '#' means decimal, '#$' means hexadecimal value and '#%' means binary value. So, as you see, it's customary to express memory addresses in hexadecimal. '$' (without '#') means value at memory address.

Huh, that was that, now onto the architecture of the Atari! Every computer comes with a processor of different (or similar) brands, the Atari has a M68000 processor, so does the Amiga 500. The PC has an 8086 and so on. The processor comes with a different set of instructions and ways of working. This is the real challenge of assembly programming, you must know how the processor behaves that you are trying to program. In high level languages, such as Pascal and C, the programming language usually handles the processor dependent parts, and the code is much more portable between different platforms (computers). Assembly programming however, deals with processor level instruction, which makes it much more powerful and controllable. What the high level languages do, is to take the source code and "translate" it into low level (assembly) language, which of course brings waste, since we can tailor make our programs in assembly, while high level languages have to be more generic (since they weren't tailor made for a specific processor).

This is a good time to take a look at Appendix A – MC68000 Instruction Set, on page 112 – which contains a set of M68000 instructions, you will need to refer to these, also a good guide to the ST's hardware would be most useful, the ST Internals for example (as a somewhat more basic alternative, you'll find such a list in Appendix B – Hardware Register Listing, by Dan Hollis, on page 132). In order to explain the GEMDOS trap routine we used in the last tutorial, I'll have to explain registers. The M68K (K here stands for kilo, meaning 1000, thus M68K and M68000 are synonyms) has 16 registers plus one extra: eight data registers and eight address registers plus a program counter. The program counter is really an address register that points on the next instruction to be performed, simply put, it keeps track of the program execution so that the Atari knows where it's at.

The eight data registers work as eight variables that can store data. The eight address registers are used to store addresses, addresses work like pointers to larger chunks of data, as explained in tutorial one. Here is a usage example of the data register.

```
move.w    #10,d0            put value 10 into data register zero
move.w    #5,d1             put value 5 into data register 1
add       d1,d0             adds d1 to d0 and stores value in d0
* d0 now holds value 15
```

The address register seven (a7) is worth special mention, this is the so called stack. The stack is a lovely pile of data that is used in many ways, for example by the trap instructions. It's of special importance to the PC programmer, since the 8086 has very few registers, the stack is used as a temporary variable, the M68K however is equipped with many registers making the usage of the stack as a temporary variable a bit unnecessary (I hope that no super smart programmer reads this and thinks; what an idiot).

OK, on to the traps. There are three parts of the Atari that can perform traps; the GEMDOS, the BIOS and the XBIOS. The GEMDOS is the hardware independent part of the operating system, this means that the GEMDOS will work on different set of hardware on the Atari. The BIOS is concerned with input and output, like keyboard input and so on, it works between the GEMDOS and the hardware. The XBIOS handles the extended features of the Atari hardware, whatever that means.

You call traps by putting the correct information on the stack, and then calling the correct trap number and then cleaning up the stack. First, we need a special number, called a function number, indicating what function we want, for example, going into super mode is number 32 of the GEMDOS, meaning that we have to put the number 32 on the stack, and then call trap #1 (which is the trap number associated with GEMDOS). Usually, you also need to pass additional information to the Atari, this information is put on the stack before the trap number, and then you call the trap. Thus, put any information associated with the function on the stack, then put the function code on the stack and then call the trap number that corresponds to the handler of the function.

It feels a bit confusing with traps and trap numbers, I'll try to sort it out. The BIOS, XBIOS and GEMDOS each have several special instructions, that aren't in the processor, and do special things. Since they all three have several functions, all three will have the function numbered 1. In the BIOS, function 1 is a function for returning the input device status, in the GEMDOS, function 1 gets a single character from the keyboard, and in XBIOS, function 1 will save memory space. This information must be gathered by referring to a list of all traps available. In the ST Internals, there is a list of all the traps available, and instructions on how to use them.

What you do is to specify what trap function you want by putting information on the stack, then you call either BIOS, XBIOS or GEMDOS, and let them do something with the information. As an example, I give you this alternative way of changing the background colour. The "-(a7)" means put on stack, we'll cover that in the next tutorial.

```
move.w    #$700,-(a7)       colour red
move.w    #0,-(a7)          in colour 0, background colour
move.w    #7,-(a7)          function 7; Setcolor
trap      #14               call XBIOS
addq.l    #6,a7             clean up the stack
```

The first two move instructions put information regarding the call on the stack, first the colour is passed, next the palette number to be changed. Then, the trap instruction number (opcode) must be put on the stack, next we call the XBIOS to process the information. Lastly, the stack needs to be cleaned up so that none of the information we entered will be left, cluttering the system. The code above does the same as

```
move.w    #$700,$ffff8240   red background color
```

The difference is that we take the way around the XBIOS instead of just smacking in the value directly to the memory, which as you can see is much easier to write. Really, since every action taken by the computer only changes the contents in memory, what the traps do is only to change the memory. This we can, in most cases, do ourselves, but in some cases, to call a trap may be easier, clearer and perhaps more also more safe (stable).

Now we are going to expand upon the program in tutorial two. It gets tedious to run the program, and then change the background back by running the program again. Good programs save all the data they change, in order to restore it once the program is complete. Also, it would be nice to be able to store the code for going in and out of super mode, since we will use this in every program.

It's a good idea to have your own libraries of code, which you can cut and paste or include as you will in your code. Create a file called initlib.s (for initialisation library) and in it put the code for the super mode. This file is for storing only, when you need the "go into super mode" instruction, you know where you have it. The file initlib.s should have the following content.

```
iinitialise
* set supervisor
                    clr.l    -(a7)
                    move.w   #32,-(a7)
                    trap     #1
                    addq.l   #6,a7
                    move.l   d0,old_stack
* end set supervisor
                    rts

restore
* set user mode again
                    move.l   old_stack,-(a7)
                    move.w   #32,-(a7)
                    trap     #1
                    addq.l   #6,a7
* end set user
                    rts

                    section  data
old_stack           dc.l     0
```

Well, actually, it can hold any content you want, since this is your personal file, where you store what you find important. I have different libraries to store different things that I need, for example graphlib.s and iolib.s. Your libraries are for storing general purpose programming instructions, like the code for entering super mode.

There are two ways of using your libraries, you can either refer to your library, or you can just include your entire library in the source code you are writing. Including your entire library into your source is somewhat unprofessional, since you then get your libraries splashed all over the place, and any changes or additions that you want to do to the library later on will have to be implemented in all of your programs. The libraries should ideally be kept in one place, to make it tidy and neat. So, in order to include your library, use the include command, followed by the path to your library, like so

```
include \libraries\initlib.s
```

However, there is a drawback to the method described above; every time you assemble your code, you will have to load all your include files into memory from disk, which takes time. Only the source code you are currently working on is in memory, and any includes will have to be loaded every time. So, in order to speed things up, I usually include my libraries (or rather, the subroutines I need) into the source code I'm working at. This can be done with the file – insert file command, which will append a file at the cursor's position. Or you can just copy and paste from your library into your source code as you see fit

```
                    jsr      iinitialise        jump to iinitialise

                    move.w   $ffff8240,d7       save background color
                    move.w   #$700,$ffff8240    red background color

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

                    move.w   d7,$ffff8240       move back old color

                    jsr      restore            jump to restore

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


iinitialise
* 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
```

Soooo, what's different? Our routines for getting into and out of super mode have been neatly packed down the bottom of the code, the command jsr (Jump to SubRoutine) will take us where we want. Then, we put the value of the background colour in d7, saving it. Smack in the red colour. Here comes another fine trap, when executed, it will wait for a key to be pressed before continuing with execution. This trap will give the user a chance to see the lovely red background colour, and then hit a key in order to progress. Again, I looked after what I wanted in the ST Internals; something that would allow the program to pause and wait for a key to be pressed, lo and behold! I found it, and just copied the information. After this, the value from d7 is put into colour 0, effectively restoring the old background colour. Lastly, a jump to the restore subroutine and a clean system exit.

Now we have begun to forcefully control the program flow, the code isn't executed "top down", but with commands such as jsr and rts (Return from SubRoutine) we can jump around. Usually, programs are built up of some initialisation routine, and then a loop more or less, which only consists of calling other sub-routines. A game of asteroids would for example look something like this.

```
                    Go into low resolution
                    Set up player
                    Set up asteroids
                    Main loop
                    Move asteroids
                    Check for collisions
                    Check for player input
                    If no asteroids left
```

```
Set up asteroids
Loop
```

In the next tutorial, we will have to go through addressing modes, that is, the way you can use addresses. This is really important since everything is dependent on what information is where, so a thorough knowledge of how to handle addresses is important. That's all for now. Happy coding!

4 Of The Ways Of Addressing Memory

"Target that explosion and fire."

- Star Trek VI, the Undiscovered Country

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 an 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 Chapter 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 feels 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	$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 :) What 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
$4		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 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 words,
                                        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.

5 Of The Workings Of The Graphics Memory And Minor Skills In Branching

"She doth teach the torches to burn bright"

- Romeo and Juliet

It's 10:13 in the morning, school will start at 1 o'clock so I have some spare time before I'm at it. I've hooked Direct Connect up on some downloads; one Bruce Lee movie and one Yun Fat Chow movie, I've loaded over 70 minutes worth of Atari chip music in Winamp, time to do some serious writing.

As promised in the title, this tutorial will be all about the graphics memory, which really is all you need to manipulate graphics on the Atari. So, if you really try hard, you should be able to do scrollers after reading this tutorial, but don't overextend; I plan to cover scrollers in the next tutorial anyways, because they are so good to practice your skills. First though, I thought I'd take a quick repetition and just go over a few basic things.

Symbol	Meaning
#	decimal value
#%	binary value
#$	hexadecimal value
$	memory address, expressed in hexadecimal
.b	Byte
.w	Word
.l	Longword

One bit is either a 1 or a 0. There are 8 bits to a byte, two bytes to a word and 2 words to a longword, meaning there are four bytes to a longword. BTW, four bits are called a nibble, which is half a byte. The smallest addressable memory block is a byte, meaning that every count of an address is a byte. This means that if a0 points to $100, and you do a move.w #10,-(a0), a0 will point to $98. A0 will decrement by two, because a word is two bytes. If a0 points at $100, and you do a move.l #10,(a0)+, the value in a0 will be $104, since a longword is four bytes. The value at memory address $100 will be 10, also, in the previous example with post decrementation, the value at $98 will be 10, we don't know the value of $100.

I find it easiest to organize my files on the PC, and then transfer what I need to the Atari. You can booo all you want, I don't care! It's really easy to transfer stuff to the Atari, all you need is a diskette formatted in the correct way, you can even use a PC diskette. If you look at an old Atari disk, and a new PC disk, you will see one big difference; there is a hole on the left side of the PC disk, on the same spot that the write protection hole is at on the right side, on the Atari disk, there is no hole. Default size for Atari disks is 720 k, whereas on the PC, it's 1.44 megs (twice 720 k).

Sometimes, you can use PC disks for the Atari without any modifications, just format it to 720 k, the default if you format it in GEM on the Atari. If this doesn't work, just put some tape over the hole, this way, the PC disk will look like an Atari disk. Great huh? Now you can

organize your files on the PC, and have loads of stuff, then, when you need it on the Atari, just put the files you want over on a disk and use it. This disk will work fine on both systems. Only restriction is that you must have it in 720 k format. This can also be done on the PC by formatting in this way "format a: /f:720". If you didn't know this, you'll probably kick my ass for not telling you earlier, hehe, suffer.

Now, on to coding again. As you may have guessed, what you see on the monitor (or TV) is controlled by memory in the Atari. Before explaining that, however, I shall go into the different resolutions. There are low, medium and high, easy as pie. High resolution is that which we find only on monochrome monitors, it's 640*400 pixels, and uses only two colours. Medium resolution is 640*200 pixels and uses four colours. Finally, the most interesting resolution is low, featuring 320*200 pixels with 16 colours. A pixel, btw, is a dot on the screen, if you look closer in a game or so, you'll se that the spaceship/dude/whatever is build up of small dots, those are pixels. The upper left corner is considered 0,0 in a coordination system, and the bottom right corner is the maximum. Thus, in low resolution, the pixel at 0 x and 0 y is in the left uppermost corner, and the pixel at 319 x and 199 y is at the bottommost right position.

How, then, is this represented in memory? For high resolution, it's very simple, each pixel is represented by a bit, either 1 (black) or 0 (white). Thus, if you change the first bit in the graphics memory (sometimes also called screen memory), you will change the bit in the left uppermost corner, the pixel at 0,0. If you change the last bit in the screen memory, you'll change the pixel at 639,399. Since one pixel is represented by one bit, it's easy to calculate how much memory is used, 8 pixels are one byte. 16 pixels one word and a longword will hold data for 32 pixels. 640*400 = 256000, the number of pixels total. If we divide this by 8, we will get how many bytes the screen memory will have to be, this is 32000 bytes.

In medium resolution, we have four colours. Hmm, four colours, how do we represent a value between 0 and 3? Well, we can use two bits, since %11 (binary 11) is 3. So now, we need two bits to represent each pixel. Also, the number of possible lines has dropped by half to 200 instead of 400, meaning that medium and high resolution both use 32000 bytes of memory. You might think that the two bits for each pixel are right next to each other, not so, they are spread over what you call bit planes, but that will come in just a little sec, since it's extremely complicated.

Low resolution has 16 colours. %1111 is 15, so we need 4 bits to represent each pixel in low resolution. The number of pixels per line is reduced by half, and the number of bits per pixel is doubled, meaning that we still have 32000 bytes of screen memory. If you don't believe me, we'll do the math again. 320*200 is 64000 pixels, each pixel needs 4 bits to represent it, meaning 256000 bits, at eight bits to a byte, we again get 32000 bytes.

On to the bitplanes, I will go through how it works in low resolution, since that is the most interesting mode and the exact same technique is used in medium resolution, but with only two bit planes instead of four. OK, here goes. The pixels are stored in words, in groups of 16 (remember, 16 bits in a word). The first 16 pixels are thus stored in 4 words, that come after one another. Thus, the first 4 words of the screen memory are used to store the first 16 pixels. I'm feeling I'm loosing it here, this is damn hard to explain, and it took me weeks before I got it myself.

The bit in the first word is the least significant bit in the colour number. Least significant means the rightmost bit, since this is the one that affect the value the least (it either adds one or zero to the final value), while the most significant bit is the leftmost bit. The bit in the fourth word is the most significant bit in the colour number. The first bits in the first four words control the first pixel. Are you confused yet? An example perhaps.

Graphics memory, expressed in binary	
%1000000000000000	first word
%0000000000000000	second word
%0000000000000000	third word
%0000000000000000	fourth word

Colour number of pixels, expressed in hex for ease of reading
$1000000000000000

The only bit that is set, is the least significant bit of the first word in the series. The term “set” means that a bit has the value 1, and not 0. This means that the first pixel will be colour 1.

Graphics memory, expressed in binary	
%1100000000000000	first word
%1100000000000101	second word
%0100000000000110	third word
%0110000000000000	fourth word

Colour number of pixels
$3F80000000000642

As you can see, just read top down, and you’ll have it.

So, in order to address the 17th pixel, you’d first have to “jump over” the first four words of graphics memory, then manipulate the first bit in the next four words. This makes pixel manipulation a pain in the ass, since not only do you have to change values in four different places, but you also have to work with bit manipulation. All in all, very tedious and time consuming work. Just for comparison, there is a graphics mode on the PC, the MCGA mode, which is extremely user friendly. It also has 320*200 pixels, but 256 colours instead. Does this value ring a bell? It’s a byte! So, each pixel is represented by a byte, making it a wonder of ease of use. In order to change the pixel, you just have to address the correct byte, which is dead simple. It would be done like this, move.b #255,(a0) where a0 points to address memory. This would change the first pixel to colour 255. Or to change the third pixel, move.b #255,3(a0). But to change the first pixel on the Atari, in low resolution, we instead have to do something like this.

```
a0 points to screen memory
move.w    #%1000000000000000,(a0)
move.w    #%0000000000000000,2(a0)
move.w    #%0000000000000000,4(a0)
move.w    #%0000000000000000,6(a0)
```

This sets the first pixel to colour 1. The numbers before “(a0)” are, as you might recall, indexes to memory, so “2(a0)” means where a0 points plus two. Since we constantly want to point to the next word, we must increase the pointer by two bytes each time. We could also have used a “(a0)+” in order to increment the pointer, but then a0 would not have pointed to the beginning of the screen memory anymore. It all depends on what you want to be doing. Also note, that since we move information in, any information previously there, will be lost. If, for example, pixel three and four already had values of some kind, and we executed the

commands above, they would become colour zero, since information regarding them would be overwritten with all zeros as shown above.

Now you hopefully possess the knowledge necessary for understanding my short little program. Let me just stress that really getting the workings of the graphics memory is very difficult. What bit goes where, what bit does what, and so forth, so don't despair when you don't get it right away; you have a long way ahead of you. Oh, I realized, I have some more things to tell you.

A scan line is a row of pixels, there are 200 scan lines in low resolution. That's easy enough. The other thing I have to tell you is about the VBL, or Vertical BLank. The Atari operates in either PAL (Phase Alternating Line) or NTSC (National Television Standards Committee): NTSC is the American standard and PAL the European. Since I'm from Europe and it also seems that most Atari related stuff is from Europe, NTSC will be given little support, take that Yankees. The PAL or NTSC has to do with how many times per second the screen is updated, in NTSC, it's 60 times per second, and in PAL it's 50. Thus, the so called refresh rate, is either 50 or 60 Hz. On game menus, you can often change between these modes. When I was little, and only played games, I never got what the 50/60 selection on the game menu was about, now I do. Since we use PAL, the refresh rate on our stuff will be 50 Hz, meaning that the monitor is updated 50 times per second.

The screen is painted by an electron beam, that starts in the upper left corner, and then works its way down, doing a scan line, and then moving on to the next. This happens 50 (or 60) times every second. It's good practice to synch your graphics with this beam, this will be further expanded in the next tutorial. There is a trap, that will put the system in pause until the next VBL, that is, the next time the electron beam is about to paint the screen. This is an excellent timer, and will allow you to know exactly how much time everything takes. Just think about it, if you put the wait for VBL trap in the beginning of your main loop, you'll know that the loop will perform 50 times per second. This is ideal for making games or demos not run to fast. The trap function number is 37, it's called by XBIOS and looks like this:

```
move.w   #37,-(a7)          wait vbl
trap     #14                call XBIOS
addq.l   #2,a7              clean up stack
```

This is a good thing to include in your graphics library if you have one, if you don't, you might think about making one.

I realize when looking over the source code again, that there are some more things to explain. Hehe, well, at least I explain them sometime, and I don't just dump the source code on you and let you browse through those instruction sets and figure things out for yourself. Of course, it's a good thing to know where the graphics memory is, unlike some other computers that has a fixed location for the screen memory, the Atari can use any part of the memory. This simple trap will put the address of the graphics memory in d0, which you then can move into the address register of your choice.

```
move.w   #2,-(a7)           get physbase
trap     #14                call XBIOS
addq.l   #2,a7              clean up stack
move.l   d0,a0              a0 points to screen
```

Actually, it might be somewhat of a bad habit to use registers d0-d2 and a0-a2 unless you have to, since those registers can be destroyed by, for example, calling traps, and other similar things handled by parts you don't have full control over. Physbase here stands for physical base, and means the physical base of the graphics memory. Note also, that when moving addresses, like the last command above, you should always use longword size. This is so because the Atari uses 24-bit addresses, each address is 24-bits long, and if you only move a word, or heavens, a byte, information will be lost.

What more, oh yes, the dbf and clr commands. We'll start with the easy one, clr. CLeaR clears all bits in the effective address operand. In clear English, this means "make something zero". For example:

```
move.l   #$100,a0
move.l   #10,d0
move.l   d0,(a0)
clr.l    d0
clr.l    (a0)
```

Now both d0 and $100 will contain zero.

The dbf command is a bit special. Instead of dbf, you can also use dbra. It is used for making a loop a certain amount of times, it's the equivalent to a for-loop in high level languages. When using the command, you give a controlling data register, and the address to loop. Each time, the data register will get decremented by one, and then it will be tested to see if it's -1, if it's not, the execution will jump to the given address.

```
move.l   #$100,a0
move.w   #4,d0              execute loop 5 times
loop
move.l   d0,(a0)+
dbf      d0,loop
```

So, can you figure out what the memory configuration will be for this?

Memory	Value
$100	4
$104	3
$108	2
$10C	1
$110	0
(some hex counting training as well, aren't I nice?)	

Since the value gets decremented right before it's tested for -1, the loop is never looped through with the value -1. So, if you want a loop to loop five times, put four in the controlling data register. Remember that on the last loop, the data register will contain zero. That should be it, finally, we can get to my training program. You should be able to figure it out yourself, but I hate it when people say that and I still have many questions, so I'll walk you through it.

The program fills the first 60 scan lines with colour 1, the next 60 with colour 2 and the next 60 with colour 3. Then it sets the colour values for these three colours to the maximum level of the three "main colours", RGB, or red, green and blue. When this set up is done, it decrements the value for each colour by one every half second, when the values reach zero (black) the program terminates itself. The countdown itself is achieved by first waiting 25 VBLs, and then running through 7 such waits.

```
                    jsr      initialise

                    move.w   #2,-(a7)           get physbase
                    trap     #14
                    addq.l   #2,a7

                    move.l   d0,a0              a0 points to screen

* clears the screen to colour 0, background
                    move.l   #7999,d1           size of screen memory
clrscr
                    clr.l    (a0)+              all 0 means colour 0 :)
                    dbf      d1,clrscr
```

```
                    move.l   d0,a0              a0 points to screen

* fills screen with colours, ok 180 scanlines :)
                    move.l   #1199,d0           60 scanlines
fill1
                    move.w   #%1111111111111111,(a0)+
                    move.w   #%0000000000000000,(a0)+
                    move.w   #%0000000000000000,(a0)+
                    move.w   #%0000000000000000,(a0)+
                    dbf      d0,fill1           filled with colour 1

                    move.l   #1199,d0           60 scanlines
fill2
                    move.w   #%0000000000000000,(a0)+
                    move.w   #%1111111111111111,(a0)+
                    move.w   #%0000000000000000,(a0)+
                    move.w   #%0000000000000000,(a0)+
                    dbf      d0,fill2           filled with colour 2

                    move.l   #1199,d0           60 scanlines
fill3
                    move.w   #%1111111111111111,(a0)+
                    move.w   #%1111111111111111,(a0)+
                    move.w   #%0000000000000000,(a0)+
                    move.w   #%0000000000000000,(a0)+
                    dbf      d0,fill3           filled with colour 3

                    move.w   #$000,$ff8240      black background
                    move.w   #$700,$ff8242      red colour 1
                    move.w   #$070,$ff8244      green colour 2
                    move.w   #$007,$ff8246      blue colour 3

                    move.l   #24,d5             25 VBLs per loop
                    move.w   #6,d6              make 7 loops
main
                    move.w   #37,-(a7)          wait VBL
                    trap     #14
                    addq.l   #2,a7

                    dbf      d5,main            loop VBLs

                    add.w    #-$100,$ff8242     subtract one from red
                    add.w    #-$010,$ff8244     subtract one from green
                    add.w    #-$001,$ff8246     subtract one from blue

                    move.l   #24,d5             reset VBL counter

                    dbf      d6,main            end of main loop


                    jsr      restore

                    clr      -(a7)
                    trap     #1

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

* save the old palette; old_palette
                    move.l   #old_palette,a0    put backup address in a0
                    movem.l  $ffff8240,d0-d7    all palettes in d0-d7
                    movem.l  d0-d7,(a0)         move data into old_palette
* end palette save
```

```
* saves the old screen adress
                    move.w    #2,-(a7)          get physbase
                    trap      #14
                    addq.l    #2,a7
                    move.l    d0,old_screen     save old screen address
* end screen save

* save the old resolution into old_resolution
* and change resolution to low (0)
                    move.w    #4,-(a7)          get resolution
                    trap      #14
                    addq.l    #2,a7
                    move.w    d0,old_resolution  save resolution

                    move.w    #0,-(a7)          low resolution
                    move.l    #-1,-(a7)         keep physbase
                    move.l    #-1,-(a7)         keep logbase
                    move.w    #5,-(a7)          change screen
                    trap      #14
                    add.l     #12,a7
* end resolution save

                    rts


restore
* restores the old resolution and screen adress
                    move.w    old_resolution,d0  res in d0
                    move.w    d0,-(a7)          push resolution
                    move.l    old_screen,d0     screen in d0
                    move.l    d0,-(a7)          push physbase
                    move.l    d0,-(a7)          push logbase
                    move.w    #5,-(a7)          change screen
                    trap      #14
                    add.l     #12,a7
* end resolution and screen adress restore

* restores the old palette
                    move.l    #old_palette,a0   palette pointer in a0
                    movem.l   (a0),d0-d7        move palette data
                    movem.l   d0-d7,$ffff8240   smack palette in
* end palette 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_palette         ds.l      8

old_resolution      dc.w      0

old_stack           dc.l      0


                    section   dss

old_screen          dc.l      0
```

Oh, naughty me, I added a bunch of stuff to my initlib without telling you about it. Well, right now, you'll just have to accept it, any problems with that private!? The thing it does is to save all information regarding resolution, screen setup and so on, then change to low resolution. When the restore subroutine is called, it restores everything as it was. While time goes by, I probably won't dump all my source code into my tutorials, for example, an include initlib.s will probably be the way in the future. I'm also thinking about sticking to just give out the separate .s file with the source code, and only comment it here in the main tutorial so you won't have the same code in two places. How does that sound? You curious types can go through the initlib code, and try to figure it out, I have commented it quite well just so you can do that.

There might be some problems with the math here, in the clear routine, 8000 is given as the screen size. Yes, 8000 longwords, 8000*4 = 32000 bytes. 1199, or rather 1200 should equal 60 scan lines? Yes, every pass through the fill-loop moves 4 words. 4 words contain information for 16 pixels, meaning that for every loop, 16 pixels will be set. 320*60=19200 pixels total (320 pixels per scan line), and since we set 16 pixels per loop, we divide this value by 16 to get the total number of loops, which, incidentally, is 1200. That should clear any trouble with the numbers.

I hope there's no trouble with the main loop part, the first little loop is all about making 25 VBLs, in other words, waiting for 0.5 seconds. Then, the colour values are changed, making the colours 1,2 and 3 go towards black. Lastly, another loop controller that makes sure the main loop is looped through seven times.

Now that you are equipped with basic knowledge of the graphics memory, I think we'll be able to handle a scroller in the next part. It depends, I'll have to write one and see if it's not too complex. If it is too complex, you'll probably be looking at a theory tutorial again.

6 Of Seeing Behind The Curtain Of An Execution And Getting Intimate With Files

"Great! I love fighting."

- Fong Sai-Yuk

Hiya'all, it's been a little while since the last tutorial. Mainly because I wanted to code a little bit for myself and not only write stuff. This tutorial will NOT be about scrolling, unfortunately, but it will cover the theoretical base which you'll need to be able to do the scrolling as will be covered in the next tutorial. However, after this tutorial, you may figure it out by yourself. Of course, the thing you have to do to scroll, is to just move the correct screen memory bytes to the correct place. This will be covered in depth in the next tutorial, promise.

We're now beginning to get past the most fundamental theory, and so our code is getting to be more and more advanced. This in turn means that often, a program will assemble without errors, but it still won't work the way we want it to. Something somewhere is not as we thought it would be, a variable might not be assigned the correct value, a mathematical equation might not produce what we thought and so on; endless possibilities. This is where the debugger comes in. Debugger? says you. To illustrate, let me tell you this fairy tale.

In the olden days, there was a big computer. So big it was that two men could not put their arms around it. The computer stood in the big country that lies west of here, and all day long it crunched numbers. It was very happy. Then, one day, it could not crunch numbers any more, something was wrong and the computer fell sick. All the people in white robes, that saw to the computers every need, were greatly distressed. No one knew what was wrong. So, in a last desperate effort, they opened up the poor computer to have a look inside. They found that a little bug had flown in, and that was the root to the sickness. So, the people in the white robes removed the bug, and the computer was again healthy. It was all smiles and could once again crunch numbers all day long. Thus endeth the tale. (since this is a fairy tale, I make no claims that the exact facts are true, but like all legends, it contain a grain of truth)

Debugging, is the art of removing errors from source code. This is actually very hard, and one can probably be as skilled in debugging as writing code in the first place. Debugging usually takes at least half the time of developing a program, so good planning and lots of time in the debugger is a good thing indeed. Nowadays, bugs are errors in the source code, rather than actually physical bugs. Debugging is getting rid of bugs, creating error free code, and a debugger is a tool that helps you with this process. Devpac comes with a debugger, called MonST, I guess it stands for MONitor ST.

After you've assembled to memory, instead of pressing alt+x and run the program, you can press alt+m and run the MonST, henceforth referred to as the debugger. Lots of information will pop at you, and after you've come over the shock, you'll start to make quick sense of it. There are three "windows", areas rather, registers, disassembly pc and the memory. The disassembly pc area is your actual source code, the other two should speak for themselves. When you are in debugger mode, instructions will be executed one at a time, this allows you to see how each instructions change the content of memory and registers. I'll go through each area and what you do with it.

Registers, here you have the content of all data registers, all address registers, the status register and the program counter. All values are given in hex, which makes every two digits

one byte, and each digit one nibble. As you can see, there are eight digits for each data register, which makes sense since you can store a longword in a data register. When data registers are beginning to get filled with values, there will pop up some symbols, sometimes strange, to the right of the register. These symbols are the ASCII equivalents for each byte in the data register. We haven't talked about ASCII I think, but it's the way to represent characters with numbers I mentioned back in tutorial one. For example, the number $41 is the letter 'A'.

The address registers are to the right of the data registers, and work pretty much the same. To the right of the address registers, are the memory content that the address register points to. Since there are four digits to every group, each group is a word. Thus, to the right of each address register, is the memory content of the first five words that the address register points to. To the right of the memory content, you'll also see ASCII representations of the content, just as with the data registers.

Below the data registers, are the status register and the program counter. The status register haven't been mentioned much either, but it takes note of several statuses of the ST, for now, it will probably be 0300 and you'll se a 'U' to the right of it. The U means User mode, and that's what we're in now until we change it to Super visor. The status register will also keep track if a mathematical operation results in a overflow and so on. An overflow is when the number generated is bigger than can be stored, for example, adding two data registers with very big values will generate a value to big to store in one data register, so data loss will occur. Below the status register is the program counter, and to the right of the program counter you'll see the instruction that it points to.

The disassembly area is the code you're currently debugging. It will look just like your source code. You can scroll up and down the code, and a little arrow will indicate your current position. To execute a line of code, press alt+z, to skip a line of code, press alt+s. Usually, you'll want to skip jumping into the initialise subroutine, because this takes some time and might also put the ST in low resolution, making it hard to see anything. You'll usually want to go to the mathematical equations directly, to see what happens. There's also a very nice way to jump straight to a position of your choosing. You can put "flags" in your source code, by entering the command "illegal", then, when in debugger mode, hit ctrl+r. This will execute all commands from your current position to the next illegal position, you'll have to skip past the illegal instruction to continue, using alt+s. A great way for executing an entire loop without stepping through it all.

The memory area is most interesting, this is where the entire content of the memory is listed. By pressing m, you can type in the name of any memory tag (variable) that you are using, and see what the memory that it points to contains. If you're smart, you'll immediately type in ff8240, which will take you directly to the palette. Unfortunately, that will get you little, since this is protected memory, you'll only see *'s.

You can change between these areas by pressing tab, and you can only issue commands in the active area. When you are done debugging, you don't have to wait for the whole program to execute and terminate, just hit ctrl+c, twice. Now this is useful, right? The best way to get to the workings of the debugger is, like always in programming, to get to it; debug some simple piece of code and see what happens to the registers and memory. Oh, yes, in the memory area, you can also type in aN (where n is 0-7) to get directly to the memory area pointed to by an address register.

Now, onto file formats! A file is simply a collection of data. There really is no such thing as a .pi1 (Degas Elite) file, or an .mp3 file. A file contains data, so, this data is interpreted. Different things will happen depending on how you interpret the data. Let's say, for example, that we have a file containing only a byte, and it holds this data

```
%01000001
```

Easy, says some paint program, these are the first eight pixels in monochrome mode. Pixel number 2 and 8 is supposed to be black, the rest are white. No, says the text editor, %01000001 is $41, which corresponds to ASCII character 'A'. This is the letter A. Nonsense, says the home taxation program, %01000001 is a control code in my program that says this

file represents a terminated account... and so on. Programs interpret files, and do something with the information. Since programs are also files themselves, interpreted by the operation system, which is itself also files more or less, the whole shit is built on subjective opinions on what to do with the data presented.

Given the information above, one might think that it's a good way to know how different programs interpret data, this is the knowledge of file formats. In order to understand this, we will examine a very simple file format, the Degas Elite .pi1 file format. It's almost too simple really, but it's useful and we're going to use it in our next tutorial. Usually, files have so called file headers, which give some information about the file. For example, a Windows BMP file, starts with the ASCII codes for 'B' and 'M', which makes sense and gives a signal of what kind of file it is. It's of a little nerdy interest to know that each .exe file on the PC, starts with the ASCII codes for the letters 'MZ', which was some hot shot in Microsoft back when they defined the file format (and perhaps still). A good example of a file header could perhaps be the resolution of an image, or the font type in a word processor file.

In order to examine files correctly, we need a so called hex editor. A normal text editor will not do, since the text editor would interpret data as ASCII code, we want a program that just presents the data in the file, and does not interpret it in any way. With this hex editor, you can "hack" files yourself. Say, for example, that you want a program that converts one graphic file format to another; you'd need knowledge of both file formats. Sit down with a paint program, and a hex editor. Do some small changes in the paint program, and watch what's changing in the file with the hex editor. This is tedious work, at best, and you're probably better off trying to locate the information somewhere. So, in order for you to begin and try out your efforts, I will tell you how the .pi1 files look like.

First, there are two bytes giving the resolution, in low resolution, it's just 0, in medium, 1, and in high resolution, 2. Then comes 32 bytes containing the palette data for the picture. After that comes the pixel information, looking exactly the way it does in the screen memory. And that is that. Very simple file format indeed. So, how big is a .pi1 file then, only knowing the above? 32034 bytes. 32000 bytes for the pixel information, 32 bytes for the palette, and two extra bytes in the beginning of the file. Here's a little program that will display a .pi1 file (a little note: in Degas Elite, there are 32 bytes in the end containing information on animation and stuff, uninteresting in our case).

```
                    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.w  #2,-(a7)          get physbase
                    trap    #14
                    addq.l  #2,a7

                    move.l  d0,a0             a0 points to screen memory
                    move.l  #picture+34,a1    a1 points to picture

                    move.l  #7999,d0          8000 longwords to a screen
loop
                    move.l  (a1)+,(a0)+       move one longword to screen
                    dbf     d0,loop

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

                    jsr     restore

                    clr.l   -(a7)
                    trap    #1

                    include initlib.s

                    section data
picture             incbin  jet_li.pi1
```

There are three new instructions here, movem, incbin and include. Include is the easy one, just consider it as though you had pasted the entire contents of the initlib.s file on the include line. As you will see, when you assemble the code, this takes a while since the Atari needs to read the file each time. Therefore, I strongly suggest you actually do paste the file in, instead of just including it. Your choice.

Incbin, as you may have guessed, is the way to include files, they fall under the section data. This puts the entire contents of the file in memory. In this particular case, I put the entire contents of the .pi1 file called jet_li.pi1 at the memory position I choose to call picture. You can achieve the same result by hand copying the content of jet_li.pi1. Something like

```
picture             dc.b      0,0,0,0,$07,$11 …(this is the beginning of the file)
```

Movem MOVEs Multiple data from memory to registers or the other way around. It can only move words and longwords. As you can see, I move the memory from picture+2 into the data registers. This is great since all eight data registers can hold all in all 32 bytes of data, since each colour is 2 bytes of data, this means that the entire palette of 16*2 bytes of data fits precisely into the eight data registers. The reason for picture+2 is that we want to skip the first two bytes, since they only contain resolution information. After filling the data registers with the palette, we just smack it in at the correct starting address.

Then, it's a question of putting the screen memory pointer in a0, and the start of the pixel part of the picture in a1. The picture+34 is because this is where the pixel part starts, 2 resolution bytes plus 32 palette bytes is 34 bytes that should be skipped in order to reach the pixel part. As shown in the previous tutorial, the screen size is 8000 longwords. I just loop through that amount, copying the content from the picture into the screen memory. Easy? This is a small loader for .pi1 files. If you assemble this piece of code as a .prg file (or just take my pre-assembled file), you'll notice that the program size will be 32494. Most of this is the .pi1 file itself, our added code is only 32494 – 32034 = 460 bytes. We now have a self-loading .pi1 image, nice.

If you think it would be amusing, you can add this little loader to all your .pi1 files, in this way, you'll never have to go through Degas to watch them, they load themselves. Of course, you'll get a .prg file instead of a .pi1 file, meaning that you can't edit it with Degas. But then you could write your own program for extracting the image information and turn it into a .pi1 file again. Fun, right? Note; you don't have to keep the original .pi1 file for this "loader" to work, since the .prg file contains the data it needs for the image.

While we're on the topic, I will mention, briefly, compression. You must know what file compression is, it's making a file smaller, but usually useless, until you decompress, or unpack, it again. How does this work? The file can't just shrink, can it? Well, more or less, it actually can. Consider this information.

```
%00000000
%11111111
```

The first byte is all 0's, and the second one all 1's. Suppose we replace the information given with

```
08
18
```

and tell the program that after each 1's or 0's, there will be a number that tells how many 1's or 0's there will be. If we have a file with big areas of similar data, for example 50 bytes of 0's and then 70 bytes of 1's, this so called compression algorithm would compress this information into four bytes. It would look like this

```
050
170
```

or, just to give you some bit mathematics, we say that the high bit of each byte controls whether it should be 1's or 0's, and the next seven bits tell how many of each kind should follow, it would look like this.

```
%00110010   50 0's
%11000110   70 1's
```

That was that on compression. The above is a very simple compression algorithm and if you use it, you may end up with files bigger than they were from the beginning. I know file compression was a bit sketchy, but if you get the part of how files work, the compression part shouldn't be that hard. Also, file compression might be covered more extensively later. So far, I know very little myself since I haven't used it for anything. I have no idea how good file compression algorithms look or anything, so don't ask. This is just the theoretical base. Study carefully, since I'm going to use a .pi1 file for the font in the upcoming scroller.

7 On Scrollers

"My grandfather taught, me the energy of life goes in a circle, in a perfect flow; balanced. He said, until I find my centre, my circle will never be whole"

- The One

Huh, so finally, as promised; the tutorial on scrollers. BTW, all my "huh" sounds aren't like American huh, as in a question or as in a "oh yeah?", but rather phew, like a sigh. Have you been waiting for this one? I hope you have, because it was a damn pain in the ass to write the scroller program, even though it's simple. It began with me reaching too high, also, forgetting about the bitplane layout of the graphics memory. When I put a little lower ambition level, for the sake of keeping it simple, things went smoother. Now, Luca Turilli playing in Winamp, the mood is set, time to write. The people who already know how to do scrollers will probably laugh their ass off at this clumsy scroller, which really is bad in every way except learning the basic stuff, I'll probably do a more advanced one later on; I've heard that building on knowledge is good.

A few happy news first. I've gotten mail from three different people, excluding Maarten Martens. Thanks guys, you know who you are! One mail from FrEd highlighted a few misses I made, concerning the compatibility with Devpac 2. My initlib had a little bug. It works fine in Devpac 3, but not in 2. Two lines had d0-7 in them, it really should read d0-d7, but it's fixed now. I know some other things may also be difficult with other assemblers than Devpac 3, so if the code doesn't work for you, just use Devpac 3. I have tested every piece of code with that on an original Atari ST(e), so there should be no problem. Thanks go out to FrEd for pointing this out, and also to mOdmate of Checkpoint for telling me a little about the workings of $fffc02.

Yep, a scroller. I'm a bit unsure of where to start, but I guess I'll just work from the top down. What does a scroller do? Letters go from the right of the screen, to the left of the screen (usually). New letters are brought in from the right, "outside" of the screen. How can this be achieved? The screen memory needs to be moved "to the left", and then we need information to bring in the new characters from the right. OK, this seems to build on an idea to have letters stored as graphics. Hum, yes, we have a font collection in a degas file. In that way, we'll have letters in graphics format, we can take the information from the font file and put it on the screen. Then, we move the screen memory to the left. Easy? No, damn hard for a first timer at least.

Included in this tutorial should be a file called FONT.PI1, this is the font file, I stole it from James Ingram's demo tutorials, so I wouldn't have to make my own. Immediately load this up and look at it, either using Degas, the program from Chapter 6 or any other method. Lucky lucky, lots of characters to choose from. Each character is 32 * 32 pixels big, resulting in 10 characters per line. This is all well and fine, the next step is to actually know how to point to the beginning of, for example, letter 'C'. If we know where this letter begins, we can put it on our screen simply by moving the data into the screen memory. Just as we did when displaying a whole picture.

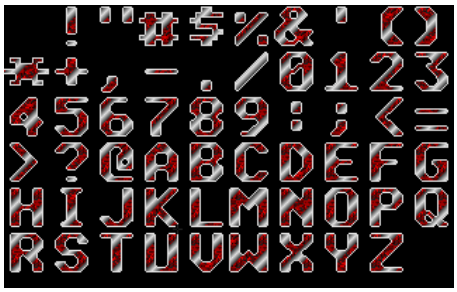


Figure 1 - FONT.PI1

The font picture is aligned with the ASCII table, meaning that it looks like the ASCII table does. In Appendix C – ASCII Table, by Stephen McNabb, on page 148 – you'll find an ASCII table, in which you can look up the number for each character. As you can see, space (the first character in the font), begins at $20, then comes '!' at $21 and so on. This means, that if we take the ASCII value for a character, and subtract by $20, we'll have the corresponding number in the font. Hum, a test perhaps. 'C' is at $43 in ASCII, subtract $20 makes $23, which is 35 (decimal). There are 10 characters per line, so we skip to the fourth line, begin counting; 0 (>), 1(?), 2(@), 3(A), 4(B), 5(C), yay, right on! (remember to start counting from 0).

Now we need to know what address this is at. The way to do this is to put the beginning of the font picture address in an address register, and increment by a number. Think of the font as a coordinate system, then 'C' would be at 3,5. We need to increment the pointer by a certain value for each coordinate, this shouldn't be to hard.

Each line is 160 bytes, and each character is 32 lines. This means that for every Y coordinate, we need to increment the pointer by 32 * 160 bytes, right? Think about it, if we want '*' which is on the second line (1,0), we need to point to the font address + 32 lines down. Each character is 32 pixels wide, 16 pixels are 4 words, taking up 8 bytes, we need twice this. So for each X coordinate, we need to increment the pointer by 16 bytes.

Does this seem right? Let's try. We want letter 'C', at 3,5. Thus we should increment by 3*32*160 + 5*16 = 15440 bytes. 'C' is about the middle of the screen and 15440 is about half of 32000, so it seems safe to assume that the formula above is working. Question is, how do we get the X and Y coordinates? We had a value for C, right, that was 35. The first digit seems to be the Y coordinate, and the second the X coordinate. If we divide 35 by 10 we get 3.5. 3 is the quotient and 5 the remainder. The instruction divu (DIVide Unsigned) puts the quotient in the lower word of a data register, and the remainder in the higher word. Swap is an instruction that swaps the low and high word in a data register. Great! We now have what we need. The code looks like this:

```
move.l   #character,a0      points to character
move.l   #font+34,a1        points to pixel start

move.b   (a0),d0            put letter ascii value in d0

add.b    #-$20,d0           align asciin with font number
divu     #10,d0             10 letters per row

move.w   d0,d1              d1 contains y value
swap     d0
move.w   d0,d2              d2 contains x value

mulu     #16,d2             16 bytes for each letter
mulu     #32,d1             32 lines per row
mulu     #160,d1            160 bytes per row

move.l   #font+34,a0        put font screen start in a0

add.l    d2,d1              add x and y value together
add.l    d1,a0              a0 points to correct letter
```

```
                    section   data
font                incbin    font.pi1
character           dc.b      "C"
```

Since each character is an ASCII value, we only use a byte to represent it. If we put things inside "", that means we want the ASCII value. So the message dc.b "C", means that message is a byte containing the ASCII value for C. We could just as well have written message dc.b $43, but this is more difficult to understand. Hopefully, the code will speak for itself with the comments and the theory given above. This is not a complete program, but just a code snippet to show the font part. More will follow.

We know how to point to the font, now we need to know how to shift the screen memory, in order to achieve the scrolling effect. One would think that all it took was a big loop moving bytes. Like so (a0 and a1 contain the address of the screen memory)

```
                    add.l     #1,a1 put a1 8 pixels ahead of a0
                    move.l    #159,d0 scroll a line
loop
                    move.b (a1)+,(a0)+
```

For each loop we take the byte one byte ahead, and move it one byte to the left. This should move 8 pixels each loop, right? Wrong! Totally wrong! The screen is made of 16 pixel clusters, each cluster being 8 bytes long. So when you just barge in and move single bytes like that, you'll misalign the whole shit. Not only will the colours be misaligned, the pixels will be as well. Consider this memory configuration.

First byte	Second byte	
%11000000	%00000000	first word
%11000000	%00000101	second word
%01000000	%00000110	third word
%01100000	%00000000	fourth word
%00000000	…	fifth word
$3F800000	$00000642	pixels

If we use the move loop from above, the first byte will drop out, the second byte will be moved into the first byte, the first byte of the second word will go into the second byte of the first word and so on, in the end, we get this.

First byte	Second byte	
%00000000	%11000000	first word
%00000101	%01000000	second word
%00000110	%01100000	third word
%00000000	%00000000	fourth word
$00000542	$17400000	pixels

Not really, the pixels we had before. So, in order to overcome this in an easy way we move 16 pixels each time. This will produce a very fast scroller, but an easy one to code for. If we move 16 pixels, we won't have to worry about getting misaligned bitplanes, since the 16 pixel clusters will never be broken up, like they were above. a0 and a1 contains the screen address, while a2 points to the character in the font.

```
                    add.l     #8,a1             put a1 16 pixels ahead of a0
                    move.l    #31,d1            32 lines to scroll
                    move.l    #18,d0            19 16 pixel clusters + font part
                    scroll
```

```
move.w    (a1)+,(a0)+
move.w    (a1)+,(a0)+
move.w    (a1)+,(a0)+
move.w    (a1)+,(a0)+       16 pixels moved
dbf       d0,scroll         keep moving 16 pixel clusters
move.l    #18,d0            reset loop counter
move.w    (a2),(a0)+
move.w    2(a2),(a0)+
move.w    4(a2),(a0)+       16 pixels of the font
move.w    6(a2),(a0)+       character moved in
add.l     #8,a1             increment screen pointer, align with
                            a0
add.l     #160,a2           next line of font
dbf       d1,scroll         do another line
```

This is all just a bunch of move words, and some adds to keep everything aligned. The first move section will move 4 words from a1, which points one 16 bit cluster ahead of a0, to a0. This is repeated 19 times. After this loop, a0 points to the beginning of the last 16 pixel cluster, and a0 points to the beginning of the second line. For the last 16 pixel cluster, we want information from the font, not from the screen. So here we move information from a2 into a0. Instead of post incrementing a2, I use indexes. After the font data is moved onto the screen, I add 8 to a1, so that it will again be 16 pixels ahead of a0. Since a0 was incremented during the font move part, and a1 was not. 160 is added to a2, so that the font pointer will now point to the next line in the font. Repeat for 32 lines.

Now the two most important techniques have been covered, how to know where the character is in the font, and how to scroll. Now we mix and match. In order to synchronize the entire scroller to the VBL, I put a wait VBL trap in the beginning of the main loop. Then I do my stuff, and in the end of the main loop, I check if the space bar is pressed, if it is, just drop out of the loop. If space bar is not pressed, then the main loop will begin again, with a VBL wait, making sure that the main loop is looped through at 50 times a second. You'll probably be wondering exactly how I determine whether the space bar is pressed.

This little piece will do the trick: cmp.b #$39,$fffc02. Uh, says you, looking at the ASCII table (hopefully) and wondering how $39 can be space, when it should be $20. The $fffc02 part can be easily guessed, this is probably where the last key press end up, but why $39? ASCII deals with characters, and special characters like line feed and so. There's also something called scan codes. Every key on the keyboard has its value, its scan code, so you'll be able to determine what key was pressed. Look at the picture below:

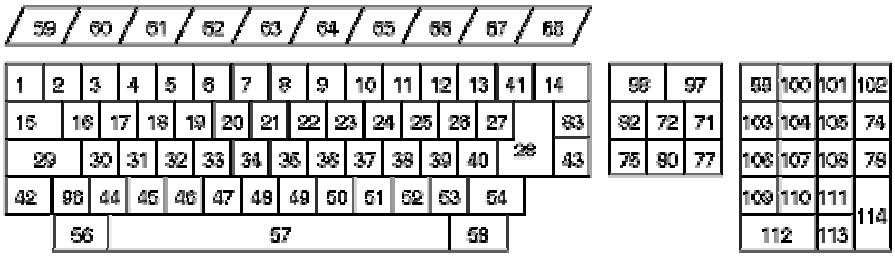


Figure 2 Keyboard Scan Codes

While we're still on the topic, I might as well give you the full detail. You can also check when a key is released, not just pressed. When the key is released, the high bit of $fffc02 is set, meaning you get a whole different value. Consider this.

%00111001	$39 space pressed
%10111001	$b9 space released

So, if you cmp.b #$b9, then you check if space is released. This can be used in many fun ways, like changing the background to red when space is pressed, then checking to see when space is released and then restore background. Or accelerate a car in a car game until the button is released, at which time you begin deceleration. I don't know how often this is updated or how fast you can really press keys. Say for example that you check $fffc02 every VBL to see what key is pressed and released, suppose this dude is like Flash, and manage to press a button, then release it and press another within 1/50 of a second, then you'd loose the check for the release of the key, but I doubt you'll have to worry about this. Back to reality, here's the scroller.

```
                    jsr      initialise

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

                    move.w   #2,-(a7)          get physbase
                    trap     #14
                    addq.l   #2,a7
                    move.l   d0,screen         store screen memory

main
                    move.w   #37,-(sp)         wait vbl
                    trap     #14
                    addq.l   #2,sp

                    cmp      #0,font_counter   check if new character in message
                    bne      has_character     if not, skip get new character

                    move.w   #2,font_counter   reset font_counter
* we need to point to a new character in the font

                    move.l   message_pointer,a0 pointer into the message
                    clr.l    d0                clear, just to be sure
                    move.b   (a0),d0           put letter ascii value in d0

                    cmp      #0,d0             end of message?
                    bne      not_end           if not, branch

                    move.l   #message,message_pointer      reset message_pointer
                    move.l   message_pointer,a0
                    clr.l    d0                clear, just to be sure
                    move.b   (a0),d0           put letter ascii value in d0

not_end
* now we have a character in d0 for sure
                    add.l    #1,message_pointer point to next character

                    add.b    #-$20,d0          align ascii with font number
                    divu     #10,d0            10 letters per row

                    move.w   d0,d1             d1 contains y value
                    swap     d0
                    move.w   d0,d2             d2 contains x value

                    mulu     #16,d2            16 bytes for each letter
                    mulu     #32,d1            32 lines per row
                    mulu     #160,d1           160 bytes per row

                    move.l   #font+34,a0       put font screen start in a0

                    add.l    d2,d1             add x and y value together
                    add.l    d1,a0             a0 points to correct letter

                    move.l   a0,font_address   store calculated pointer

has_character
                    add.w    #-1,font_counter
```

```
                    move.l   screen,a0
                    move.l   screen,a1
                    move.l   font_address,a2
                    add.l    #8,a1              put a1 16 pixels ahead of a0

                    move.l   #31,d1             32 lines to scroll
                    move.l   #18,d0             19 16 pixel clusters + font part
scroll
                    move.w   (a1)+,(a0)+
                    move.w   (a1)+,(a0)+
                    move.w   (a1)+,(a0)+
                    move.w   (a1)+,(a0)+        16 pixels moved
                    dbf      d0,scroll          keep moving 16 pixel clusters

                    move.l   #18,d0             reset loop counter

                    move.w   (a2),(a0)+
                    move.w   2(a2),(a0)+
                    move.w   4(a2),(a0)+        16 pixels of the font
                    move.w   6(a2),(a0)+        character moved in
                    add.l    #8,a1              increment screen pointer, align with
                                                a0
                    add.l    #160,a2            next line of font

                    dbf      d1,scroll          do another line

                    add.l    #8,font_address    move 16 pixels forward in font

                    cmp.b    #$39,$fffc02       space pressed?
                    bne      main               if not, repeat main


                    jsr      restore

                    clr.l    -(a7)
                    trap     #1

                    include  initlib.s


                    section  data

font                incbin   font.pi1

screen              dc.l     0

font_address        dc.l     0

font_counter        dc.w     0

message             dc.b     "A  COOL  SCROLLER!    BUT  A  BIT  FAST,"
                    dc.b     "  SCROLLING  16  PIXELS  EACH  VBL."
                    dc.b     "    THAT'S 2.5 SCREENS EACH SECOND!"
                    dc.b     "             ",0

message_pointer     dc.l     message
```

There are really only two more small things that are new; the font counter and the message pointer. Also take note how I put in the scrolling message, by just using lots of dc.b, and a ‘0’ as an end control character. By changing the text here, you can obviously change the scroller message. Perhaps to the well known “Hello World!”, which I most deliberately avoided.

So what’s the font counter and message address? Well, the font counter keeps track of when it’s time to calculate a new address for a new character. This is set to 2, because every second loop, a whole character has been moved to the screen and the address for the next character in the message will have to be calculated. Had we scrolled 8 pixels each VBL, the font counter would have been set to 4 instead.

The message pointer is an index into the message. In order to know which character to get next time, we must have some pointer into the message. The message pointer begins by pointing to the message, which is good, since that's where the first character is. The first time through the main loop, the font counter will announce that an address for a new character will have to be calculated. That address is calculated and stored. The message pointer will then point to the next character in the scroller message, which is space, and so on. When the whole message has been scrolled through, the value 0 (not character '0') will be moved from the message. This is a signal that the message is at an end, and the message pointer will be reset, once again pointing to the start of the message. I hope it's understandable, the one tricky part is all the tests and branches, just walk through them a couple of times, slowly. You can use pen and paper for this, or the MonST.

We've begun to get somewhere. If you've paid attention so far, you'll have acquired quite some programming skills. There are still some basic things to cover, in order to be really self sufficient (mainly timers, double buffering, sprites and bit manipulation) but you're on a good way. Now might be the time to look at alternative sources and learn something from there. For example, you could begin to look at James Ingram's demo coding tutorials. That was where I began, I found them quite hard but now we've gotten more or less to the level where he begins his stuff. That is that from me right now. Upon request from mOdmate of Checkpoint, the next tutorial will probably be on timers, I think. This means we'll be able to remove the bottom and top borders, cool stuff!

8 Of Scrolling 8 Pixels Per VBL Using Double Buffer

"Be formless, shapeless, like water. Now you put water into a cup; it becomes the cup. You put water into a bottle; it becomes the bottle. You put it into a tea pot; it becomes the tea pot. Now water can flow, or it can crash. Be water my friend."

- Bruce Lee

In the last few days, I've had the great opportunity to get lots of introduction to the Atari scene. mOdmate of Checkpoint told me about #atariscne, and since then he's guided me through the stuff, giving me links to good sites and generally telling me what I need to know to orient myself. I've met some great people that have helped me understand things and being a better coder. Also, let's not forget the importance of Maarten Martens for converting this text file to html and banging me on the head whenever I take a wrong step. I could not write this stuff alone, lots of thanks to all of you who make this text possible. I also want to thank God, for giving me the luck and opportunity to be where I am, my mother for giving birth to me and always being there and all ... (end of Hollywood speech)

In order to get an even better understanding of the bit planes, I've done an 8 pixel scroller. The thing with this is that you must be careful not to misalign the bit planes, which we didn't have to worry about when scrolling 16 pixels per VBL. Since not to much have changed since the 16 pixel scroller, I thought I'd cover some other stuff as well.

First, I need to cover the shift command in order to be able to tell you about double buffering (there are more than one shift command, but they'll be covered later). The shift command will shift bits either left or right, as many "slots" as you want to. The command for shifting left is lsl, meaning Logical Shift Left, and right is lsr for Logical Shift Right. If you have a number in d0 and right shift, like so

```
move.l    #%10110001,d0      d0 = 177
lsr.l     #2,d0
```

then d0 will contain

```
          %00101100          44
```

all bits will jump two spaces to the right, and 0's have moved in from the left. Also note that this was the same as dividing 177 by 4 and throwing away the remainder. Left shifting will move bits to the left, and move 0's in from the right. Right shifting one is the same as dividing by 2. Thus a lsr.l #2 is the same as divu.l #4, and a lsl.l #2 is the same as a mulu.l #4. Only thing is that a shift is soooo much faster than a mulu or divu, but more on that later. It's very important to note how big the shift area is, if you have a data register filled with bits, but only shift a word, lsr.b, only the first 8 pixels will be affected. Like so

	Upper byte	Lower byte
d0 =	%10101010	%10101010

```
lsr.b     #4,d0
```

	Upper byte	Lower byte
d0 =	%1010101	%000001010

Note how the upper byte of the word was completely unchanged by the shift operation, since we used a lsr.b operation.

Now we can go on with double buffering. This is an extremely important technique. The screen is painted by an electron beam that goes from upper left, and then sweeps one horizontal line, down to the bottom right, just as the screen coordinates. Now, what happens if you start to make changes to the screen where the electron beam is painting? You will experience flicker or a distorted line or any other horrible thing. In short, when you write to screen memory, you'll most likely interrupt the electron beam in its work.

It is possible to change the area of memory that is the screen memory, any area of memory can be the screen memory actually. So for every VBL (or even often), we can change what area of memory is the screen memory. A solution begins to crystallize. We have to screen area sized areas of memory, one which is the actual screen memory (being shown on the monitor) and the other works as a buffer.

What we do is to update the buffer, while leaving the other screen alone, in this way, nothing will happen to the screen memory while the electron beam is painting. Then, just in the beginning of the next VBL, we make the buffer the screen memory and the screen memory the buffer. In this way, we will never paint to the actual screen memory. One can also all the memory that is being displayed for the physical base, and the area of memory not being displayed for the logical base. So far, we've gotten the address to the physical base by calling trap #2 of the XBIOS, if you call trap #3, you'll get the logical base. Usually, both of these point to the same memory area.

Instead of getting the physical address from the Atari, we will now define our own area of memory and input that address directly into memory. There's only one important thing to know about the screen memory; it must be on a 256 byte boundary (unless you have a Ste). What this means is that the start address of the screen memory must be a multiple of 256. This can be achieved by clearing the lower byte of the address, meaning that you'll need 256 bytes extra memory for your screen memory, so you can clear the lower byte. Why? Because clearing away the byte will clear away anything not multipliable by 256, the size of a byte.

So, how do we make a memory area the screen memory? Smack up the memory.txt file, and search for something appropriate, like "screen". We see this.

$FF8201	byte	Video screen memory position (high Byte)	R/W
$FF8203	byte	Video screen memory position (mid Byte)	R/W
$FF820D	byte	Video screen memory position (low Byte)	R/W (Ste)

Sure, ok, seems to be what we need. The low byte in $ff820d is for STe's only, and should be cleared at all times to avoid trouble. Then the middle byte of the screen address goes into $ff8203 and the high byte goes into $ff8201. In order to get the middle and high byte of the screen address, we need to shift the address. By shifting down the eight bits constituting the byte, we can easily move out bytes from the screen address by move.b commands.

	High byte	Middle byte	Low byte	
screen	%00010111	%01001101	%10111110	$174dbe

first we clear the low byte in order to put it on a 256 boundary.

```
move.l   #screen,d0
clr.b    d0
```

	High byte	Middle byte	Low byte
screen	%00010111	%01001101	%00000000

now we need to move the middle byte into $ff8203

```
lsr.l    #8,d0
```

	High byte	Middle byte	Low byte
screen	%00000000	%00010111	%01001101

```
move.b   d0,$ff8203
```

As you see, the middle byte gets shifted into the lower byte. With a move.b command the only thing we move is the lowest byte of d0. Thus, we have isolated the middle byte by shifting it into a more convenient position. Now for the last one.

```
lsr.w    #8,d0
```

	High byte	Middle byte	Low byte
screen	%00000000	%00000000	%00010111

```
move.b   d0,$ff8201
```

And that's it. We have now cleared the lowest byte of the screen address, and moved the middle and high bytes of it into the correct memory position. screen is now the screen memory. The compact code snippet looks like this.

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

                    clr.b    $ffff820d          clear STe extra bit
                    lsr.l    #8,d0
                    move.b   d0,$ffff8203       put in mid screen address byte
                    lsr.w    #8,d0
                    move.b   d0,$ffff8201       put in high screen address byte

                    section  bss
                    ds.b     256                256 byte clear buffer
screen              ds.b     32000              the screen
```

Now, this doesn't make for any double buffer at all, since we're only using one screen. In order to achieve double buffering, we need two screen areas, and two pointers to point to each area. In each VBL, one screen is made into screen memory, and then the pointers are flipped so that the other screen is made screen memory for next VBL. This really makes what you see on the screen appear 1/50th of a second slower than what you draw.

```
                    prepare addresses
                    make next and last point to screen1 and screen2
main
                    wait VBL

                    move.l   next,d0
                    make address in d0 screen address

                    move.l   last,a0
                    move.l   next,a1            load screens
                    move.l   a1,last            and flip them for next time around
                    move.l   a0,next            double buffering :)
* loads the screen addresses and flips them around
                    do your stuff, like putting graphics to the address in a1
```

```
                    repeat main loop

last                dc.l    0
next                dc.l    0

                    ds.b    512
screen1             ds.b    32000
screen2             ds.b    32000
```

I also thought we might mention timing as well. This is quite the issue really, as you must have understood, you can't perform an infinite number of instructions. Important information is in two appendixes. Appendix F – MC68000 Instruction Execution Times, on page 153 – explains how much time it takes to do each instruction. This can vary greatly, for example, a division takes way over 100 clock cycles, and a shift takes under 10, so you see, it's a good thing to replace your divu's with lsl's if possible. Also, when you can, work with byte or word size, instead of long, since this saves some time also. Clock cycle is the quantity in which "time" is measured. Each instruction takes a certain amount of clock cycles.

Appendix G – Pixel Timings, by Jim Boulton, on page 157, was extracted by me from the ST Internals text file by Jim Boulton. One interesting thing to note there is the amount of clock cycles per VBL; 160256. This is a very exact number, and if your main loop ever takes more time than that, you're screwed (if you work with VBL main loops as we've done so far that is). One way to get a graphical pointer of how much time your main routine does take, is to change the background colour just at the start of the routine, then change it back in the end.

Let's say we have a routine that takes 80000 clock cycles, our original background is black, but in the beginning of our main loop, we set it to red. What will happen is that the electron beam will paint red background, but when our 80000 clock cycles worth of instructions have taken place, the background is switched back to black, which means that for the time it takes to wait for the next VBL, the electron beam will paint black. So, in this case, the screen would be half red background and half black background. If we use this technique, we'll see exactly how much time our main routine takes. The example program in this tutorial takes up most of the processor, which leaves little time for other stuff to be done. Granted, the scroller is completely un-optimized.

Phew, now we have covered lots of small things of big importance. Finally, now comes the 8 pixel scroller part. Just look at the source code, it's well commented. Nah, I'm just kidding with you, of course I'll explain. Since we now want to scroll 8 pixels, this means for starters that we need to move bytes. The first byte represents the first 8 pixels, and the second the coming 8 pixels. Then, the third word again has to do with the first 8 pixels, and the fourth word has to do with the 8 coming pixels and so on. Thus, we cannot simply barge in and do some scroll loop. We need to move every second byte.

Index	First byte	Index	Second byte	
0	%11000000	1	%00000000	first word
2	%11000000	3	%00000101	second word
4	%01000000	5	%00000110	third word
6	%01100000	7	%00000000	fourth word
0-7	$3F800000	8-15	$00000642	pixels

Index	First byte	Index	Second byte	
8	%00000110	9	%00100000	first word
10	%00000010	11	%00100100	second word
12	%00000000	13	%10000010	third word
14	%00100010	15	%00010000	fourth word
16-23	$008001B0	24-31	$40380240	pixels

It is tempting to read the memory top down, but this is not so, it is to be read from left to right. So index 5 for example is the second byte in the third word, and affects pixels 8 – 15. The memory without comments look like this, split into bytes for ease of reading.

%11000000, %00000000, %11000000, %00000101, %01000000, %00000110, %01100000, %00000000, %00000110, %00100000, %00000010, %00100100, %00000000, %10000010, %00100010, %00010000, …

So in order to scroll 8 pixels, index 0, 2, 4 and 6 will de dropped, because they represent the first 8 pixels. Then index 1, 3, 5 and 7 will be moved into index 0, 2, 4 and 6. Then index 8, 10, 12 and 14 will be moved into index 1, 3, 5 and 7. Then index 9, 11, 13 and 15 will be moved into index 8, 10, 12 and 14. This will make pixels 0-7 to drop, 8-15 to be moved into 0-7, 16-23 will be moved into 8-15 and 24-31 will move into 16-23. After these move instructions, the memory will look like this

Index	First byte	Index	Second byte	
0	%00000000	1	%00000110	first word
2	%00000101	3	%00000010	second word
4	%00000110	5	%00000000	third word
6	%00000000	7	%00100010	fourth word
0-7	$00000642	8-15	$008001B0	pixels

Index	First byte	Index	Second byte	
8	%00100000	9	…	first word
10	%00100100	11	…	second word
12	%10000010	13	…	third word
14	%00010000	15	…	fourth word
16-23	$40380240	24-31	…	pixels

It is of the utmost importance that you realize why this is so. If you do not, set yourself down and work it out until you get it and understand it 100%. Without understanding this, you'll not understand bit planes, without understanding bit planes, you can't understand how the graphics on the Atari works. Expressed in code, this will be (a0 points to screen memory)

```
move.b   1(a0),(a0)
move.b   3(a0),2(a0)
move.b   5(a0),4(a0)
move.b   7(a0),6(a0)        8 pixels moved
move.b   8(a0),1(a0)        watch carefully!
move.b   10(a0),3(a0)
move.b   12(a0),5(a0)
move.b   14(a0),7(a0)       first 4 word area filled
move.b   9(a0),8(a0)        start of second 4 word area
move.b   11(a0),10(a0)
…
```

and so on. So first, four bytes are moved just one step to the left, but then you need to go into the next 4 word area, to fetch the bytes that go into the second area of the first 4 word area and so on. This is the theory behind 8 pixel scrolling, I don't think I can explain it better than that. This is the source code for the scroller.

```
jsr      initialise

move.l   #screen1,d0        put screen1 address in d0
clr.b    d0                 put on 256 byte boundary
move.l   d0,next            store address
add.l    #32000,d0          next screen area
move.l   d0,last            store address
```

```
                    movem.l  font+2,d0-d7
                    movem.l  d0-d7,$ff8240     palette moved in

main
                    move.w   #37,-(sp)         wait vbl
                    trap     #14
                    addq.l   #2,sp

                    move.l   next,d0

                    clr.b    $ffff820d         clear STe extra bit
                    lsr.l    #8,d0
                    move.b   d0,$ffff8203      put in mid screen address byte
                    lsr.w    #8,d0
                    move.b   d0,$ffff8201      put in high screen address byte

                    move.w   #$707,$ff8240     to see clock cycles

                    cmp      #0,font_counter   check if new character in message
                    bne      has_character     if not, skip get new character

                    move.w   #4,font_counter   reset font_counter
* we need to point to a new character in the font

                    move.l   message_pointer,a0 pointer into the message
                    clr.l    d0                clear, just to be sure
                    move.b   (a0),d0           put letter ascii value in d0

                    cmp      #0,d0             end of message?
                    bne      not_end           if not, branch

                    move.l   #message,message_pointer      reset message_pointer
                    move.l   message_pointer,a0
                    clr.l    d0                clear, just to be sure
                    move.b   (a0),d0           put letter ascii value in d0

not_end
* now we have a character in d0 for sure
                    add.l    #1,message_pointer point to next character

                    add.b    #-$20,d0          align ascii with font number
                    divu     #10,d0            10 letters per row

                    move.w   d0,d1             d1 contains y value
                    swap     d0
                    move.w   d0,d2             d2 contains x value

                    mulu     #16,d2            16 bytes for each letter
                    mulu     #32,d1            32 lines per row
                    mulu     #160,d1           160 bytes per row

                    move.l   #font+34,a0       put font screen start in a0

                    add.l    d2,d1             add x and y value together
                    add.l    d1,a0             a0 points to correct letter

                    move.l   a0,font_address   store calculated pointer

has_character
                    add.w    #-1,font_counter

                    move.l   last,a0
                    move.l   next,a1           load screens
                    move.l   a1,last           and flip them for next time around
                    move.l   a0,next           doubble buffering :)
                    move.l   font_address,a2   font address

                    move.l   #31,d1            32 lines to scroll
                    move.l   #18,d0            19 16 pixel clusters + font part
```

```
scroll
                    move.b   1(a0),(a1)
                    move.b   3(a0),2(a1)
                    move.b   5(a0),4(a1)
                    move.b   7(a0),6(a1)       8 pixels moved
                    move.b   8(a0),1(a1)       watch carefully!
                    move.b   10(a0),3(a1)
                    move.b   12(a0),5(a1)
                    move.b   14(a0),7(a1)      first 4 word area filled

                    add.l    #8,a0             jump to next 4 word area
                    add.l    #8,a1             jump to next 4 word area
                    dbf      d0,scroll         keep moving 16 pixel clusters

                    move.l   #18,d0            reset loop counter

                    move.b   1(a0),(a1)
                    move.b   3(a0),2(a1)
                    move.b   5(a0),4(a1)
                    move.b   7(a0),6(a1)       152 pixels scrolled

                    move.b   (a2),1(a1)        now last 8 pixels from font
                    move.b   2(a2),3(a1)
                    move.b   4(a2),5(a1)
                    move.b   6(a2),7(a1)       8 pixels from font

                    add.l    #8,a0             point to beginning of next line
                    add.l    #8,a1             point to beginning of next line
                    add.l    #160,a2           next line of font
                    dbf      d1,scroll         do another line

                    add.l    #1,font_address   next byte in font
                    cmp      #2,font_counter   see if it's time to change
                    bne      font_increment
                    add.l    #6,font_address   align to next 16 pixels
font_increment

                    move.w   #$0,$ff8240       black background again

                    cmp.b    #$39,$fffc02      space pressed?
                    bne      main              if not, repeat main

                    jsr      restore

                    clr.l    -(a7)
                    trap     #1

                    include  initlib.s


                    section data

font                incbin   font.pi1

screen              dc.l     0

font_address        dc.l     0

font_counter        dc.w     0

message             dc.b     "A COOL SCROLLER!   MOVING 8 PIXELS PER VBL "
                    dc.b     "AND USING DOUBBLE BUFFERING    ",0

message_pointer     dc.l     message

next                dc.l     0
last                dc.l     0
```

```
                    section bss

                    ds.b     256
screen1             ds.b     32000
screen2             ds.b     32000
```

Not too much has been changed since the 16 pixel scroller. In the beginning, there's the code for setting up two screen areas. Then, in the main routine, we put one screen address in. Notice also how the font_counter is now 4 instead of 2, because we only need new font data every fourth VBL. The scroller part however is completely new, not surprising is it? It begins with loading both screen areas into a0 and a1, and then flips them for next time around. Data is moved as described above for 19 loops, this means 304 pixels are moved, the last 16 need special care though.

First 8 pixels scrolled as usual, but the last 8 must come from the font. This is also not to strange, since every second byte is moved into the second bytes of the words on the screen. Then 1 is added to the font address, to point to the second bytes in the words. However, this won't quite do, as you may know. The step from the second byte of the first 16 pixels to the first byte of the coming 16 pixels is a bigger jump than 1, as described above.

In order to make this bigger step, I test the font_counter, to see if it's time, and then add another extra 6 to the font, making it point to the right place. If we don't do this extra addition, 16 pixels will be moved in from the font ok, but when pixels 16 – 24 are about to be moved, the font address will point to index 2 (meaning the first 8 pixels again) instead of index 8 into the font memory. Just scroll up to the memory example, then work through the scroll loop on a piece of paper or in your head and it will hopefully become obvious. If it doesn't, mail me.

That, I think, was that. The big problem here is the understanding and alignment of bytes in the bit plane. What to keep in mind really is that first, take every second byte, then jump a bit to get on the next 16 pixel boundary, then continue in that way. Indexing goes like 0, 1, 8, 9, so to speak. Thus, every second time there's a little gap. Since I didn't do any timers this tutorial, maybe we'll do them next time.

9 Of Revealing The Unseen And Expanding Our Consciousness Without The Use Of Illegal Drugs

"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

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 EOR (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		
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 EOR 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	%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 ANDing 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 EOR, is good for different things. As we have seen, AND is good for clearing bits. EOR is good for many things, but the most obvious one is flipping bits, if you EOR 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 EOR 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 Atari 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 also 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 simplest 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

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 addresses. 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 this.

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 from 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 (Phase Alternating Line) to NTFS (National Television Standards Committee) 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 Appendix B – Hardware Register Listing, by Dan Hollis, on page 132 – 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 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 ORing 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 229th 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 229th scan line and kill off the border; khazam! (note: if the top border is not killed, the numbers are 199th and 200th 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 229th 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 possible 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.

10 Of Lighting A Candle (And Casting A Shadow)

"I fear I will never find anyone

I know my greatest pain is yet to come

Will we find each other in the dark

My long lost love"

- Nightwish, Beauty of the Beast

Hello again! I just got four Jaguar games from Aldebaran, he's away over the midsummer feast so he was kind enough to leave me four Jaguar titles, since I own a Jaguar but have no games, heh ... I thought I'd wait until tomorrow before knocking myself out though, and do some good right now (or perhaps it's because the TV is blocked). Ahh, I've gotten hold of the new Nightwish CD; Century Child, if you don't own it already, make sure to do so! They play Finnish metal combined with real cute songs and the main singer is schooled in classic opera, so they have a real cool sound and are probably my favourite musical artists.

We're up to tutorial number 10; which makes me very glad and proud over the work achieved. This had not been possible without the support of readers and other VIP's. To celebrate the tenth "anniversary", I'm going to give you something special; a putpixel routine. Actually, that statement was almost meant as a kind of ironic, funny statement. I mean, not until tutorial number 10 do we learn how to code a putpixel routine, in the PC's MCGA mode for example, this is something you hardly have to learn, it's implied. However, as anyone who knows this much about programming the ST will know, coding putpixel for the ST is a pain in the butt.

The putpixel will hopefully be a prequel to a tutorial on sprites. Sprites, by the way, are anything that moves on the screen, such as a little spaceship, a funny rabbit or just a bouncing ball. A putpixel routine is a routine designed to put a single pixel on the screen, so on the ST, it lights a single dot in one of 16 colours. To achieve this, four bits have to be changed in four different places in memory, and nothing else must be changed in the screen memory, else too much will be changed.

Let's first see how we will know which pixel to change. We want to be able to provide information in coordinates, like pixel 160,100, which is about the middle of the screen. The ST would treat any such coordinates with a big question mark, so we have to find a way to translate the coordinates. All pixels come after one another in the screen memory, starting with the top left one, and ending with the bottom right.

This means that X value is of course worth 1 position, since pixel 2,0 is the third pixel on screen. Each Y value is worth as much as the number of X coordinates on one scan line, in the case of ST low resolution; 320. So if we have the coordinate 160,100, that would be the 160 + 100 * 320 = 32161th pixel on screen (one extra is added to the value since we start counting from 0). The total number of pixels on screen is 320 * 200 = 64000 pixels, which is about twice as much as 32160, so the formula seems to work. However, this won't work on a ST, because we can't simply count pixels that easily.

The information for a single pixel is contained in four bits in four consecutive words, one bit in each word. So, instead what we need to know is at what word the first bit is, and which bit exactly it is we want to deal with. There are 16 bits in each word, by dividing our X value with

16, we will get the number of word clusters to count in the result, and the exact bit in the remainder.

Here's why: suppose we want the 19th pixel, this would mean jumping over the first word cluster, which contains information for 16 pixels, and then manipulate the third most significant bit in the next word cluster. 19 / 16 = 1.1875, which means we get the result 1 and the remainder 16 * 0.1875 = 3, it works! So, given the ST's screen memory configuration, how exactly will we treat coordinate 160,100?

We assume the screen memory points to the start of the screen memory. First, the Y coordinate, which is the simplest; each scan line is 160 bytes, so multiply the Y coordinate with 160 and add it to the screen address. Then divide the X value with 16, multiply the result with 8 and add to the screen memory. Why 8? Because each word cluster that we are to jump over is 8 bytes; two bytes to a word and four consecutive words. Now, the screen memory points to the first word of the four consecutive words we need to alter, in each word we want to alter one bit. The exact bit is obtained from the remainder of the X division. There may be other intricate methods, but this one is robust, straightforward and good for learning. Supposing a0 points to the beginning of screen memory, d0 holds the X-coordinate and d1 holds the Y-coordinate, this is how it's done.

```
mulu.w    #160,d1          160 bytes to a scan line
add.l     d1,a0            add y value to screen memory
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,a0            add the cluster part to screen memory
clr.w     d0               clear out the cluster value
swap      d0               put the bit to alter in low part of
                           d0
```

There is some magic worked here with high and low parts of the data registers. With high and low part, we mean the two first, and two last bytes of the register, thus the high part is the first 16 bits, and the low part the last 16 bits (reading from left to right). By performing instructions with word size, you only affect the lower part of a register, and leave the higher part unchanged.

The divu instruction, leaves the result in the low part and the remainder in the high part of the register. After the divu, the move.w will only move out the lower part, the cluster part, of d0. Following is a multiplication on the cluster value, and an addition to screen memory. Finally, clear out the cluster part of d0, and a swap instruction. The swap instructions flips the high and low part of a register, so now d0 neatly holds only the value for the bit to be changed, and a0 points to the correct place.

So, now that we know where to change, how do we know what to change? We will have a value between 0 and 15, that is supposed to be put in those four bits in the screen memory (the colour of the pixel). We can't just move in the data, we could devise some plan with bset and bclr instructions, but that may be clumsy and will probably involve branches for testing, which is slow. Instead, we will use our knowledge of masks and Boolean algebra to solve the problem.

By putting the colour data in the high part of a register, we can then rotate the least significant bit of the colour into the lower part, and then do a shift on only the lower part of the register, to put the colour bit in the correct place; mask prepared! Suppose we have the colour value in d2, and the number of bits to change in d0, obtained from the example above, this is how it works.

```
swap      d2               put colour value in high part
lsr.l     #1,d2            put one bit of colour in shiftable
                           position
lsr.w     d0,d2            shift by number in d0
```

Memory will perhaps look something like this, d1 = 5.

	High part	Low part
d2	%0000000000000000	%0000000000001011

```
swap d2
```

	High part	Low part
d2	%0000000000001011	%0000000000000000

```
lsr.l #1,d2
```

	High part	Low part
d2	%0000000000000101	%1000000000000000

```
lsr.w d0,d2
```

	High part	Low part
d2	%0000000000000101	%0000010000000000

Ah, now the lower part of d2 will hold a terrific mask, we have one bit set, the one bit that we want to alter in screen memory, a simple ORing of the mask will make sure that the bit is set in screen memory, then we just add two to the screen pointer, and repeat the process. Wrong! Problem is, depending on our value, we want to either clear or set the bit, as you can see, on the third run through, the bit should be cleared, not set (the third bit counting from the left in the colour value is 0). If we just OR in the mask, and the bit we want to clear in the screen memory, is set to begin with, we will end up with a set bit where we want a cleared bit. Buh, that sounded awful, example! This is how it will look the third run through

	High part	Low part
d2	%0000000000000001	%0000000000000000

note how the fifth most significant bit in the lower part is not set.

Screen memory	%1111111111111111

By ORing d2 with the screen memory, we won't be clearing out the fifth most significant bit in the screen memory, although we need it to be cleared in order for the pixel to have the correct value. So, before ORing in our mask, we need to make sure the bit is cleared. This is done by ANDing in a mask with all bits set except the one bit we want to change. Mask preparation looks like this.

```
move.w    #%0111111111111111,d1
ror.w     d0,d1
```

d1	%11111011111111111

The ror instruction, for ROtateRight, will rotate the register, making sure that whatever goes out the right (or left) will then come in to the left. Thus, if the least significant bit is 0, a 0 will me moved in the most significant bit, if it's a 1, a 1 will be moved in. The difference between a logical shift and a rotate, is that a logical shift will move in 0's, while rotate will move in whatever went out. Examine Appendix F – MC68000 Instruction Execution Times, on page 153 – if you wish to further your knowledge on this, there are also arithmetic shifts, but I

don't use them here. Now, by first ANDing in the clear mask, we can then safely or in our pixel mask, like so.

```
                    swap     d2                    colour in the high part of d2
                    move.w   #%0111111111111111,d1
                    ror.w    d0,d1                 clear mask prepared

                    lsr.l    #1,d2                 shift in the next colour bit
                    ror.w    d0,d2                 shift colour bit into position
                    and.w    d1,(a0)               prepare with mask (bclr)
                    or.w     d2,(a0)+              or in the colour
                    clr.w    d2                    clear the old used bit
```

Then just repeat that over and over, or rather, three times more. The only thing not covered before is the last line, clearing out the old used bit, without this, remnants might be left on the next time around. This is one way of putting a pixel to screen. Bit planes make a putpixel routine so incredibly slow and clumsy. This though, is just a generic putpixel routine, a pixel routine designed for a specific purpose might be much faster, involving only one bit plane perhaps. You don't have to mess around with all four bit planes every time, say you only want to use four colours for your stuff, then just leave two bit planes alone, since they aren't needed, this will speed up things. This is the entire putpixel routine.

```
putpixel:
* a0 screen address
* d0 X coordinate
* d1Y coordinate
* d2 colour

                    mulu.w   #160,d1           160 bytes to a scan line
                    add.l    d1,a0             add y value to screen memory
                    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,a0             add the cluster part to screen memory
                    clr.w    d0                clear out the cluster value
                    swap     d0                put the bit to alter in low part of
                                               d0

* now a0 points to the first word of the bitplane to use
* d0.w              holds the bit number to be manipulated in the word

                    swap     d2                colour in the high part of d2
                    move.w   #%0111111111111111,d1
                    ror.w    d0,d1             clear mask prepared

                    rept     4                 do this 4 times
                    lsr.l    #1,d2             shift in the next colour bit
                    ror.w    d0,d2             shift colour bit into position
                    and.w    d1,(a0)           prepare with mask (bclr)
                    or.w     d2,(a0)+          or in the colour
                    clr.w    d2                clear the old used bit
                    endr

                    rte                        return form putpixel
* end putpixel
```

That was that. A nice putpixel routine to use for our convenience, slow as hell because of two multiplications and one division. Because this is a tutorial, and I want to push against practical use, I've also written a stupid little program that puts 50 pixels a second on the screen, like a screen saver. However, that program also includes some nice tricks so read on!

The ST has a number of system variables, they are found at very low addresses, starting at $400 and ending at $516. Like the name suggests, these variables contain lots of special information on the system, and they can provide quite the shortcut to finding out some information. For example, $44e, called _v_bas_ad, is a long word containing a pointer to the

screen memory (the logical screen memory). If you just want a quick and dirty program, like this one, and want to find out the screen address without traps, or hooking it up yourself, simply read the value here.

```
                    move.l    $44e,a0           a0 points to screen memory
```

There are some other useful system variables, which will be presented when the need arises. If we want to have a screen saver like program, we want to be able to output random pixels, right? So we need a way to generate a random number. Random numbers are usually obtained from reading the system clock, and then applying some algorithm to the obtained value. We don't want to mess with that, especially not when there is a very nice trap that will do the job nicely for us. Trap number 17 in XBIOS will generate a 24-bit random number and put it in d0.

```
                    move.w    #17,-(a7)         trap number 17, random
                    trap      #14               call XBIOS
                    addq.l    #2,a7             clean up stack
* random number in d0
```

Well, a random number of 24-bits is a number between 0 and 16777216. We want values in the range of 0 to a maximum of 319; 0-15 for the colour, 0-319 for the X coordinate and 0-199 for the Y coordinate. So how can we get the random number "down to our level", so to speak? We could put the random call in a loop, and in the end of each loop check the random value, and if it is to big, just repeat the loop. While this would work, it would be so incredibly slow because the odds of the value falling within parameters are extremely small. However, by first lessening the value, we gain tremendous time.

As we so well know, the colour value consists of 4 bits. By ANDing the random value with %1111, we effectively set all bits to 0 except the first four (which may be 0 or 1, depending on the initial value). Thus, our initial random value of 24-bits has been reduced to a 4-bit value, making it perfect for our needs. The X and Y coordinates however, are a bit different, since their representation does not consist of a complete set of bits (numbers that do so are the ones immediately before the powers of 2, i.e. 1, 3, 7, 15, 31, 63, 127, 255, etc). We can't simply mask off the X co-ordinate's unnecessary bits like we did the colour, rather we have to keep one bit more than what is needed.

The value 319 uses 9 bits, so we will have to AND the X coordinate with %111111111, but %111111111 = 511, so after masking off the bits in our random value, we'll have a number between 0 and 511. Now, we must use a loop to check the value, and make a new random number if it should prove to be over 319. The odds for the value of being within parameters are greatly increased though.

```
get_x
                    move.w    #17,-(a7)
                    trap      #14
                    addq.l    #2,a7             get random number
                    and.l     #%111111111,d0    make it maximum 511
                    cmp       #319,d0
                    bgt       get_x             loop until d0 < 320
```

The instruction bgt will branch if the value compared is greater than. This loop then, will loop until the value in d0 is not greater than 319. The Y coordinate is obtained by doing much the same thing, but we only need to AND with 8 bits, because the Y coordinate should be < 200, and 8 bits make up 255. There, all the theory we need, this is the complete source of the program.

```
                    jsr       initialise

                    move.l    $44e,a0           a0 poins to screen memory
                    move.w    #0,$ff8240        black background
                    move.l    #7999,d0
                    clear
                    clr.l     (a0)
                    dbf       d0,clear          clears screen to colour 0
```

```
main
                    move.w   #37,-(a7)
                    trap     #14
                    addq.l   #2,a7              wait retrace

                    get_x
                    move.w   #17,-(a7)
                    trap     #14
                    addq.l   #2,a7              get random number
                    and.l    #%111111111,d0     make it maximum 511
                    cmp      #319,d0
                    bgt      get_x              loop until d0 < 320
                    move.l   d0,d7              store x coordinate

                    get_y
                    move.w   #17,-(a7)
                    trap     #14
                    addq.l   #2,a7              get random number
                    and.l    #%11111111,d0      make it maximum 255
                    cmp      #199,d0            loop until d0 < 200
                    bgt      get_y
                    move.l   d0,d6              store y coordinate

                    move.w   #17,-(a7)
                    trap     #14
                    addq.l   #2,a7              get random number
                    and.l    #%1111,d0          make it maximum 15
                    move.b   d0,d2              put colour number in d2

                    move.l   d7,d0              put x coordinate in d0
                    move.l   d6,d1              put y coordinate in d1

                    move.l   $44e,a0            a0 points to screen memory
                    jsr      putpixel           put pixel on screen

                    cmp.b    #$39,$fffc02       space pressed?
                    bne      main               if not, repeat main

                    jsr      restore

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


putpixel:
* putpixel routine
* a0 screen adress
* d0 x-coordinate
* d1 y-coordinate
* d2 colour

                    mulu.w   #160,d1            160 bytes to a scan line
                    add.l    d1,a0              add y value to screen memory
                    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,a0              add cluster part to screen memory
                    clr.w    d0                 clear out the cluster value
                    swap     d0                 bit to alter in low part of d0

* now a0 points to the first word of the bitplane to use
* d0.w              holds the bit number to be manipulated in the word

                    swap     d2                 colour in the high part of d2
                    move.w   #%0111111111111111,d1
                    ror.w    d0,d1              clear mask prepared

                    rept     4                  do this 4 times
```

```
                    lsr.l    #1,d2              shift in the next colour bit
                    ror.w    d0,d2              shift colour bit into position
                    and.w    d1,(a0)            prepare with mask (bclr)
                    or.w     d2,(a0)+           or in the colour
                    clr.w    d2                 clear the old used bit
                    endr

                    rts
* end putpixel


                    include  initlib.s
```

Yes, first a normal initialisation, then the neat trick of putting the screen address in a0, followed up by putting the background black and clearing the screen. Then, a main routine, the question here is why I didn't I use the $70 as described in tutorial 9. The reason is a bit farfetched, but valid. Because of the random loops, there is a theoretical possibility of the main routine taking longer than 1/50th of a second, it's virtually impossible, but it could happen. If this were to happen, the $70 vector would be called while the previous one were still being executed, resulting in a crash. With the method I use here, however, there is no danger of a crash.

Obtaining the X coordinate, as described above, only new thing is storing the coordinate in d7. This is because the coming random trap for the Y coordinate will destroy everything in register d0, and then some, register d7 however, is safe. Same goes for the Y coordinate. Finally, the colour value is obtained, and moved over to d2, then the X and Y coordinates are moved into their respective registers. These registers could be anything, or a variable or whatever storage possibility, but the putpixel routine is designed to have the X coordinate in d0 and the Y coordinate in d1, so this is how it's supposed to be. After the screen address has been put in a0, all is set for the putpixel call.

The putpixel routine is exactly as described above, so nothing new there. Signing off with a check for a pressed spacebar, and that concludes the program. Note how I put the putpixel routine in its' own subroutine, instead of including it in the main program, which could also have been done. This results in tidier code, the downside being that it takes more time to execute, but time is no issue here.

Speaking of time, I actually think that I'll fill up some space here with a bit on optimisation, something that will have to come one day or another anyway. There are two multiplications and one division in the putpixel routine, horrible. These can be replaced with shift instructions, but it's a bit tricky. Each shift either doubles or halves the value in the register. So how do we do a multiplication of 160 and a division by 16, where we also keep the remainder?

First, the Y part; here, we want to have a result equalling d1 * 160. 160 is not a value you may shift by, since all shift will produce multiplication results of 2, 4, 8, 16, 32, 64, 128, 256 and so on. However, 128 + 32 = 160, the value we want to multiply with, and when things come to multiplication, we are allowed to split the multiplication in two and add the result; d1*160 = d1*32 + d2*128. All we have to do is copy our Y coordinate into another register, shift one register with 5 (multiplication of 32), shift the other with 7 (multiplication of 128) and add the results together.

```
                    move.w   d1,d3              copy Y coordinate
                    lsl.w    #7,d1              mulu #128,d1
                    lsl.w    #5,d3              mulu #32,d3
                    add.w    d3,d1              add results together
                    add.l    d1,a0              add result to screen address
```

Note the word size used in all operations. There may still be garbage in the upper part of d3, but this is never touched in any of the operations. Since the maximum value we will handle is 199 * 160 = 31840 is less than the maximum for a word size, which is 2^{16} = 65536, it's ok to only use word size instructions, it also saves time. Our mulu instruction would take a maximum of 70 clock cycles, but in this case I think it's 42. The technique of shifting takes

$$12+6+2\times 7+6+2\times 5+8=58$$

Heh, seems we wasted time rather than saving. Let that be an important lesson, sometimes the job's just not worth doing. :-)

So now the X part, first, put the thing in the upper part of d0 with a swap. Now, with a right shift of 4, we will effectively divide the number by 16, which is what we want to achieve, the result will be in the upper part, and the remainder will be in the highest bits in the lower part.

Now, what we need to do is simply to put the remainder down in the lowest bits in d0, so we right shift by 12. The reason for right shifting by 12 is that the remainder takes up a maximum of 4 bits (remainder maximum is 15, %1111), and 12 + 4 = 16 which is the number of bits in the lower part of a data register. Unfortunately though, you can't shift by 12 when shifting with a number, so we'll just have to divide the shift in one 8 and one 4 part, 8 being the highest number you may shift by.

Swap down the result in the lower part, and shift it left by 3 in order to multiply with 8. We make sure to keep the operation word size in order not to affect the remainder in the upper part. Then, add the result to the address register, but only use a move with word size, in order to only add the multiplied result, and leave the remainder well alone. Lastly, a clear out of the result part and a swap to put everything right for the next part of the putpixel.

```
swap     d0              put in upper part
lsr.l    #4,d0           divide by 16
lsr.w    #8,d0           shift down remainder …
lsr.w    #4,d0           … by 12 bits total
swap     d0              result in lower part
lsl.w    #3,d0           multiply with 8
add.w    d0,a0           add result to screen address
clr.w    d0              clear out result
swap     d0              put remainder in lower part
```

That was that, now let's see if this optimisation did us some good. Unoptimized takes about

$$140+6+4+40+12+4+4=210$$

The division is an approximation. Also, I don't think we really need to move some data to d1 to manipulate it, so the unoptimized could do some optimization too, but that's not too important. Now let's see what the shift-optimized part will take

$$4+8+4\times 2+6+8\times 2+4+6+3\times 2+8+4+4=88$$

Even though I'm a bit unsure of some values here, it's obviously quite a save in any case. That was a little taste on how to optimize easy, just replace multiplications and divisions with shifts, sometimes quite a saving, but not always.

11 Of Making The Mountain Move To Mohammed

"Is it possible that we two, you and I, have grown so old and inflexible that we have outlived our usefulness? Would that constitute … a joke?"

- Star Trek VI, the Undiscovered Country

Well well, finally, as promised, we will delve into the technique of sprites; the essence of a platform or shoot-em-up game, and lots of other stuff. In fact, anything that needs something moving that is not 3D or real time rendered (that is, it's being drawn while the program runs, and not stored previously as a picture). It was really challenging and great fun to code this one, and it's probably the most satisfying coding experience ever, I hope I can convey the knowledge it brought me.

In the last tutorial, we learned something on pixels, in order to be able to address a single pixel anywhere, the data must be shifted into a correct position. Why is this? Because, each instruction except the bit instructions, deal with at least byte size. What it means is that if we use instructions with byte size, all pixels "snap" at 8 pixels, because that's the minimum addressable size. However, by shifting the data before using it in graphic instructions, we can in a way address any pixel we want to.

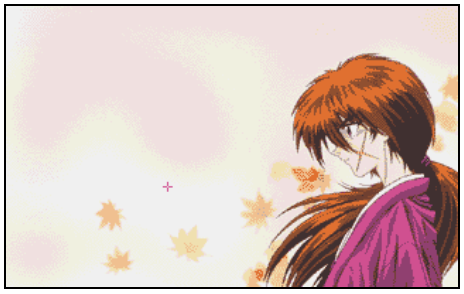


Figure 3 – AUTUMN.PI1

I actually suggest you load up the pre-assembled program, and both the picture files that comes with the tutorials, the pictures being AUTUMN.PI1 and SPRITE.PI1. A little note on the pictures, they are in STe palette, meaning that they will look a bit ugly on a ST, but STeem should handle this nicely. Yes, the character seen is the same one as in TUT9: Kenshin. He's the main character in a Japanimation, a former assassin for the government who now tries to atone by living a quiet life and helping people. This series is awesome and has given me much inspiration, the first Kenshin OVA series is one of the most beautiful pieces of art I've ever seen.

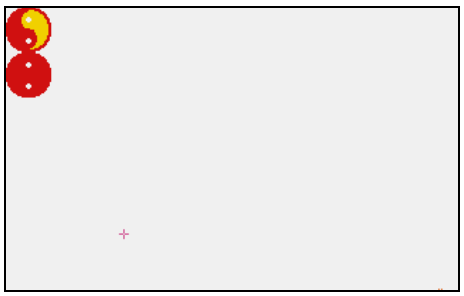


Figure 4 - SPRITE.PI1

So, after you've been impressed by the Tai Ji symbol (a.k.a. Yin and Yang symbol, Yin and Yo in Japanese) bouncing around the screen, you are eager to learn for yourself, right? As you can see, the background is provided in the AUTUMN.PI1, and the bouncing ball, which is the sprite, is in SPRITE.PI1. Actually, only 14 colours are used for the background, the last two being reserved for the sprite, this isn't necessary and the sprite may well share colours with the background. The sprite seems to appear twice, in the SPRITE.PI1, there are two balls, one of them is the sprite mask, if confusion occurs, just read on.

Painting the background is easy, just smack in the pixel data and set the palette, bouncing will be dealt with later, what we need to focus on now is getting the sprite nicely on the screen, and being able to put it anywhere on the screen, preferably expressing the location in X and Y coordinates for human compatibility. How exactly to put the sprite data on screen, the most obvious choice is a move instruction. This won't do at all though, check this out.

Screen memory		
%00000000	%00001110	first word
%00000000	%00000000	second word
%00000000	%00001011	third word
%00000000	%01010101	fourth word
Pixel colours		
$00000000	$0808595C	

Sprite data		
%00000000	%00000001	first word
%00000000	%00100000	second word
%00000000	%00000000	third word
%00000000	%00001010	fourth word
Pixel colours		
$00000000	$00208081	

Now, if we move the sprite data onto the graphics memory, we get

Screen memory		
%00000000	%00000001	first word
%00000000	%00100000	second word
%00000000	%00000000	third word
%00000000	%00001010	fourth word
Pixel colours		
$00000000	$00208081	

Move instructions destroy all data and replace it with new, in other words, the background is completely lost and the sprite has taken over completely. Doing it like this will also create an ugly squared looking sprite, since the sprite background will not be transparent (actually rectangular, but more on this below), as you can see on Figure 5. This will not do.

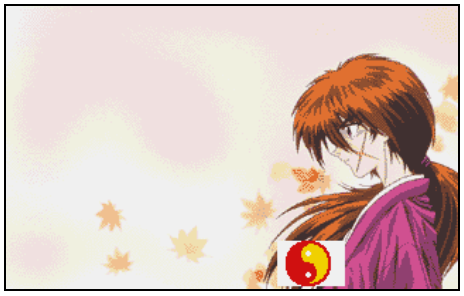


Figure 5 – Result of simply MOVE'ing the sprite data

OR instructions, on the other hand, will not overwrite the original data, we try an OR instruction with the above configuration.

Screen memory		
%00000000	%00001111	first word
%00000000	%00100000	second word
%00000000	%00001011	third word
%00000000	%01011111	fourth word
Pixel colours		
$00000000	$0828D9DD	

Dang! By ORing in the sprite, we mixed the sprite with the background, this is also bad since the sprite will not look as it should, although it will create quite a nice effect and is good if you simply want a "colour distortion" effect, but we don't want that now (check Figure 6 to see the effect). An EOR would only flip the colours around in strange ways (Figure 7), and an AND instruction clears data (Figure 8).

Figure 6 – Using an OR instead.

Figure 7 – An EOR is also no good...

Figure 8 – ...as it isn't an AND either.

But wait, if we clear out the sprite data, with an AND, leaving the background intact, and then OR in the sprite, it would all work. The sprite mask has the same look as the sprite, but is only two colours. Colour 15 where the background is, making sure all bits there are set, and colour 0 where the real sprite form is, making sure all bits are cleared. Have a look at SPRITE.PI1 (Figure 4, on page 67), and you will see clearly (well, ok, in the picture, the mask is colour 15 and the background is colour 0, but it will get inverted later, read on ...).

Sprite mask		
%11111111	%11010100	first word
%11111111	%11010100	second word
%11111111	%11010100	third word
%11111111	%11010100	fourth word
Pixel colours		
$FFFFFFFF	$FF0F0F00	

All pixels that were colour 0 (background) in the sprite, are now colour 15 (F), and all pixels that had one colour or another in the sprite are now colour 0. By ANDing the sprite mask with the background, we will make sure to clear out all sprite pixels (since they get ANDed with 0) and keeping the status of all other bits (since they are ANDed with 1). It is imperative that you understand this step, if you don't, reread the Boolean algebra part in Chapter 9 (on page 49) check some external sources and think again, or send me an e-mail :). After applying the mask, the screen memory will look like this:

Screen memory		
%00000000	%00000100	first word
%00000000	%00000000	second word
%00000000	%00000000	third word
%00000000	%01010100	fourth word
Pixel colours		
$00000000	$08080900	
#bbbbbbbb	#bbsbsbss	b=background, s=sprite

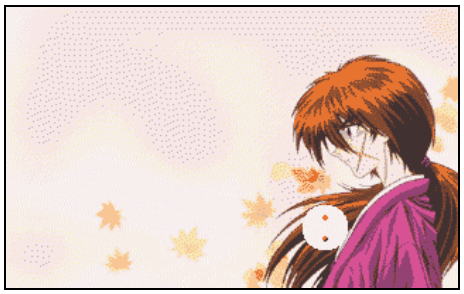


Figure 9 – What the screen looks like, after the mask is applied.

As you can see (also in Figure 9), the background has been preserved, while everything concerning the sprite is wiped out. Now is the time to OR in the sprite data: this instruction will in no way affect the background (since the background colour in the sprite is 0). This is what it will look like after the OR operation:

Screen memory		
%00000000	%00000101	first word
%00000000	%00100000	second word
%00000000	%00000000	third word
%00000000	%01011110	fourth word
Pixel colours		
$00000000	$08288981	
#bbbbbbbb	#bbsbsbss	b=background, s = sprite

To summarise: first we take an inverted version of our sprite with only two colours, and AND that with the background. This clears all pixels that are concerned with the sprite and leaves the background intact. After the mask is applied, it is safe to OR in the sprite data, since after the previous clearing of the sprite pixels, there is no risk of mixing the sprite with the background. The background in the sprite is colour 0, thus the OR instruction will have no effect on the background, the background part in the mask is colour 15 (all 1's) and thus the AND instruction will not affect the background.

OK, now we know how to put the sprite on screen, but we are still faced with the problem of not being able to put it anywhere. To solve this, the sprite and mask data must be shifted. Like with the putpixel, in order to put the sprite at say 0,2, we need to shift the sprite data right two bits. With the putpixel, we shifted "real time", but there is much more data involved in a sprite, so we'll be pre-shifting the sprite instead. When using the pre-shifted method, we assign a storage area that is 16 times larger than the sprite data, and store the sprite in that area shifted in all possible sixteen combinations we need.

I see a big ? in your face right now. Think about it, in the putpixel routine, we could end up shifting the pixel 15 bits to the right, at most, so what we do here with the sprite is to store all those possibilities after one another. When the time comes to put the sprite out, instead of shifting the original sprite data, all we have to do is access the storage area with the correct offset. An offset is the value added to the starting address of something. For example, the middle of the screen is the screen address with an offset of 100*160+80=16080 bytes.

Sprite data		
$00001111		...
Sprite storage area		
$00001111	...	first position, offset 0
$00000111	...	second position, offset 1
$00000011	...	third position, offset 2
$00000001	...	fourth position, offset 3
(the offset number is completely fictional, it's not even an even number)		

Let's say we want to put the sprite at 0,2, we know what that means, it means point to the start of the screen memory, shift the sprite data right by 2, and put it in place. Say a0 points to the screen memory, and a1 to the sprite storage area, then we just need to add offset 2 to a1, and a1 will point to correctly shifted sprite data. Pre-shifting is way faster than loading up the sprite data in a1, and then shifting it, especially since the sprite data consists of several words that all need to be shifted. The downside of course is loss of memory.

Now, there is a problem here, if we have a 32*32 pixel sprite, like in the sample program, the data for the sprite is 16*32 bytes, arranged like this:

Sprite data (W = word)		
First 16 pixels	Last 16 pixels	
WWWW	WWWW	first line
WWWW	WWWW	second line
WWWW	WWWW	third line
... and so on for a total of 32 lines		

When we begin to shift, we want the last bit that go out the first word, to be shifted in as the first bit in the fifth word. This is comparable to the tutorials on scrolling. The last bit that goes out the fifth word, should not go into the first bit of the ninth word, because then a pixel from the first line would go into the second line, but there is no room to shift it out right on the first line. So, we have to add a buffer to every line so that no data will be lost in the shift. In the last shift, the first four words will be all but empty, and the buffer will be all but full. So the sprite storage area will have to look like this.

Sprite storage area (B=buffer, word size)			
First 16 pixels	Middle 16 pixels	Last 16 pixels	
WWWW	WWWW	BBBB	first line
WWWW	WWWW	BBBB	second line
WWWW	WWWW	BBBB	third line
... and so on for a total of 32 lines and 16 such blocks to cover all possible shifts			

Even though the sprite storage area covers a total of 48 pixels, 16 of these will be 0, thus not affecting the background. See the sprite as a 48 pixel wide block, with only 32 pixels coloured. Within this 48 pixel block, the 32 colour pixels will be shifted more and more to the right as X coordinates increase, then when it becomes critical, the block will move 16 pixels to the right in one sweep, and the 32 pixel colour area will be reset, starting the procedure all over again. Run the TUT11BLK.PRG, to see this clear.

Alright, theory part on pre-shifting done, now we need it in direct coding practice as well. First off, we'll need a good instruction with which to shift. Sure, LSR seems a good choice, but we need to be able to preserve the bit that gets shifted out, and LSR doesn't preserve anything. The instruction ROXR, for ROtate eXtended Right, is good in this case. The extended bit is rotated in from the left, and the bit rotated out the right is saved in the carry and extended flag. So, by ROXRing with one each time, we will save what we shift out, and shift it in the next time around (btw, when speaking about the user bits in the status register, flags and bits are used synonymous). Looki looki:

	d0	X (extended bit)
	%00001101	0
`roxr #1,d0`	%00000110	1
`roxr #1,d0`	%10000011	0
`roxr #1,d0`	%01000001	1

What we do is to first copy the sprite data to the sprite storage area, then we take the data from the storage area, rotate extended right with one, and save that data into the next position of the storage area. What we have to think about when coding this is that the data from the first word, goes into the fifth word and so on. In code, it looks like this

```
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
```

First, point to the sprite data, jump over the palette and load up the sprite storage area, which is a `DS.L 3072`: 16 bytes per line, plus 8 for the buffer, totalling 24 bytes per scan line. The sprite is 32 lines and there should be 16 such blocks. This adds up to 24*32*16 = 12288 bytes, which is 3072 long words. In the loop, just copy data from the sprite picture to the storage area, the buffer word area is skipped since it contains nothing at this time. Now comes the challenging part, writing the generic 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
```

First off, load up the storage area in a0 and a1, and make a1 point to the next storage area. This one is empty and should contain the sprite data shifted one bit to the right. Since we have already filled the first position in the storage area, 15 positions are left. 32 lines to each sprite and 4 bit planes to each line. Since all these are treated the same way, we only need one big loop so to speak.

Now comes the fun part, put the first word in d0, this word comes from the previous storage position. Rotate it, and put it in at the next storage position. Now the extended flag holds the bit that was shifted out the right, and this one needs to be shifted in on the left in the first word in the next word cluster. So, a byte offset of 8 (4 words) is added when fetching and storing the next word. The buffer must also come into play, so the last word will get a byte offset of 16. Now, we have pre-shifted three words.

By adding 2 to both a1 and a0 we will be at the next bit plane. It will also clear the extended flag, which is good because otherwise a bit from the last word might come over to the first word on the next bit plane, which is undesirable. Repeat for all four bit planes. We have now moved a line of the sprite. After the four bit planes have been rotated, a0 and a1 will point to the first word in the second 16 pixel cluster, or 8 bytes from the beginning of the data. By adding 16, we will point to the next scan line (16+8 = 24). Repeat for 15 positions. Pretty compact explanation, yes? A graphical representation follows.

Storage area with data, beginning at $0			
16	16	16 pixels	
W W W W	W W W W	W W W W	
... for 32 lines			
0 2 4 6	8 10 12 14	16 18 20 22	byte offset
Storage area without data, beginning at $768			
16	16	16 pixels	
0 0 0 0	0 0 0 0	0 0 0 0	(each 0 is word size)
... for 32 lines			
0 2 4 6	8 10 12 14	16 18 20 22	byte offset

```
* a0 = $0
* a1 = $768

move.w   (a0),d0
roxr     d0                C = leftmost bit from W offset 0
move.w   d0,(a1)           put rotation in 0 at offset 0

move.w   8(a0),d0          as you see, first word of second
                           cluster
roxr     d0                bit preserved and shifted from offset
                           0
move.w   d0,8(a1)          put it at offset 8

move.w   16(a0),d0         first word last cluster
roxr     d0                rotate, carry bit may now be set
move.w   d0,16(a1)         at offset 16

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

Finally, a0 and a1 will both be at offset 8, the first word of the first bit plane, by adding 16 to this, the offset will be 24, the value for a whole line, effectively putting us at the beginning of the next line. That concludes the pre-shift of the sprite.

The mask data has to be pre-shifted a bit differently. Where the sprite colour is, we need the mask to be 0, and where the background is, the mask must be 1, as explained above. A look at the sprite picture will show that the sprite colour area is colour 15, all 1's, and the background is colour 0, all 0's. For the mask to be correctly pre-shifted, we need to invert it, making the background all 1's and the sprite colour area all 0's. When shifting, we must also always be shifting in 1's, not 0's as the case was with the sprite data.

The instruction NOT, for NOT :), will take any value and invert it, this means changing all 1's to 0's and all 0's to 1's. In order to have all bits except those concerning the sprite colour area set, we must make sure to put 1's in the buffer area. Also, at the beginning of each plane loop, we must also make sure that the highest bit of d0 is set, so that 1's are shifted in. Other than that, the sprite and mask pre-shift share ideas. The mask area is as big as the sprite area. Even though this isn't necessary since all bit planes in the sprite look alike, we could have reduced the size by ¾, but for ease of understanding, this was not done.

```
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
```

Unlike the sprite pre-shift where we could set up the storage area with direct memory moves from the sprite picture to the storage area, here we move data, and then perform the NOT instruction to invert the data. Also, instead of just skipping the buffer area like in the sprite, here we fill it with 1's. The bit plane loop is almost identical, with the one exception that the first shift must be guaranteed to shift in a 1, not a 0. A simple OR instruction will make sure the most significant bit is set. That was that, all pre-shifting done.

The method which we use to get the coordinates is the exact one found in Chapter 10. So when we send in our coordinates, we will be provided with a pointer to the screen address, and the number of shifts to be done in d0. The number in d0 is an offset for the sprite data and mask data. By putting the address to the sprite data in an address register, multiplying

d0 with 768 and adding that to the address register, we will get a pointer to correctly shifted sprite data. The reason for the number being 768 is that it is the size of a sprite block.

OK, now comes the problem of actually moving the sprite. We can put a sprite at any coordinate we want, but we can't move it yet. A simple bounce routine here, the sprite will move with a certain X speed and a certain Y speed, and change direction when it hits "walls" (edges of the screen). What we need is a heading, and a speed. For simplicity, we express the heading as either 1 or 0 for both X and Y respectively. 1 is towards bottom right and 0 is towards upper left. X heading is either right or left, and Y heading either up or down. The X and Y speed is how many pixels to move the sprite in desired direction each VBL. So with an X heading of 1, and an X speed of 2, the sprite would move 2 pixels right each VBL.

What the move routine needs to do is to add X and Y coordinates in accordance with heading and speed, as well as checking for wall hits. When a wall hit occurs, the sprite must change direction. A change in direction simply means flipping between 1 or 0 in heading. This might be a good time to tell about the EQU, for EQUals method. Any label can have an EQU applied to it, meaning that whenever one uses the label, it is replaced by the EQU. Easy huh?

```
number              equ       2
                    move.l    #number,d0            same as move.l #2,d0
```

One can say that EQU's, are constants. It's good practice to have as many EQU's as possible, because if you realize you have to change a constant, you only need to change it in one place instead of every place the constant appears. X speed and Y speed is a good example (unless you want variable speed), X coordinate is a terrible thing, since it needs to change all the time. Actually, I think it's best to express the move routine in pseudo code first.

```
If (x_coord > 319 – 32 – x_speed + 1) Then
         x_heading = 0
If (x_coord < 0) Then
         x_heading = 1
If (y_coord > 199 – 32 – y_speed + 1) Then
         y_heading = 0
If (y_coord < 0) Then
         y_heading = 1
```

First we check to see if the heading needs change, as long as the sprite is in any way outside the screen coordinates, we need to change the heading. Since we check the heading before we move the sprite, and move the sprite before drawing it, the sprite will never be drawn off screen. The only trouble here is where all numbers come from. Think of it first without x_speed added, every VBL the sprite just moves one pixel. Then the formula is x_coord > 319 – 32. This is easy to grasp, the X coordinate must not be more than the screen can hold, which is 319, minus the width of the sprite itself of course, which is 32.

The so called "hot spot" of the sprite is the upper left corner. This is the point against which all sprite coordinates are measured. We say that the sprite is at coordinates 13,13, but this really means that the sprite hot spot is at 13,13. Exactly what pixels the sprite inhabits is unknown to us, since the sprite can have any form, but for simplicity, we think of the sprite as a square, with the coordinates in the upper left corner. Thus, when seeing if the sprite hits the right wall, we take the coordinates of the upper left corner, the hot spot, and add the width of the sprite.

The x_speed is also to be taken into account. Imagine the sprite moving with 100 pixels per VBL, then the sprite will be way outside the screen if it's anywhere over the right half of the screen, so the sprite is only ok if it's on the left half of the screen, obviously, the speed must be taken into account. Think of the speed as just enlargement to the sprite. The Y check works exactly the same way, but with a different max coordinate for obvious reasons. It looks a bit different in assembly though.

```
                    cmp       #319-32-x_speed+1,x_coord
```

```
                    blt     x_right_ok          see if x is < 319-32 for width
                    move.w  #0,x_heading        if x >=319, change heading
x_right_ok

                    cmp     #0,x_coord
                    bgt     x_left_ok           see if x is > 0
                    move.w  #1,x_heading        if x <=0, change heading
x_left_ok

                    cmp     #199-32-y_speed+1,y_coord
                    blt     y_low_ok            see if y is < 199-32 for lines
                    move.w  #0,y_heading        if y >=199, change heading
y_low_ok

                    cmp     #0,y_coord
                    bgt     y_high_ok           see if y is > 0
                    move.w  #1,y_heading        if y <=0, change heading
y_high_ok
```

We check if the X coordinate is lesser than the number, and if it is, it's ok and a little branch will skip the changing of the X heading. Whereas the pseudo code's IF statements took place if the check was true, our checks affect if the statements are false. This may look messy, but it's really quite simple, just take a second look at it. We also need to update the coordinates, here's some more pseudo code.

```
If (x_heading = 0) Then
        x_coordinate = x_coordinate – x_speed
Else
        x_coordinate = x_coordinate + x_speed
If (y_heading = 0) Then
        y_coordinate = y_coordinate – y_speed
Else
        y_coordinate = y_coordinate + y_speed
```

No problem there, just change the coordinates according to speed and heading. In assembly it becomes more troublesome though.

```
                    cmp     #0,x_heading        check x heading
                    bne     x_move_right        if 1, move right, otherwise left
                    sub.w   #x_speed,x_coord    move sprite left
                    bra     x_move_done         done moving sprite in x
x_move_right
                    add.w   #x_speed,x_coord    move sprite right
x_move_done

                    cmp     #0,y_heading        check y heading
                    bne     y_move_down         if 1, move down, otherwise up
                    sub.w   #y_speed,y_coord    move sprite up
                    bra     y_move_done         done moving sprite in y
y_move_down
                    add.w   #y_speed,y_coord    move sprite down
y_move_done
```

First, a check to see if X heading is 0, if it is, move to the left, otherwise move to the right. If we move to the left, we subtract the X coordinate by the X speed, and we must also make sure to jump past the move to the right. The Y part is exactly as the X part. Again, just look one more time at the code and if it seems confusing, write it down on paper if you must and go through the different possible branches, it's not too complex once you structuralize it.

Hoah, this takes time to explain, hope you're still with me 'cus we are almost done. Now we know how to pre-shift, apply the sprite and move it. One would think that we have all we need, there is just one more thing to take into account. If we would apply the things we know and fire away, we would have a sprite that moves over the screen and leaves a trail.

The damn thing will never go away, making it most ugly. Why? Because the background must be restored when the sprite has passed it.

So, on every VBL, the background must first be restored, then it must be saved after the sprite coordinates are updated, since the save and restore routine is dependent on the sprite coordinates. Then we can paint the sprite. The save routine just copies a sprite sized block from the screen memory into a save buffer, and the restore routine copies the data from the buffer onto the screen.

What we have now is a main routine that restores background, moves the sprite (rather updates the sprite coordinates), saves the background, applies the mask and lastly paints the sprite. All of this is so fast, that we don't even have to bother with double buffering, so we pull a fast one and just skip that. Here comes the entire source code, don't panic, most of the stuff will be familiar.

```
x_speed             equ      2                 how many x coord to move each VBL
y_speed             equ      1                 how many y coord to move each VBL

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

                    dbf      d2,line
```

```
                    dbf      d1,positions
* pre-shift of sprite done, all 16 sprite positions 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,a0              next bit plane, clears X flag (bad)
                    add.l    #2,a1              next bit plane
                    dbf      d3,plane_mask

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

                    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

                    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
* moves the sprite one pixel in x and y
* see if any headings need to be changed
                    cmp      #319-32-x_speed+1,x_coord
                    blt      x_right_ok        see if x is < 319-32 for width
                    move.w   #0,x_heading      if x >=319, change heading
x_right_ok

                    cmp      #0,x_coord
                    bgt      x_left_ok         see if x is > 0
                    move.w   #1,x_heading      if x <=0, change heading
x_left_ok

                    cmp      #199-32-y_speed+1,y_coord
                    blt      y_low_ok          see if y is < 199-32 for lines
                    move.w   #0,y_heading      if y >=199, change heading
y_low_ok

                    cmp      #0,y_coord
                    bgt      y_high_ok         see if y is > 0
                    move.w   #1,y_heading      if y <=0, change heading
y_high_ok
* all eventual heading changes now made

* move sprite coordinates (change coordinates)
                    cmp      #0,x_heading      check x heading
                    bne      x_move_right      if 1, move right, otherwise left
                    sub.w    #x_speed,x_coord  move sprite left
                    bra      x_move_done       done moving sprite in x
x_move_right
                    add.w    #x_speed,x_coord  move sprite right
x_move_done

                    cmp      #0,y_heading      check y heading
                    bne      y_move_down       if 1, move down, otherwise up
                    sub.w    #y_speed,y_coord  move sprite up
```

```
                    bra      y_move_done        done moving sprite in y
y_move_down
                    add.w    #y_speed,y_coord   move sprite down
y_move_done
* finnished moving sprite

                    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 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     0
y_coord             dc.w     0
x_heading           dc.w     1
y_heading           dc.w     1

spr_dat             incbin   SPRITE.PI1
bg                  incbin   AUTUMN.PI1
old_70              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
```

The longest source to date, I'm truly starting to doubt the wisdom of putting the source code here as well as in a separate file. Anyways, starting from beginning going down, this is what it's all about. The first two lines are the X and Y speed, you may play around with these values to your hearts content, of course, setting them both to the same value will make the sprite move in 45 degrees, while any other values will make the sprite move differently.

Then, the pre-shifting of the sprite and the mask, this has been dealt with extensively and there is nothing more to add. After this, the background is also prepared, it's just another put-degas-file-in-screen-memory. Note here, that there is a background save, this is to make sure something is in the save buffer before starting the main routine, otherwise the main routine would start off by "restoring" a blank area, effectively deleting a sprite sized block of the screen.

All preparations are done, just install the main routine, as described in Chapter 9. Put our main routine in the $70 vector, to have it executed every VBL. Wait for a key press, during

which the main routine will execute continuously, and make a clean exit. Now, take note of how nice and tidy the main routine is, it just consists of subroutine calls, making the structure of the program very easy to read, and isolating each major part of the program for ease of reading.

Each subroutine in turn relies upon the get_coordinates routine, which translates the x_coord and y_coord into data intelligible to the program. As you can see in the comments at the start of the get_coordinates subroutine, what the routine does is to put a pointer to the screen memory in a0 and the sprite position (offset) in d0.b (meaning the least significant 8 bits of the d0 register). Since each subroutine relies upon the get_coordinates routine, if a bug is detected in the coordinate routine, it will only have to be dealt with in one place.

The save/restore background routines are short and simple and do little. They begin by calling the get_coordinates routine in order to get a screen pointer, the sprite position is uninteresting since they both deal with the entire sprite block.

The sprite and mask routines are very similar. Both begin by calling the get_coordinates routine, in order to get a screen pointer and a sprite position. Then either the sprite or mask area is loaded as appropriate, and the sprite position applied as an offset. Then comes a loop of moving data from the background and sprite or mask. Then this data is either ANDed or ORed as appropriate. The result is put back in the screen memory.

Well, that is that. I haven't gone into everything in minute detail, but by now you shouldn't have to be baby nursed through every operation. The source code has many fun things you can do with it yourself, so test around some in the critical areas. The obvious change is the speed change, then, you can try commenting out some things in the main routine, and change an OR to a MOVE in the sprite routine for example. I think it's a good idea to play around some with the source code, and try to predict the changes, in this way, you'll really understand the underlying mechanics.

The next tutorial will probably be a very small one, I'm even considering of calling it tutorial 11 part B, and might cover the well known "infinite trail" of sprites, since it's ridiculously easy. Somewhere soon I suppose I'll do one on joystick and perhaps also mouse operation. I won't promise anything though. Big thanks again go out to Bruno Padinha, for providing valuable feedback and hitting me on the head. Damn, now I have to think of a good quote as well, this part is the hardest. :)

12 Of Controlling The Puppets

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

- Apocalypse Now

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. Appendix H (on page 158) 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 bytes 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

                    dc.l      ikbd_vec           old IKBD vector storage
                    dc.l      old_joy            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 on Appendix H – Intelligent Keyboard (IKBD) Protocol – on page 158, 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

F 0 0 0 R L D U
7 6 5 4 3 2 1 0
(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
ikbd_vec            ds.l      1                  IKBD vector storage
old_joy             ds.l      1                  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.

13 Of Hearing That Which Is Spoken

"I can kill with a word."

- Dune, the Movie

No demo or game is complete without music. The nice blip-blop tunes known as chip music is one of the sweetest forms of music that's ever reached my ears. Seems some people don't quite fancy the type of music the ST has to offer, but I think it's divine. I'm no musician, that's probably why a tutorial on sound has been somewhat delayed, but here it finally is. It's a small tutorial to get down the basics of the sound chip, I intend to follow up with another tutorial on playing the .ym file format, created by Arnaud Carré for the ST-sound project. He's got a homepage over at http://leonard.oxg.free.fr.

The Atari ST comes equipped with a so called PSG: Programmable Sound Generator. This is yet another chip in the ST, the Yamaha YM-2149, fondly called "yammy". According to the ST Internals, this cool chip sports lots of features, for example three independently programmable sound generators, 15 logarithmically volume levels and 16 registers. Registers, yes, just give me the tech specs, register addresses and I'll start to outdo Mozart!

Things aren't that easy though, it would take insanity to hard code the sound chip. By hard coding I mean just entering numbers into the registers, rather than using some program to make music. There is also a little something here, again according to the ST Internals, it's not possible to directly address the yammy registers. Instead, you have to put the desired register number in $ff8800, and then you can put data in $ff8802, or read data from $ff8800. Don't worry, soon comes an explanation of how that applies to real life, but just to be complete, here's a listing of the registers.

Register	Effect
0,1	Period, length and pitch of channel A
2,3	Period, length and pitch of channel B
4,5	Period, length and pitch of channel C
6	Noise generator
7	Bit 0: channel A tone on/off (0 = on, 1 = off) Bit 1: channel B tone on/off (0 = on, 1 = off) Bit 2: channel C tone on/off (0 = on, 1 = off) Bit 3: channel A noise on/off (0 = on, 1 = off) Bit 4: channel B noise on/off (0 = on, 1 = off) Bit 5: channel C noise on/off (0 = on, 1 = off) Bit 6: port A input/output Bit 7: port B input/output
8	Bit 0-3 channel A volume, if bit 4 set, envelope and bit 0-3 ignored
9	Bit 0-3 channel B volume, if bit 4 set, envelope and bit 0-3 ignored
10	Bit 0-3 channel C volume, if bit 4 set, envelope and bit 0-3 ignored
11,12	Sustain, 11 low byte and 12 high byte
13	Waveform envelope
14,15	Port A and Port B, used for output

Like I said, I'm not a musician and I don't understand too much of this. For me, there are only 4 interesting registers: 7-10. Why? Because with register 7, I can turn on and off

different channels, and with 8-10 I can determine the current volume and also set the volume. In other words, registers 8-10 can be used to fade music in and out, as well as create cool bars that go up and down to the beat of the music (or you can have three pulsating sprites or whatever). Here's some example code on how to use the yammy.

```
                    move.b    #7,$ff8800         access Yamaha register 7
                    move.b    #%101,$ff8802      turn off channel A and C
```

As the comments say, this will turn off all (tone) sound from channels A and C. Note how only a byte is moved in both instances, and note also that setting a bit means turning the channel off. Here's how to read the volume of channel A:

```
                    move.b    #8,$ff8800         channel A volume
                    move.b    $ff8800,d0         channel A volume in d0
```

Yep, when you want to read data, you put the register number in $ff8800 as usual, but then you move the data from $ff8800. This obviously means that $ff8800 gets updated between the two move instructions in some way. That's all there is to it actually, but in practice it becomes a little harder.

Sure, we have a basic working of the yammy, well, actually we don't but we know how to use it anyway. Time to play some music perhaps. Most music plays by hooking it up to the VBL, and then just jump to some address in the music file. This of course means that the music file has it's own code routines to play the music, and does not only contain raw music data. The XLR8 chip composer, which can be found at http://dhs.nu, comes with both some example files and the source code for playing them. A good place to start. Here is the code the XLR8 chip composer suggests for playing the music:

```
                    pea       0.w
                    move.w    #32,-(sp)
                    trap      #1
                    addq.l    #6,sp

                    moveq     #1,d0              ; normal song-play mode
                    bsr       music

                    move.l    #music+2,$4d2      ; music in VBL
                    move.w    #7,-(sp)           ; wait for a key
                    trap      #1
                    addq.l    #2,sp

                    moveq     #0,d0              ; exit music
                    bsr       music

                    clr.l     $4d2               ; clear VBL
                    pea       0.w                ; Back to desktop
                    trap      #1

music               incbin    f:\1.xms           ; music file to include
```

Well well, they don't even go out of supervisor mode, naughty naughty. Fairly straightforward and easy, there is only one thing that would trouble us, the $4d2 address. We're used to have the VBL hooked up to $70. At address $4ce there are eight long words that point to VBL routines. These VBL routines are executed one after another. So by writing to say $4ce and $4ce+4, we can have two different VBL routines that get executed one after the other.

In the source code above, the author chooses to put the VBL routine in the second of these eight VBL routines. Our way of writing to the $70 instead, is a bit rawer. Writing to $70 disables all VBL routines except the $70. This means that we know that our, and only our VBL routine is the one to run. In any way, if we spot a memory address close to $4ce in some future source code, we know that it's a VBL routine. Translated into how we would code it, it looks like this:

```
                    jsr       initialise
```

```
                    moveq    #1,d0              normal song play
                    bsr      music              start music

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

                    moveq    #0,d0
                    bsr      music              stop music

                    clr.l    -(a7)
                    trap     #1                 exit

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

                    bsr      music+2            play music

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

                    include  initlib.s

                    section data
                    music    incbin             1.xms      music file to include
```

Well, what do you know, ours became longer, but it's built for more add-ons, and it also has some backup feature such as getting out of supervisor mode, and it also has a `SECTION DATA`. Doesn't really matter, both ways are equally fast really. Speaking of fast, there's an instruction here that I don't think we've encountered before, the moveq instruction. Moveq stands for MOVEQuick. It works pretty much as a normal move, but it can only move quantities in the range of -128 to +127 (a byte). The data does get sign extended though, meaning it will take up a 32-bit quantity (long word). Thus a `MOVEQ.L #0,D0` clears d0 faster than a `CLR.L D0`. Handy little instruction actually. You will also notice how I put $70 on the stack instead of saving it to a variable.

So we have a way of playing music, at least music composed with the XLR8 chip composer. This is a little thin, so against our better knowledge, we decide it would be fun to do some volume meters as well. In order to do these VU bars, we have to play the music, read registers 8-10 (for volume) and then paint the VU bars. Yes, it's true, the yammy has three sound channels, meaning it can play up to three different sounds at once. It also has some noise generator I think so it's able to play four different sounds at once.

The volume is represented by the four least significant bits, meaning it's a value between 0 and 15 (%1111), would be smart to paint one line of VU bar for each volume, right? So at volume 15, the VU bar takes up 15 lines, this can be a little small though, so just for fun we decide to leave every second line blank (background coloured), thus volume 15 will take up 30 lines instead. Since we have 15 different VU lines, each line can have it's own colour and we'll still have one over for the background as well. Seems the ST was made for these things! Actually, it wasn't, it's just normal for computers to have many things on binary bounders. Thus the powers of two (such as 16, or 0-15) show up a lot.

As we know, the volume data may contain other stuff than just the four volume bits, so in order to keep only those bits, we have to and off the other bits. Otherwise the volume data might contain a number larger than 15 and that will screw us up big time, making us do stupid things like drawing outside of the screen, which will probably result in a crash.

Now, to decide on how to draw the volume bars. What, should this be a problem? Just do a dbf loop according to the volume and paint as many lines as the volume. Yes, that won't

work. Say one VBL the volume is 12, then the other VBL it's 5, but the VU bar will still be 12 lines high since we don't delete it.

So on every VBL, we first delete the VU bar and then paint it. This can be done, but actually it's smarter to first paint the VU bar, then delete it. The delete part is more generic, and thus easier to fit into a loop, while the paint part requires colour updates and so on. Thus, the VU bar routine will be to first paint all three VU bars to the max, then delete as many lines as the inverted volume (volume 15 means delete nothing, volume 0 means delete 15). This will work nicely. Actually, theory part over, time for source code:

```
                    jsr     initialise

                    moveq   #1,d0             normal song play mode
                    bsr     music

                    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  $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 old $70

                    moveq   #0,d0             stop music
                    bsr     music

                    jsr     restore

                    clr.l   -(a7)
                    trap    #1                exit

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

                    bsr     music+2           play music

* put in VU meters
                    move.l  $44e,a0           get screen address
                    add.l   #160*199-(15*2)*160,a0 bottom area of screen
                    move.l  #meter,a1         point to meter colours

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

* delete VU meters 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_meter

                    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 meter
```

```
                    jsr      del_meter

                    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 meter
                    jsr      del_meter

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


del_meter
* 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

meter
* colour data for each line of VU meter
                    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,$093,$09b,$094,$09c,$095,$09d,$096
                    dc.w     $09e,$097,$29f,$39f,$49f,$59f,$69f,$79f

music               incbin   instinct.xms       music file
```

Here we go! Setup the music to work, then put in the palette. Backup the old $70 by putting it on the stack, put in my own $70 routine, wait for keypress, shut down music, restore and exit. All the usual stuff.

The main routine starts off by playing the music, that one simple command is enough to keep the music running. Now comes the interesting part: putting in the VU bars. I get the screen address, and go to the bottom area of the screen. This means go to the absolute bottom, line 199, and then hop 30 lines up. Because I want to paint the entire VU bar, and the VU bar is 15*2 lines high (max 15 volume and every second line is interlaced).

The actual painting of the VU bar is a bit tricky. I point to the bar label, which contains colour data for the VU bars. Each little four word block here is data for a bit plane, so the first line is colour 15. The reason for the two leading 0's, is that I don't want the entire bit plane filled, in this way, only 8 pixels out of sixteen will be set.

```
dc.w    $00ff,$00ff,$00ff,$00ff
```

is the same as

```
dc.w    %0000000011111111
dc.w    %0000000011111111
dc.w    %0000000011111111
dc.w    %0000000011111111
```

And we know that by putting this into the screen memory, we will have 8 pixels with colour 0, and then 8 pixels with colour 15. The next entry is

```
dc.w    $0000,$00ff,$00ff,$00ff
```

which is the same as

```
dc.w    %0000000000000000
dc.w    %0000000011111111
dc.w    %0000000011111111
dc.w    %0000000011111111
```

When we put this into screen memory, we get colour 14 in the last 8 pixels.

When pointing to the bar label, a1 points to memory that contains this data: $00ff00ff00ff00ff. This data I move into d0 and d1 with a movem instruction, then I put that data into the screen memory. Adding 8 to the screen memory pointer will put me on the next VU bar, 16 pixels to the right, and then I move that same colour data into the screen memory there, and repeat one last time. Then I need to correct the screen pointer: by adding 320, I move two lines down, and then I need to subtract 16 from that to be on the first VU bar position. Repeat all this 15 times to paint in all three VU bars full.

Now the time has come to delete the VU bars, so that they will reflect the value of the volume. Again, get screen memory and go to the bottom area, pointing right at the topmost line of the first VU bar. Clear d0 just to be sure there's no garbage, and read Yamaha register 8, which is channel A volume. Now the volume is in d0, and the screen address is in a0, jump into the del_bar routine to delete the bar.

The del_bar routine is also a bit tricky, and uses an almost dirty method. Backup the registers so that they don't get destroyed. This is good practice for sub routines, so that other programmers can count on calling routines without having data destroyed. And away all bits but the first four. Now we have pure volume data in d0.

It would be tempting to just go through a dbf loop to clear out the lines, but this won't work. A bdf loop always executes once, but in some circumstances, we don't want the delete loop to execute even once. So instead of having a loop, I have 15 blocks of delete data, each block deletes one line of VU bar. By jumping into the correct block, I take away the exact number of VU bar lines. Each delete block looks like this:

```
clr.l    (a0)              clear two bit planes
clr.l    4(a0)             clear two bit planes
add.l    #320,a0           hop two lines down
```

This will clear out the four bit planes of a line, and then hop two lines down. This block takes 12 memory positions. Usually, an instruction takes a long word to store, these three instructions are no exceptions. Since all instructions just get loaded into memory, we can easily jump to them. Go into MonST mode to see this clear, in the instruction window, you'll see all instructions, and to the left of them you'll see what memory position they occupy. By adding 4 to the program counter, you usually skip one instruction.

For example, by jumping to the start of the delete blocks+12, we will skip one delete block. In the del_bar routine, I have 15 delete blocks, I let a1 point to the beginning of these blocks. Then I multiply the volume by 12, since this is the size of each delete block, add that value to a1, and jump to the address a1 contains.

Say that we have volume one, this means execute 14 delete blocks, which will leave only one line of VU bar left. 1*12 = 12, thus we will jump to the beginning of the delete blocks+12, which will let us skip one delete block, and then we have 14 left. Here's how it looks:

```
Memory position (fictional)
$0                  move.l   $10,a1            beginning of delete blocks
$4                  mulu     #12,d0            length of one delete block
                    d0 contains 12
$8                  add.l    d0,a1             skip some delete instructions
                    a1 now contains $1c
$c                  jmp      (a1)              jump to $1c

$10                 clr.l    (a0)              clear two bit planes
$14                 clr.l    4(a0)             clear two bit planes
$18                 add.l    #320,a0           hop two lines down

$1c                 clr.l    (a0)              clear two bit planes
$20                 clr.l    4(a0)             clear two bit planes
$24                 add.l    #320,a0           hop two lines down

$28                 clr.l    (a0)              clear two bit planes
$2c                 clr.l    4(a0)             clear two bit planes
$30                 add.l    #320,a0           hop two lines down
                    …        13 more delete blocks
```

That's that. In short, the program only runs a VBL routine. This VBL routine plays the music, and then paints in VU bars at max. Then the volume is read from yammy registers, for each volume read, the del_bar routine is called which deletes as many lines as the inverted volume. Then add 8 to the screen memory to point to the next VU bar, read the volume and call the del_bar routine.

With this knowledge of the volume workings, you can have just about any effect tied to the volume. I first had the background colour be set by the three channels, channel A for red colour, channel B for green and channel C for blue. This created quite the psychedelic background I can tell you :) One cool thing would be to have three Xenon 1 ships, that morph back and forth between tank and ship, say that volume 15 means complete tank morph, and 0 means ship, then volume 7 would be morph in between those. Once again, your fantasy can run free!

In the next tutorial, I hope to cover the .ym file format as described in the beginning. This will mean setting up our own routine to write raw data into the sound chip, which should be quite easy. Just put register number as usual, then write the data found in the .ym file. Stay tuned ...

14 Of Using The Gramophone

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

- Braveheart

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 plug-in 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 an .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), an .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 digi-drums 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 hex-editor. 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 an 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 digi-drums, 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 this yourself if you find songs that should use digi-drums, 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 digi-drums). To do this, we put ourselves at the beginning of the text field, which always start at the same place, and 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, and 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 (digi-drums 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 an .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 key press 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.

15 On Fading To Black

"I wish for this night-time

to last for a lifetime

The darkness around me

Shores of a solar sea

Oh how I wish to go down with the sun

Sleeping

Weeping

With you"

- Nightwish, Sleeping Sun

It has occurred to me that by striving ever forward, we've forgotten to speak about some basic things, so for this tutorial and the next one, we'll be taking a step back and reviewing some things. You may have guessed these techniques yourself, but it never hurts to have it spelled out. Also, I thought I'd share some new thoughts on development, we'll take that first.

Most of the source for the tutorials in the past I've actually written in Devpac on a real Atari, but it has now become clear to me that developing in Windows on an IBM compatible is easier and more efficient. I got the tip over at www.atari-forum.com, a discussion forum for all topics Atari (where I'm one of the moderators for the coding section, yay). Have one "launcher file" with only one line

```
include whateveryoursourcename.s
```

By doing this, you'll assemble any source files you want, and you can edit those source files outside of Devpac, and then assemble them in Devpac. When I wrote this tutorial, I had a file named _WRAP.S that had the line "include tut15.s" in it. Then I used Ultraedit (my editor of choice) to edit TUT15.S, I also had Devpac running under STEem. Whenever I felt like assembling my source, I just saved in Ultraedit, alt-tabbed into STEem and hit alt-a to assemble my source; smooth and easy.

Speaking of Ultraedit, there is a topic going on over at http://www.atari-forum.com/viewtopic.php?t=946 to try and work out good syntax highlighting for Atari assembly in Ultraedit (www.ultraedit.com). Wow, that's a lot of various things you wouldn't have seen pop up in a tutorial from say 1994. Now onto the serious stuff.

The palette is an extremely powerful thing when you want to change colours quick and easy. Unfortunately it has the obvious limitation of not changing the pixels. Using the palette you can black out the screen without erasing the contents (by setting all colours to black), make things pulse (by incrementing and decrementing colour intensity) or wait with displaying a picture. Say you want to calculate a big fractal, just set the palette to all 0, calculate your fractal, then whip in the palette to display the result. The effect will be that no one will see you draw the fractal, only the final result will be shown.

As we've been through before, there are 16 colours in the palette, the first one being the background colour, located at $ff8240. Each colour is a word long, making the palette end at $ff825e. Each word is built up like this

```
00000RRR0GGG0BBB
```

The first three bits control blue intensity, then there's a zero bit, the next three bits control green intensity, a zero and the final (non-zero) three bits control red intensity. The maximum value you can get out of three bits is 8, and since the colour intensities are at 4 bit boundaries, they are very easy to access in hex (since each character in hex mode is a 4 bit quantity). Thus $700 means max intensity of red and zero intensity for green and blue, $444 means medium intensity for all three colours.

When they built the STe, they thought that it would be nice to have more colours in the palette, and indeed, it's easy to just add an additional bit since that would still have the palette on a 4 bit boundary, making each colour range from 0-15. However, there was a problem, they could not add a bit in the beginning and just shift the other bits to the left, since that would mean all old palette values would in effect be shifted left one bit creating an entirely different value than was originally intended.

The solution to this problem is cunning, but unfortunate. They added the least significant bit where the zero bit used to be. This maintains backwards compatibility, and adds 8 new possible colour intensities. So the STe palette looks like this

```
0000rRRRgGGGbBBB
```

This means that $700 is still (almost) maximum intensity of red. What in the memory is perceived as the most significant bit, is in palette terms the least significant bit. This sounds very confusing perhaps, but just picture moving the uppermost bit of each colour intensity first. Let's say then that we want the intensity between $100 and $200, this would be colour $900, since that would be

```
0000rRRRgGGGbBBB
0000100100000000
```

Which we can interpret as

```
0000RRRrGGGgBBBb
0000001100000000
```

Thus, when using the STe palette, we must think about the fact that the most significant bit for each colour, is in actuality the least significant bit. The number order for intensities, from lowest to highest is 0, 8, 1, 9, 2, A, 3, B, 4, C, 5, D, 6, E, 7, F. So if you use colour $fff, the STe will interpret this as intensity 15 for all colours, and the ST will interpret it as colour intensity 7, since the ST doesn't care about whether the fourth bit is set or not.

That should be all there is to the palette, making full utilization of it will be up to each one. In order to do something I thought we'd just do a simple fade in effect. Fading in a picture is so much nicer than just whipping it onto screen. Fading out is also much nicer than just zapping it away, you can also fade to white and make the screen sort of flash away.

What we want is to begin with a black palette and pixel data on the screen, then increment the colour values of the palette until they reach the values intended for the picture. In order to keep things simple, I opted to skip the STe palette since there's lots of shifting involved whenever you want to use it. So the fade will only have a maximum of 7 intensities to work with, making it a pretty bad looking fade effect.

We'll need a copy of the original palette, and a current palette which we increment until it reaches the original. It would be tempting to compare the real palette to the current one and add $111 (one intensity of each colour) if they don't match, but that won't work. Say one colour is supposed to be $100, if we compare our current $000 with that, they don't match, so we add $111 making the current colour $111, which is more than $100. Instead, we must compare each red, green and blue value individually. This can easily be done by just masking off all bits except the three controlling the intensity for either red, green or blue.

```
and.w    #%011100000000,d0  mask off all but red values
and.w    #%011100000000,d1  mask off all but red values

cmp.w    d1,d0              see if red is correct intensity
```

```
                    beq     red_fin           if not ...
                    add.w   #%000100000000,d1 ... add one intensity of red
red_fin
```

Let's assume d0 holds the real colour, and d1 holds the temporary. All bits except the ones controlling red are masked off, then values compared. If they do not match, add one to the value. The value to add will be different depending on which intensity we check for, since different intensities begin at different bit positions. That's pretty much it, here's the entire source

```
                    section text

                    jsr     initialise

                    movem.l picture+2,d0-d7   put picture palette in d0-d7
                    movem.l d0-d7,pal         copy palette to pal

                    movem.l temp_pal,d0-d7    put current palette in d0-d7
                    movem.l d0-d7,$ff8240     apply current palette (all 0)

                    move.w  #2,-(a7)          get physbase
                    trap    #14
                    addq.l  #2,a7

                    move.l  d0,a0             a0 points to screen memory
                    move.l  #picture+34,a1    a1 points to picture

                    move.l  #7999,d0          8000 longwords to a screen
loop
                    move.l  (a1)+,(a0)+       move one longword to screen
                    dbf     d0,loop

                    move.l  $70,old_70        backup $70
                    move.l  #main,$70         start main routine

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

                    move.l  old_70,$70        restore $70

                    jsr     restore

                    clr.l   -(a7)
                    trap    #1

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

                    add.l   #1,counter        increment counter variable
                    cmp.l   #15,counter       only execute main sometimes
                    bne     do_nothing        skip instructions
                    clr.l   counter           reset counter

                    move.l  #pal,a0           a0 points to values to reach
                    move.l  #temp_pal,a1      a1 points to current values

                    rept    16                do for each color
                    jsr     check_red         see if red intensity should increase
                    jsr     check_green       see if green intensity should
                                              increase
                    jsr     check_blue        see if blue intensity should increase
                    add.l   #2,a0             point to next color
                    add.l   #2,a1             point to next color
                    endr

                    movem.l temp_pal,d0-d7    put current palette in d0-d7
```

```
                    movem.l  d0-d7,$ff8240      apply current palette

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


check_red
                    move.w   (a0),d0            move one final color into d0
                    move.w   (a1),d1            move one temp color into d1

                    and.w    #%011100000000,d0  mask off all but red values
                    and.w    #%011100000000,d1  mask off all but red values

                    cmp.w    d1,d0              see if red is correct intensity
                    beq      red_fin            if not ...
                    add.w    #%000100000000,(a1)    ... add one intensity of red
red_fin
                    rts


check_green
                    move.w   (a0),d0            move one final color into d0
                    move.w   (a1),d1            move one temp color into d1

                    and.w    #%000001110000,d0  mask off all but green values
                    and.w    #%000001110000,d1  mask off all but green values

                    cmp.w    d1,d0              see if green at correct intensity
                    beq      green_fin          if not ...
                    add.w    #%000000010000,(a1)    ... add one intensity of green
green_fin
                    rts


check_blue
                    move.w   (a0),d0            move one final color into d0
                    move.w   (a1),d1            move one temp color into d1

                    and.w    #%000000000111,d0  mask off all but blue values
                    and.w    #%000000000111,d1  mask off all but blue values

                    cmp.w    d1,d0              see if blue at correct intensity
                    beq      blue_fin           if not ...
                    add.w    #%000000000001,(a1)    ... add one intensity of blue
blue_fin
                    rts



                    include  initlib.s

                    section data
old_70              dc.l     0
picture             incbin   sleepsun.pi1
counter             dc.l     0

                    section bss
pal                 ds.w     16
temp_pal            ds.w     16
```

First I save the palette of the picture in a storage space, then I put the temporary palette in, since the temporary palette is initialized to all 0's, this has the effect of blacking out the screen. Next I load up the picture as described in tutorial 6 and set up the main routine.

The counter code is for delay purposes; otherwise the fade effect would hardly be visible. I make a0 point to the palette to reach, and point a1 to the temporary one. Then I check the

individual intensities, and add 2 to each pointer in order to point to the next colour, repeating this for the number of colours in the palette, namely 16.

You will notice that the check sub-routines are a bit different than the one described above, I add to the value pointed to by a1, which is the current palette. It may be considered slightly bad program habit to just assume that a1 points to the current palette like that, but coding demos and assembly in general depends on tight kept code that knows what it's doing. Besides, the tutorials aren't really for teaching you how to make good code; they are intended as basic introductions to various coding techniques.

That's that, one easy effect achieved by manipulating the palette. If you want to fade to white, just set the temporary palette to the real palette, and increment until you reach $777. If you want to experiment, I suggest trying to implement the effect with a STe palette instead, the included picture has a STe palette so it's ready to go. This should involve shifting the fourth bit of each colour intensity down as the first when adding to the colour intensity, and then shift it back. For the next tutorial, I think we'll handle full screen scrolling, without moving any picture data!

Appendix A MC68000 Instruction Set

A.1 ABCD - Add Binary Coded Decimal

This instruction is a specialized arithmetic instruction that adds together two bytes (and only bytes) containing binary-coded decimal numbers.

The addition can either be done between two data registers or between two memory locations. If performed on bytes in memory, only address register indirect with pre-decrement addressing can be used. This facilitates easy manipulation of multiple-precision BCD numbers. The extend bit is added along with the BCD bytes to allow this multi-precision data manipulation. Also note that the Zero flag is only changed if the result becomes non-zero.

Therefore, both the Extend and Zero bits in the condition code register should be preset before the operation is performed. The Extend bit would normally be preset to a zero (to prevent extension on the first addition), and the Zero bit to a one (to preset a zero result prior to the first addition). A MOVE #4,CCR would setup these flags correctly.

Syntax: ABCD Dn, Dn

or ABCD -(An), -(An)

Flags affected: The Extend, Zero, and Carry flags are affected as per the result of the operation. The state of the Negative and Overflow flags is undefined.

A.2 ADD - Add Binary

The ADD instruction adds the source to the destination operand with the result appearing in the destination. It is possible to add bytes, words, or long words with this opcode. Either the source or destination (or both) must be a data register. The source operand can be any memory location or data register, and the destination operand can also be any memory location or data register.

Syntax: ADD Dn, Dn

or ADD address, Dn

or ADD Dn, address

Flags affected: The Extend, Negative, Zero, Overflow, and Carry flags are all affected as per the result of the addition.

A.3 ADDA - Add Address

This variant of the ADD instruction only differs from ADD in that an address register is specified as the destination. As an address rather than data is being manipulated, the condition code flags are left unaltered. Only sign-extended words or long words can be added.

A.4 ADDI - Add Immediate

This variant of the ADD instruction is used to add a constant value to the destination. The immediate operand can be any 8-, 16-, or 32-bit value as specified by the .B, .W, or .L opcode suffix. The destination can not be an address register or a program counter relative address.

Syntax: ADDI #imm, Dn

or ADDI #imm, address (where address is any memory addressing mode except program counter relative)

Flags affected: The Extend, Negative, Zero, Overflow, and Carry flags are all set as per the result of the addition.

A.5 ADDQ - Add Quick

This variant of the ADD instruction is used to add a small positive integer between one and eight to the destination. The destination can be a memory location, a data register, or an address register. If it is an address register, the condition code flags are unaffected and the operand length can not be a byte. This operation takes the place of the increment instruction found on other processors.

Syntax: ADDQ #imm, Rn

or ADDQ #imm, address

Flags affected: The Extend, Negative, Zero, Overflow, and Carry flags are all set as per the result of the addition unless the destination is an address register.

A.6 ADDX - Add Extended

This variant of the ADD instruction adds two numbers plus the Extend bit from the condition code register. This allows multiple-precision additions to be performed. For this reason, the Zero flag is only affected when a non-zero result is obtained. This means that if multiple numbers are added together using ADDX, the Zero flag will stay reset if any of those numbers were non-zero.

Syntax: ADDX Dn, Dn or ADDX -(An), -(An)

A.7 AND - Logical AND

This instruction logically ANDs bits in the source operand with the same number of bits in the destination operand were the result is left. The number of bits can be 8, 16, or 32 as per the .B, .W, or .L opcode suffix. One or both operands must be a data register.

Syntax: AND Dn, Dn

or AND Dn, memory

or AND memory, Dn

A.8 ANDI - Logical AND Immediate

This instruction logically ANDs an immediate byte, word, or long word value with the destination. The destination can be a data register, memory, or one of two special cases: the condition code register, only a byte-length immediate value is allowed. If the destination is the status register, only a word-length immediate value is allowed, and the processor must be in supervisor mode or a priviledge voilation will occur.

Syntax: ANDI #imm, Dn

or ANDI #imm, memory

or ANDI.B #imm 8-bit value, CCR

or ANDI.W #imm 16-bit value, SR (Privileged).

Flags affected: The Overflow and Carry bits are reset, the Negative and Zero bits set as per the result, and the Extend bit is unaffected.

A.9 ASL - Arithmetic Shift Left

This instruction shifts the destination operand left by a specified number of bits. If you are shifting a data register, the number of bits to be shifted can be specified as an immediate

value or as a value in another data register. The immediate value can be 1 to 8, whereas the data register can be 1 to 64 (where zero acts as the 64 count). Data registers may be shifted as 8, 16, or 32 bit quantities. Only 16-bit word values can be shifted in memory and then only by one bit. As shown below zeroes are shifted in at the right hand side of the operand. As each bit is shifted out of the left of an operand, it is placed in the Carry and Extend bits of the condition code register. If the sign of the operand changes during the shift, the Overflow bit is set in the condition code register.

C < < | <<< ASL <<< | < 0 X <

Syntax: ASL Dn, Dn

or ASL #imm 3-bit value, Dn

or ASL memory (1 bit shift only)

A.10 ASR - Arithmetic Shift Right

This instruction shifts the destination operand right by a specified number of bits. If you are shifting a data register, the number of bits to be shifted can be specified as an immediate value or as a value in another data register. The immediate value can specify a shift of 1 to 8, while the data register can specify a shift of 1 to 64 (where zero acts as the 64 count). Data registers may be shifted as 8, 16, or 32 bit quantities. Only 16-bit word values can be shifted in memory and then only by one bit. Each bit shifted out of the right hand side of an operand is placed in the Carry and Extend bits of the condition code register. As shown below the bit shifted in at the left hand side is the current sign bit (the most significant bit is therefore preserved throughout the shift).

> C Current MSB > | >>> ASR >>> | > > X

Syntax: ASR Dn, Dn

or ASR #imm 3-bit value, Dn

or ASR memory (1 bit shift only)

A.11 BRS - Branch Always

This instruction changes the program counter register so execution continues at a different point in the program code. The destination of the jump is specified as a signed displacement to the program counter. This signed displacement can be an 8- or 16- bit quantity. With a bit 8-bit quantities, this allows branches of +126 to -128 bytes; 16-bit quantities can specify branches of +32766 to -32768 bytes. The value of the program counter when the displacement is added is taken to be the first word after the BRA opcode. This is the actual opcode address plus two. Normally an assembler will assume a 16-bit quantity as the displacement, but if an opcode suffix of .S is appended th the BRA, a short 8-bit displacement will be used instead.

Syntax: BRA label (16-bit displacement)

or BRA.S label (8-bit displacement)

Flags affected: None.

A.12 Bcc - Branch Conditionally

Other variants of the BRA instruction allow a branch to be made only if a certain condition is met in the condition code register. These Bcc instructions can be divided into three different categories. Whether or not this instruction is actually executed depends on the required condition, which is verified by means of the flags. A minus sign before a flag indicates that it must be cleared to satisfy the condition. Logical operations are indicated with "*" for AND and "/" for OR.

A.12.1 Branches depending on flag status

BCC - Branch if carry clear –C

BCS - Branch if carry set C

BNE - Branch if zero clear –Z

BEQ - Branch if zero set Z

BVC - Branch if overflow clear –V

BVS - Branch if overflow set V

BPL - Branch if negative clear –N

BMI - Branch if negative set N

A.12.2 Branches after unsigned comparison

BHI - Branch if higher than -C * -Z

BHS - Branch if higher than or same as

BLO - Branch if lower than

BLS - Branch if lower than or same as C / Z

BEQ - Branch if equal to Z

BNE - Branch if not equal to -Z

A.12.3 Branches after signed comparison

BGT - Branch if greater than N * V * -Z / -N * -V * -Z

BGE - Branch if greater than or equal to N * V / -N * -V

BLT - Branch if less than N * -V / -N * V

BLE - Branch if less than or equal to Z / N * -V / -N * V

BEQ - Branch if equal to Z

BNE - Branch if not equal to -Z

Syntax: Bcc label (16-bit displacement) or Bcc.S label (8-bit displacement).

Flags affected: None.

A.13 BSR - Branch to Subroutine

This instruction causes control to be passed unconditionally to the specified program counter displacement as in the BRA opcode. However, before the branch is made, the address of the opcode following the BSR is saved on the stack so return can later be made to that address to continue processing at that point. This is achieved as follows:

1. The 24-bit address following the opcode is pushed on the stack as two words.
2. The program counter is loaded with its new value and processing continues at the new address.

Syntax: BSR label (16-bit displacement)

or BSR.S label (8-bit displacement).

Flags affected: None.

A.14 BCHG, BCLR, BSET, BTST - Bit Test and Change, Clear, Set

These instructions allow the manipulation and testing of single bits. The bits are numbered from the right to the left starting with bit no zero. Thus a byte contains bits 0 to 7; a word bits 0 to 15; and a long word bits 0 to 31. The number of the bit to be tested is specified in a data register or as an immediate value. The value of the bit is reflected in the Zero flag of the condition code register. This means that the if the bit tested was zero, the Zero flag will be set (Z=1). Therefore the Zero flag is always the opposite of the bit being tested. Once the test is made and the Zero flag is set up, the tested bit is manipulated as follows:

BCHG - The bit is reversed.

BCLR - The bit is cleared to zero.

BSET - The bit is set to a one.

BTST - The bit is unchanged.

Syntax: Bxxx Dn, address

or Bxxx #imm, address.

Flags affected: Zero flag only.

A.15 CHK - Check Against Bounds

This instruction checks its first operand against a data register's word contents. If the data register contains less than zero or greater than its first operand, a trap to the address specified by vector 6 occurs. Thus, CHK can be used to ensure that an element of an array is neither below nor above its boundaries.

Syntax: CHK bounds, Dn (where bounds is anything except an address register)

Flags affected: All flags are undefined after this operation.

A.16 CLR - Clear Destination to Zero

This instruction allows a byte, a word or a long word to be cleared to a zero according to the operand suffix .B, .W, or .L. The destination can be either a data register or memory. Address registers cannot be cleared with the CLR instruction (Use MOVE.L #0, An).

Syntax: CLR Dn

or CLR address.

Flags affected: Negative, Overflow, and Carry are all set to zero, the Zero flag is set to a one, and the Extend flag is unaffected.

A.17 CMP - Compare

This instruction compares two operands and sets flags in the condition code register according to the result. Except for the Extend flag, the flags are set as if the source operand were subtracted from the destination. However, the result of this subtraction is not actually retained so the destination remains unchanged. The information about the comparison that is stored in the condition flags can then be acted upon by a Bcc-instruction. CMP may be used with byte, word, or long word source operands. Note that although any addressing mode can be used to specify the source operand, an address register can only be used if a word or long word comparison is performed.

Syntax: CMP address, Dn.

A.18 CMPA - Compare Address

This variation of the CMP instruction is used to compare a source operand with an address register as destination operand. Only word or long word compares are allowed with CMPA. If a word is used as source, if is sign-extended to 32 bits before the comparison is made.

Syntax: CMPA address, An.

Flags affected: Same as CMP instruction.

A.19 CMPI - Compare Immediate

This variation of the CMP instruction is used to compare a source operand consisting of an immediate value with either a data register or memory. The comparison length can be byte, word, or long word as specified by the .B, .W, or .L opcode suffix.

Syntax: CMPI #imm, Dn

or CMPI #imm, memory.

A.20 CMPM - Compare Memory

This variation of the CMP opcode is used to compare sequential memory locations. These locations can be of type byte, word, or long word as specified by the .B, .W, or .L opcode suffix. To perform the sequencing automatically through memory, both source and destination operands must be specified using address register indirect with postincrement. Thus, after the compare is made, the address registers of both source and destination operands will be incremented by the length of data compared.

Syntax: CMPM (An)+, (An)+.

Flags affected: Same as the CMP opcode.

A.21 DBRA - Decrement and Branch

This instruction is used to control the program counter register in much the same way as BRA instruction is except that this allows greater power and versatility. By using DBRA, a specified data register is decremented and the branch made only if that register goes past zero. Thus, the count from a positive number will count down until zero and branch one more time. This allows loops where an index of zero is the last element. Note that as a result of this, the value left in the register will be -1 when an exit is made at the end of the loop. As an example, if eight locations were to be accessed, the data register specified in the DBRA instruction would be loaded with seven. The countdown, including the final zero, would go through eight cycles. The program counter register is modified as in the BRA instruction whereby a sign-extended 16-bit displacement is added to the program counter. No short 8 bit form is available. only bits 0 to 15 (that is, one word) of the data register is used. The destination of the branch is usually supplied as a label from which the assembler automatically calculates the displacement needed to that label.

Syntax: DBRA Dn, label.

A.22 DBcc - Decrement and Branch Conditionally

This is a whole series of instruction that resemble the conditional versions of the BRA opcode. Conditional decrement and branch instruction work in a similar manner to the DBRA instruction except that one step is added to the execution process. Before the decrement is performed as in DBRA, the condition specified in the mnemonic is tested (in the opposite order to that suggested by the opcode name). If the condition is true, control drops through to the next instruction - the branch is not made. This is the opposite to the normal branch instruction where the conditional branch is made if the condition is true. Thus this mnemonic might more accurately be read as "decrement and branch if the condition is not fulfilled". Powerful loops can be constructed using the decrement and branch conditional instruction; an exit can be made from the loop either if the data register passes zero or if a pretested

condition is met. The following list displays the conditions available for testing before the decrement and possible branch is made. This list is similar to that for the Bcc opcode with the addition of the F (false) and T (true) conditions, which specify an always false or always true precondition. Therefore a DBF is always false, so it will never drop through to the following opcode. Thus, the branch after the decrement will always be performed. Conversely, a DBT is always true, so it will always drop through and never perform the decrement. (This would only be likely to be of use during program development.)

DBEQ - Decrement, branch equal.

DBF - Decrement, branch false. (Same as DBRA.)

DBGE - Decrement, branch greater than or equal.

DBGT - Decrement, branch graeter than.

DBHI - Decrement, branch higher.

DBLE - Decrement, branch less than or equal.

DBLS - Decrement, branch lower than or same.

DBLT - Decrement, branch less than.

DBMI - Decrement, branch minus.

DBNE - Decrement, branch not equal.

DBPL - Decrement, branch plus.

DBRA - Decrement, branch unconditionally.

DBT - Decrement, branch true. (Branch never taken.)

Syntax: DBcc Dn, label.

A.23 DIVS, DIVU - Divide Signed, Unsigned

These instructions allow a 16-bit divisor (n„mnare) to be used as a source and a 32-bit destination to be specified as dividend (t„ljare) in a divide operation. DIVS assumes both numbers are signed, whereas DIVU assumes both to be unsigned. The destination must be a data register. The source can be a memory location or another data register. The result is stored in the low word of the destination data register and the remainder in the high word of the same register. If the result will not fit in the 16 bits of the low half, the Overflow flag is set in the condition code register. It is possible that the overflow condition can occur during the internal processing of the divide, in which case the Negative and Zero flags will be undefined as will be the result. Either a conditional branch on overflow or a TRAPV can be placed after the divide opcode to act upon the error.

Another problem occurs if a divisor of zero is specified. In this case a division-by-zero exception processing sequence is automatically initiated which causes a trap through vector 5.

Syntax: DIVx Dn, Dn

or DIVx address, Dn.

Flags affected: The Carry flag is always set to zero. The Zero, Overflow, and Negative flags are set as per the result. The Extend flag is unaffected.

A.24 EOR - Logical Exclusive OR

This instruction performs a logical exclusive OR of the source operand with the same number of bits in the destination operand where the result is left. The number if bits can be 8, 16, or 32 as specified by the .B, .W, or .L opcode suffix.

Syntax: EOR Dn, Dn or EOR Dn, address or EOR address, Dn.

Flags affected: The Overflow and Carry flags are reset. The Negative and Zero flags are set as per the result, and the Extend flag is unaffected.

A.25 EORI - Logical Exclusive OR Immediate

This instruction performs a logical exclusive OR on a length of byte, word or long word between an immediate value and a destination. The destination can be a data register, memory or one of two special cases: the condition code register or the status register. If the destination is the the condition code register, only a byte-length immediate value is allowed. If the destination is the status register, only a word-length immediate value is allowed, and the processor must be in supervisor mode or else a priviledge voilation will occur causing a trap through vector 8.

Syntax: EORI #imm, Dn

or EORI #imm, memory

or EORI.B #imm 8-bit value, CCR

or EORI.W #imm 16-bit value, SR (Privileged).

Flags affected: Same as the EOR instruction.

A.26 EXG – Exchange Registers

Exchange the contents of two registers. The size of the instruction is a longword because the entire 32-bit contents of two registers are exchanged. The instruction permits the exchange of address registers, data registers, and address and data registers.

One application of EXG is to load an address into a data register and then process it using instructions that act on data registers. Then the reverse operation can be used to return the result to the address register. Doing this preserves the original contents of the data register.

Syntax: EXG Xn, Xn.

A.27 EXT – Extend Register

This instruction allows the sign bit (the most significant bit) to be extended up to the next higher size. Thus if an opcode modifier of .W is used, the bit in position 7 of the lower-order byte will be extended into the rest of the word (in bits 8 to 15). If an opcode modifier of .L is used, the bit in position 15 of the low-order word will be extended into the rest of the long word (bits 16 to 31). If a byte value has to sign-extended to a long word, both an EXT.W and an EXT.L have to be performed on the data register.

Syntax: EXT Dn.

Flags affected: The Negative and Zero flags are set as per the result. The Overflow and Carry are reset to zero, and the Extend flag is unaffected.

A.28 JMP - Jump

This instruction allows execution of the program to be transferred anywhere within the entire addressing space of the 68000. The jump address can be specified using any memory mode except register indirect with postincrement or predecrement. It should be borne in mind that an absolute address specified in a jump instruction will load the program counter immediately with that value. Because absolute addresses are not position- independent. If the program is moved in memory it has to be reassembled if the label is contained within the program. The JMP instruction with an absolute address is more properly used for jumps to static locations such as ROM routines. To keep the jump position-independent, a program-counter-relative address should be specified.

Syntax: JMP address (where address is any addressing mode except (An)+ and -(An))

Flags affected: None.

A.29 JSR - Jump to Subroutine

This instruction allows control to be redirected in a similar manner to the JMP instruction; however, before the jump is made, the address of the following opcode is pushed onto the stack. (See BSR for a description of the stack save process.) Thus a subroutine can perform a task, and when it finishes, it can execute a Return instruction to return to the address saved on the stack. As far as the destination address of the JSR instruction is concerned, the same caveats apply as for the JMP instruction. Absolute addresses, even as labels inside your program, should be avoided where possible to avoid a program which is not position-independent. Unless using such things as ROM routines or memory-mapped hardware locations, which have absolute addresses, use program counter relative or address register indirect addressing.

Syntax: JSR address (where address is any addressing mode except (An)+ and -(An))

Flags affected: None.

A.30 LEA - Load Effective Address

This instruction provides a simple way of loading any address register with the address resulting from nearly any addressing mode. Only two such modes are excluded from the list of possibilities. Due to the fact that address register indirect with postincrement or predecrement represent a dynamically increasing or decreasing addresses, these two modes cannot be used with LEA. But any other address, no matter how complicated, (including address register indirect with displacement and index) can be loaded into the specified address register. This saves performing the address arithmetic within the program. The processor will automatically take the same value as the calculated address - or in other words "the effective address". Only address registers can be used with this instruction, and the destination address register is loaded with a 32-bit long value even though the address will only be 24 bits long.

Syntax: LEA address, An (where address is any addressing mode except postincrement and predecrement)

Flags affected: None.

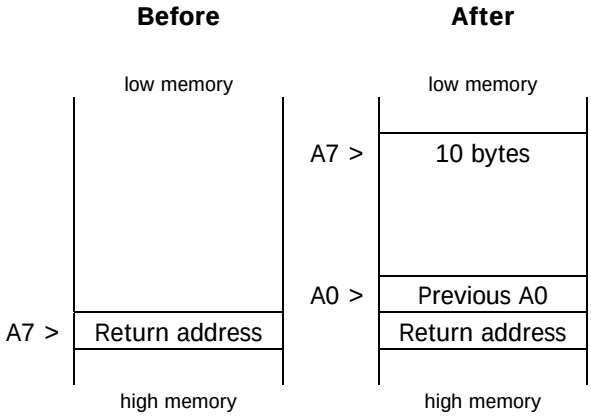
A.31 LINK - Link Subroutine

This instruction is a specialized data area allocation opcode for use by subroutines that require a temporary work area that will be relinquished after use. Normally, when a subroutine has been entered from a JSR or BSR instruction, the return address (that is, the address of the instruction after the JSR or BSR) has automatically been saved on the stack by the processor before transferring control to the subroutine. This is part of the regular linkage for a subroutine call, which is automatically performed by any computer processor. The LINK instruction adds another automatic-linkage option after control has been handed to the subroutine.

Assume the subroutine needs ten bytes of temporary storage in order to perform its function. The ideal place for this would be on the stack, which is the usual place for dynamic registers saves during a program's operation. As the stack pointer saves numbers in a downward direction in memory, simply subtracting ten from the stack pointer register A7 would reserve ten bytes of the stack space with A7 pointing at it. However, A7 may not point to the ten bytes for long, as other items may subsequently be pushed onto the stack changing A7 to point lower in memory. So ideally, another address register should be loaded with the contents of A7 before it was decremented by ten so we have a firm pointer to the stack before it is changed. This is exactly what the LINK instruction does. An address register is elected to save the current pointer to the stack in A7; this assigned will become the pointer to the temporary reserved stack space. The stack pointer A7 is then decremented by however many bytes needed, but before being decremented, the assigned register itself is saved on the stack. This way, the called subroutine can perform a LINK to reserve space, knowing that

it can call yet another subroutine, which can also perform a LINK with no registers being corrupted. The diagram shows what happens.

```
LINK A0,#-10
```



Note that because ten bytes are required on stack going downwards in memory (as per normal stack practice), a negative displacement is specified in the LINK instruction. As the displacement is a signed 16-bit immediate value, a stack displacement of plus or minus 32K can be specified. The address register assigned to point to the top of the reserved space, or stack frame, is generally known as a frame pointer when used in this way. Note that as this register will be used with predecrement instructions, it initially points to one word above the frame.

Syntax: LINK An, #imm where #imm is plus or minus 32K.

Flags affected: None.

A.32 LSL - Logical Shift Left

This instruction shifts the destination operand left by a specified number of bits. If you are shifting a data register, the number of bits to be shifted can be specified as an immediate value or as a value in another data register. The immediate value can be 1 to 8, whereas the data register value can be 1 to 64 (where zero acts as the 64 count). Data registers may be shifted as 8, 16 or 32 bit quantities. Only 16-bit word values can be shifted in memory and then only by one bit. Each bit shifted out of the left-hand side of an operand is placed in the Carry and Extend bits in the condition code register. As shown below, the bit shifted in at the right hand side is always a zero.

C < < | <<< LSL <<< | < 0 X <

Syntax: LSL Dn, Dn

or LSL #imm 3-bit value, Dn

or LSL memory (1 bit shift only)

Flags affected: The Carry and Extend bits are set as per the most significant operand bit before the shift. The Overflow flag is reset to zero. The Negative and Zero flags are set as per result.

A.33 LSR Logical Shift Right

This instruction shifts the destination operand right by a specified number of bits. If you are shifting a data register, the number of bits to be shifted can be specified as an immediate value or as a value in another data register. The immediate value can specify a shift of 1 to 8, while the data register value can specify a shift of 1 to 64 (where zero acts as the 64 count). Data registers may be shifted as 8, 16, or 32 bit quantities. Only 16-bit word values can be

shifted in memory and then only by one bit. Each bit shifted out of the right hand side of an operand is placed in the Carry and Extend bits of the condition code register. As shown below, the bit shifted in at the left hand side is always a zero.

> C 0 MSB > | >>> ASR >>> | > > X

Syntax: LSR Dn, Dn

or LSR #imm 3-bit value, Dn

or LSR memory (1 bit shift only)

Flags affected: The Carry and Extend bits are set as per the least significant operand bit before the shift. The Overflow flag is reset to zero. The Negative and Zero flags are set as per result.

A.34 MOVE - Move Data

This is the 68000's general purpose data-transfer instruction. Using one single opcode, data can be moved from register to register, register to memory, memory to register and memory to memory. The MOVE instruction can also be used to move data to (but not from) the condition code register, thus explicitly setting a particular set of conditions. If you are in privileged (or supervisor) mode, the MOVE instruction can be used to move data to the status register and to or from the user stack pointer. (Privileged mode is not required to move data from the status register.)

With so many potential sources and destinations of data moves, the 68000 makes life easier by allowing all addressing modes to be used for the source. For destination, all except program counter relative addressing modes may be used. With data transfers involving memory and / or data registers, the data transfer can be made using 8, 16, or 32 bit quantities and is specified by appending .B, .W, or .L to the MOVE mnemonic. If the high-order bits of a data register are not involved in the data move, those bits remain unaffected by the transfer. Care should be used when mixing length of operands during routines using MOVE; if a byte is moved from a location using MOVE.B and then stored back again using MOVE.W, it will be stored in a memory location one byte higher than it was fetched from. Similarly, storing it back with MOVE.L would store it three bytes higher than its original location.

If the destination operand of the MOVE is the condition code register, the length of the source operand can only be eight bits. If the status register is involved as either source or destination of the move, only 16-bit transfers allowed. The instruction involving the user stack pointer is the only circumstance under which the 68000 allows optional access to either the user or the system stack pointer. Normally, the stack pointer is accessed as register A7. Whichever of two A7 registers is in effect depends on whether the processor is in supervisor or user mode. However, the supervisor mode may have a need to access the user stack pointer even though it would normally only access the system stack pointer. This is why the privileged mode is required to access a normally unprotected register.

Syntax: MOVE source, destination (where source can be any addressing mode, and destination can be any addressing mode except program counter relative and immediate; either of the above can be CCR, SR and USP (privileged mode only)).

Flags affected: When the MOVE source, destination format is used, the Negative and Zero flags are set as per the data moved, the Overflow and Carry flags are reset to zero and the Extend flag is unaffected. When the MOVE source, CCR / SR formats are used, the flags are set directly from the data. When the MOVE is done with the USP as an operand, no flags are affected.

A.35 MOVEA - Move Address

This specialized version of the MOVE command is used when the destination is an address register. The instruction only allows transfers of 16 or 32 bits in length. Byte transfers are not

allowed with an address register as the destination. Also note that unlike the normal MOVE command, no flag bits are affected.

Syntax: MOVEA source, An (where source is any addressing mode)

Flags affected: None.

A.36 MOVEM - Move Multiple

This variation of the MOVE instruction allows multiple registers to be saved and restored using a single operation. Any of the 16 data or address registers can be moved this way. At the source code level, the registers chosen to be saved or restored are specified to the assembler in a list separated by slashes. Thus, to save D0, D3 and A1, the register list would be specified as D0/D3/A1. If a consecutive number of registers are included in the list, they can be idetified as such by a hyphen. So to save D0, D1, D2, D5 and A1, the register list can be specified as D5/D0-D2/A1. Notice that the order of register between slashes is unimportant; however, when the 68000 saves these registers, it does so in a definite order. It also retrives them in a definite (but opposite) order, so that if the registers are saved on the stack, they can be pulled off in a typical stack-like fashion (that is, last in first out). The order in which the 68000 saves registers is first A7 through A0, and then D7 through D0. Then in reverse order, D0 is restored first, and restoration continues all the way through to A7. As the registers are most often saved in a stack formation, normally an address register is chosen to point to that stack. Then a predecrement addressing mode is used to push the registers down onto the stack. Conversely, when registers are being restored, a postincrement addressing mode is used. As an example, to save two registers at a memory location pointed to by A3, the instruction MOVEM D1/A1, -(A3) might be used. To restore them at another point in program, MOVEM (A3)+, D1/A1 would be correct. Note that registers can only be saved as words or long words. If they are saved as 16-bit words, then when they are restored, the upper half of the register is automtically sign-extended so that bit 15 fills the upper half of the register. Although less memory is used to save registers this way, such a loss of control of the upper 16 bits of every restored register may present problems unless you remain acutely aware of the possible corruption of an upper register half.

The MOVEM instruction may be used with addressing modes other than predecrement and postincrement. By specifying other addressing modes as the source or destination of the multiple transfer, registers can be saved and restored in ascending locations in memory. The same register order is used, but they will not be stacked in at last in, first out order. Note that no flags are affected by this operation. Thus a subroutine can affect the condition code register, restore multiple registers with MOVEM, and return with the condition code register still intact.

Syntax: MOVEM register list, destination address

or MOVEM source address, register list

or MOVEM register list, -(An)

or MOVEM (An)+, register list

Flags affected: None.

A.37 MOVEP - Move Peripheral Data

This variation of the MOVE instruction is used to transfer data between the 68000 and certain peripherals. As input and output on the 68000 is memory-mapped, certain addresses will not actually be memory at all but will instead be external devices. The 68000 has a special design to allow it to use the many hardware interfaces that exist for 8-bit microprocessors, in particular the 6800. What this means to the programmer is that if a peripheral is interfaced to the 68000 and is normally addressed at consecutive address on an 8-bit microprocessor, it will be addressed at every other address on the 68000 due to the design of its peripheral hardware bus. Thus the MOVEP instruction was included to address such peripherals. A long

word of data from a data register can be transferred high byte first to every alternate memory (pheripheral) address with a single MOVEP to the first address.

This also works the other way round in that every other word will be addressed starting with the source address specified in the MOVEP instruction. Only word or long word transfers are allowed. (A normal MOVE would be used for a single byte.) The only addressing mode allowed to specify the memory location is address register indirect with displacement, and only a data register can be used as the other operand.

Syntax: MOVEP Dn, disp(An)

or MOVEP disp(An), Dn (where disp is a 16-bit displacement)

A.38 MOVEQ - Move Quick

This variation on the MOVE instruction allows the quick loading of a data register with an immediate value. The MOVEQ variant works like a MOVE immediate value to the data register except that MOVEQ is much faster and only takes up two bytes in memory. The immediate value that is moved into a data register can only be in the range -128 to +127. This value is sign- extended into the entire 32 bits of the data register, so it is always of type .L despite the small immediate value. As this instruction works so fast, it is quicker to clear a data register with a MOVEQ #0, Dn than to use CLR Dn. MOVEQ cannot, however, be used with address registers or numbers larger than eight bits.

Syntax: MOVEQ #imm 8-bit signed value, Dn.

Flags affected: The Negative and Zero flags are set as per the immediate value; the Overflow and Carry flags are reset to zero, and the Extent flag is unaffected.

A.39 MULS, MULU - Multiply Signed, Unsigned

This instruction allow a multiplication to take place between a 16-bit source operand and the low order 16 bits of a data register. MULS assumes both numbers are signed, whereas MULU assumes both to be unsigned. The source can be a word from any memory location or the low-order 16 bits of a data register. The destination has to be a data register. The result is stored as a 32-bit signed or unsigned value in the destination register. The Negative flag in the condition code register is affected whether or not the operands are signed, and reflects the most significant bit of the result.

Syntax: MULx Dn, Dn

or MULx address, Dn where address is any addressing mode.

Flags affected: The Negative and Zero flags are set as per the result. The Overflow and Carry flags are reset to zero. The Extend flag is unaffected.

A.40 NBCD - Negate Binary Coded Decimal

This specialized arithmetic instruction allows a single byte containing two binary coded decimal digits to be negated. The byte can be contained in the low portion of a data register or in memory. If the number is in memory, any memory addressing mode except program counter relative may be used. If the number is in data register, bits 8 to 31 are not affected.

Syntax: NBCD Dn or NBCD address.

Flags affected: The Negative and Overflow flag is undefined. The Zero flag is set per the contents of register. Carry and Extend are set as per the result of operation.

A.41 NEG, NEGX - Negate Binary, Negate with Extend

This instruction negates its operand. The result is the same as if the operand were subtracted from zero. The operand may be 8, 16, or 32 bits long as specified by the .B, .W, or .L

mnemonic suffix. All flags are affected by this operation. A variation of this instruction exists to facilitate the manipulation of multiple-precision quantities where data is handled in segments. This is achived by using the Extend flag as set or reset from a previous arithmetic operation. The NEGX instruction works by subtracting its operand from zero then subtracting the Extend bit. All flags are affected by the result of the NEGX operation, but the Zero flag is only changed if the result becomes non-zero thus reflecting the nonzero state of a segmented number. For this reason, the Zero flag should be reset before performing code involving multiple use of NEGX.

Syntax: NEG Dn

or NEG address (where address is any addressing mode except program counter relative)

Flags affected: All.

A.42 NOP - No Operation

This instruction is a do-nothing opcode. It is used during program developement to leave room in a section of code. This space can be patched with machine-code instruction as necessary during debugging to test new routines within a previously written machine code level by substituting NOP instruction for the instructions and operands.

Syntax: NOP.

Flags affected: None.

A.43 NOT - Logical NOT

This instruction takes its operand and simply inverts all of its bits. (Each one-bit becomes zero and each zero-bit becomes one.) The operand can either be in a data register or memory and can be 8, 16, or 32 bits in length as per the .B, .W, or .L operand suffix.

Syntax: NOT Dn

or NOT address (where address is any memory addressing mode except program counter relative)

A.44 OR - Logical OR

The OR opcode performs a logical OR operation. A number of bits in the source operand are ORed with the same number of bits in the destination operand where the result is left. The number of bits can be 8, 16, or 32 as the .B, .W, or .L opcode suffix. One or both operands must be a data register.

Syntax: OR Dn, Dn

or OR Dn, address

or OR address, Dn (where address is any addressing mode with the proviso that program counter relative may not be used as destination)

A.45 ORI - Logical OR Immediate

This instruction logically ORs a byte, word, or long word immediate value with the destination. The destination address can be a data register, memory, or one of two special cases: the condition code register and the status register. If the destination is the condition code register, only a byte-length immediate value is allowed. If the destination is the status register, only a word-length immediate value is allowed, and the processor must be in supervisor mode or else a privilege voilation will occur.

Syntax: ORI #imm, Dn

or ORI #imm, address

or ORI.B #imm 8-bit value, CCR

or ORI.W #imm 16-bit value, SR (Privileged).

Flags affected: The Overflow and Carry bits are reset, the Negative and Zero bits set as per the result, and the Extend bit is unaffected.

A.46 PEA - Push Effective Address

This instruction takes the effective address of its operands and pushes it onto the stack as pointed to by the stack pointer A7. The operand can be nearly any addressing mode and is represented as a 32-bit long word. Only two addressing modes are excluded from the list of possibilities. Due to the fact that address register indirect with postincrement or predecrement represent a dynamically increasing or decreasing address, these two modes cannot be used with PEA. But any other address, no matter how complicated, (including address register indirect with displacement and index) can be pushed onto the stack. This saves performing the address arithmetic within the program. The processor will automatically push the same value as the calculated address - or in other words "the effective address". The destination address on the stack is loaded with a 32-bit long value even though the address will only be 24 bits long. No flags are affected by the result of the address calculation.

Syntax: PEA address (where address is any memory addressing mode except postincrement and predecrement)

Flags affected: None.

A.47 RESET - Reset External Devices

This instruction sends out a pulse from the RESET pin of the 68000. It is normally used when a system is first powered up to reset all devices to a known state. It is only likely to be used after that if a hardware fault-condition developes. Because it is such a powerful opcode, it is restricted to use in supervisor mode only.

Syntax: RESET

Flags affected: None.

A.48 ROL, ROXL - Rotate Left, Rotate Extended Left

These two instructions both rotate the destination operand left by a specified number of bits. If you are rotating a data register, the number of bits can be specified as an immediate value or as a value in another data register. The immediate value can be 1 to 8, whereas the data register value can be 1 to 64 (where zero acts as the 64 count). Data registers may be rotated as 8, 16, or 32 bit quantities. Only 16-bit word values can be rotated in memory and then only by one bit. As shown, each bit rotated out of the left hand side of the operand is placed in the Carry bit of the condition code register, and in case of ROXL, also in the Extend bit. The bit rotated in at the right is the most significant bit for ROL or the Extend bit for ROXL. Thus, one more bit is involved in the ROXL rotate than in the ROL rotate. Note that ROL does not affect the Extend flag in the condition code register.

C < | <<< ROL <<< | < C < > > > > > > > > < | <<< ROXL <<< | < < X < > > > > > > > > > > > > > >

Syntax: ROL Dn, Dn

or ROL #imm, Dn

or ROL address.

Flags affected: The Negative flag is set as per most significant bit before the rotate. The Zero flag is set as per resultant operand. The Overflow flag is reset to zero. The Extend flag is unaffected by ROL, but contains the previous most significant bit for ROXL.

A.49 ROR, ROXL - Rotate Right, Rotate Extended Right

These two instructions both rotate the destination operand right by a specified number of bits. If you are rotating a data register, the number of bits can be specified as an immediate value or as a value in another data register. The immediate value can be 1 to 8, whereas the data register value can be 1 to 64 (where zero acts as the 64 count). Data registers may be rotated as 8, 16, or 32 bit quantities. Only 16-bit word values can be rotated in memory and then only by one bit.

As shown, each bit rotated out of the right hand side of the operand is placed in the Carry bit of the condition code register and in case of ROXR, also in the Extend bit. The bit rotated in at the left is the least significant bit for ROR or the Extend bit for ROXR. Thus, one more bit is involved in the ROXR rotate than in the ROR rotate. Note that ROR does not affect the Extend flag in the condition code register.

> | >>> ROR >>> | > C > C < < < < < < < < < > | >>> ROXR >>> | > > X > < < < < < < < < < < < < < <

Syntax: ROR Dn, Dn

or ROR #imm, Dn

or ROR address.

Flags affected: The Negative flag is set as per most significant bit before the rotate. The Zero flag is set as per resultant operand. The Overflow flag is reset to zero. The Extend flag is unaffected by ROR, but contains the previous most significant bit for ROXR.

A.50 RTE, RTR, RTS - Return from Exception, Return and Restore CCR, Return from Subroutine

These instructions change program control by loading the program counter with an execution address previously saved on the stack. The most common version is RTS, which simply pulls the saved address from the stack, increments A7 to allow reuse of the stack space, and reloads the program counter. RTE excepts to find a previously saved status register word on the stack, which it pulls and restores prior to reloading the program counter. As RTE accesses the privileged byte of the status register, it can only be executed in supervisor mode or else a privilege voilation trap will occur. RTR expects to find a previously saved condition code register word on the stack, which it pulls and restores prior to reloading the program counter.

Syntax: RTS

or RTE

or RTR.

Flags affected: No flags are affected by RTS. All flags are reloaded by RTE and RTR.

A.51 SBCD - Subtract Binary Coded Decimal

This instruction a specialized arithmetic instruction that subtracts one bytes from another (only bytes) when each byte containis binary-coded decimal numbers.

The subtraction can be performed either on two data registers or between two memory locations. If performed on bytes in memory, only address register indirect with predecrement can be used. This facilitates easy manipulation of multiple-precision BCD numbers. The extend bit is subtracted along with the BCD bytes to allow this multiprecision data manipulation. Also note that the zero flag is only changed if the result becomes nonzero. Therefore, both the Extend and Zero bits in the condition code register should be perset before the operation is performed. The Extend bit would normally be preset to a zero (to prevent extension on the first subtraction) and the Zero bit to a one (to signify a zero result prior to the first subtraction). A MOVE #4, CCR would preset these flags correctly.

Syntax: SBCD Dn, Dn

Or SBCD -(An), -(An)

Flags affected: The Zero flag is cleared if the result becomes nonzero. The Carry and Extend flags are set if a decimal borrow is generated. The Negative and Overflow bits are undefined.

A.52 Scc - Set from Conditions Codes

This instruction sets a single byte specified in the operand to all zeroes or all ones according to the condition codes. The condition codes which may be used are the same as for the decrement and branch opcode. If the specified condition is true as reflected in the condition code register, the destination byte is set to all ones ($FF). If it is not true, the destination byte is set to zero. The destination can be the low-order byte of a data register or a byte in memory. This instruction is of particular value saving status of a specific condition code.

EQ - Equal to

NE - Not equal to

MI – Minus

PL – Plus

CS - Carry set

CC - Carry clear

VS - Overflow set

VC - Overflow clear

HI – Higher

LS - Less than or same

HS - Higher or same

LO – Lower

GT – Greater

GE - Greater than or equal to

LE - Less than or equal to

LT - Less than

F - False Always false => MOVE #$0

T - True Always true => MOVE #$FF

Syntax: Scc Dn

Or Scc address (where address is any addressing mode except program counter relative)

Flags affected: None.

A.53 STOP - Stop processor and wait

This is a privileged instruction that first copies its operand (which is an immediate word value) into the status register and then halts the processor. The processor will remain in this state until it receives an interrupt that is not masked by the interrupt mask into the status register.

Syntax: STOP #imm 16-bit value (Privileged).

Flags affected: All flags are set as per the immediate value.

A.54 SUB - Subtract Binary

The SUB instruction subtracts the source operand from the destination operand with the result appearing in the destination. It is possible to subtract bytes, words, or long words with this opcode by appending .B, .W, or .L to the mnemonic. Either the source or destination (or both) must be a data register. The source operand can be any memory location or data register, and the destination operand can also be any memory location or data register.

Syntax: SUB Dn, Dn

or SUB address, Dn

or SUB Dn, address

Flags affected: The Extend, Negative, Zero, Overflow, and Carry flags are all affected as per the result of the subtraction.

A.54.1 SUBA - Subtract Address

This variant of the SUB instruction only differs in that an address register is specified as the destination. As an address rather than data is being manipulated, the condition code flags are left unaffected. Only sign-extended words or long words can be subtracted.

A.54.2 SUBI - Subtract Immediate

This variant of the SUB instruction is used to subtract a constant value from the destination. The immediate operand can be any 8-, 16-, or 32-bit value as specified by the .B, .W, or .L opcode suffix. The destination cannot be an address register or a program counter relative address.

Syntax: SUBI #imm, Dn

or SUBI #imm, address (where address is any memory addressing mode except program counter relative)

Flags affected: The Extend, Negative, Zero, Overflow, and Carry flags are all set as per the result of the subtraction.

A.54.3 SUBQ - Subtract Quick

This variant of the SUB instruction is used to subtract a small integer between one and eight from the destination. The destination can be a memory location, a data register, or an address register. If it is an address register, the condition code flags are unaffected and the operand length cannot be a byte.

This operation takes the place of the decrement instruction found on other processors.

Syntax: SUBQ #imm, Rn

or SUBQ #imm, address.

Flags affected: The Extend, Negative, Zero, Overflow, and Carry flags are all set as per the result of the subtraction unless the destination is an address register.

A.54.4 SUBX - Subtract Extended

This variant of the SUB instruction subtracts two numbers and the Extend bit of the condition code register. This allows multiple-precision subtractions to be performed. For this reason, the Zero flag is only affected when a non-zero result is obtained. This means that if multiple numbers are subtracted using SUBX, the Zero flag will stay reset if any of those numbers were non-zero.

Syntax: SUBX Dn, Dn

Or SUBX -(An), -(An).

A.55 SWAP - Swap Data register halves

This instruction takes the lower 16 bits of the specified data register and swaps it with the upper 16 bits. It can only be used with data registers and only on the fixed word length in each half.

Syntax: SWAP Dn.

Flags affected: The Negative and Zero flags are set to reflect the 32-bit result. The Overflow and Carry flags are reset to zero. The Extend flag is unaffected.

A.56 TAS - Test and Set

This is a highly specialized instruction that is used to test a byte in memory or in a data register. When the condition codes are set as per the byte's contents, bit 7 (the most significant bit) of the byte is set to a one. This operation is achived in an uninterruptible read-modify-write cycle. It is the only instruction on the 68000 that uses this method. Its importance lies in the fact that no interrupt can cause a read of the accessed byte before the operation is finished. If the operation were done in two steps, an interrupt could occur before the byte was changed, which would allow the interrupting routine to scan the byte and draw an erroneous conclution as to its status.

Syntax: TAS Dn

or TAS address (where address is any addressing mode except program counter relative)

Flags affected: The Negative and Zero flags are set as per the byte before modification. The Overflow and Carry flags are reset to zero. The Extend flag is unaffected.

A.57 TRAP - Software Trap

This instruction causes a trap to occur in the same manner as if it had been caused by a hardware-detected condition. The processor will jump to one of the 16 special addresses set up in the first 1024 bytes of memory. The actual address that will be jumped to is determined by the operand supplied with the opcode. This will be a number from 0 to 15. The software trap vectors are 32-bit addresses stored in memory starting at location #128. Before the specified vector is taken, the status register and program counter are pushed onto the stack to facilitate a return via an RTE instruction.

Syntax: TRAP #imm (where #imm is an immediate value from 0 to 15)

Flags affected: None.

A.58 TRAPV - Trap if Overflow

This instruction causes a trap to occur to the address in location #28 in low memory if the Overflow flag is set in the condition code register. Before the overflow vector is taken, the status register and program counter are pushed onto the stack to facilitate a return via an RTE instruction.

Syntax: TRAPV

Flags affected: None.

A.59 TST - Test Operand

This instruction causes the processor to scan the operand and set the condition code flags according to the contents. The operand can be 8, 16, or 32 bytes as specified in the .B, .W, or .L opcode modifier. No registers other than the condition code register are changed. The operand can be either a data register or a memory location.

Syntax: TST Dn

or TST address (where address is any addressing mode except program counter relative)

A.60 UNLK - Unlink

This instruction is the reverse of the LINK opcode. It takes the address in the specified address register and loads the stack pointer (A7) with it. This removes any space allocated on the stack for temporary storage. The stack pointer then points at the previous contents of the address register (the frame pointer). This contents would have been placed there by a previous LINK instruction. The frame pointer is automatically reloaded by pulling the value from the stack. Both the frame pointer and the stack pointer are therefore returned to their values before the last LINK. This entire operation is performed automatically by a single UNLK instruction.

Syntax: UNLK An

Flags affected: None.

Appendix B Hardware Register Listing, by Dan Hollis

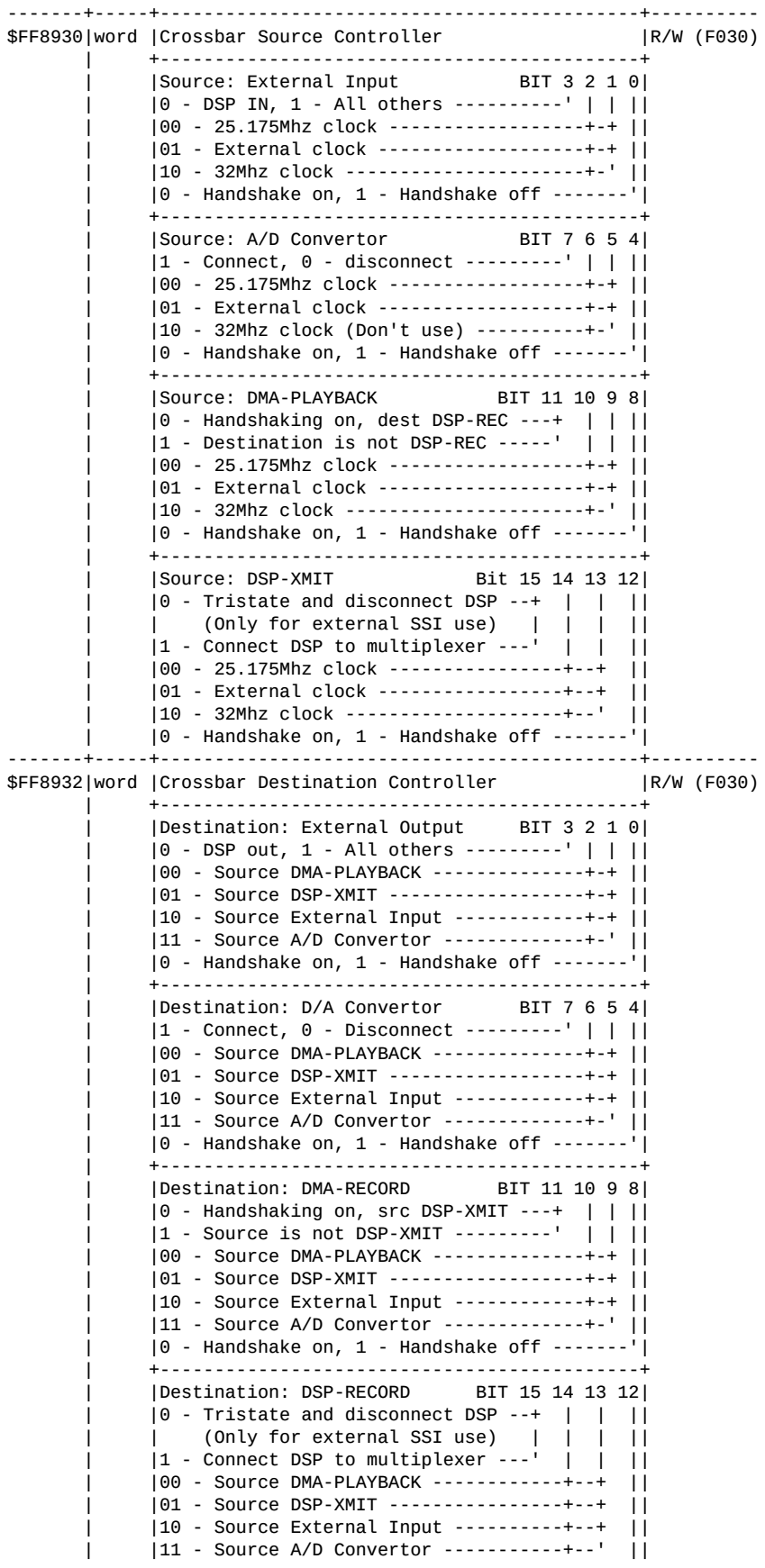
```
            .-------------------------------------------------.
            |Atari ST/STe/MSTe/TT/F030 Hardware Register Listing|
            `-------------------------------------------------'

                        Version 5.81 - 6/15/93
                            By Dan Hollis
                  Copyright (C) 1993 MicroImages Software

------------------------------------------------------------------------------
This document may only be copied unmodified, in its entirety. This document
may only be copied freely, and may not be sold. I make no guarantees as to the
accuracy of this document. I cannot be responsible for the use or misuse of
information contained within this document. Use at your own risk! In any case,
every effort has been taken to ensure this document is as complete and
accurate as possible.
------------------------------------------------------------------------------

Many thanks to the following people for their contributions!

Markus Gutschke, Alexander Herzlinger, Karsten Isakovic, Thomas Binder,
Julian Reschke, Georges Kesseler, Torbjoern Ose


Corrections, additions, or comments should be sent to me. I can be contacted
at the following addresses:

InterNet : dhollis@zero.cypher.com
           dhollis@bitsink.gbdata.com
Snail : Dan Hollis
        P.O. Box 580448
        Houston, TX 77258

Address Size  Description                          Bits used Read/Write
-------+-----+-----------------------------------------+----------
###############ADSPEED Configuration registers             ###########
-------|-----|-----------------------------------------|----------
$F00000|byte |Switch to 16 Mhz                         |W
$F10000|byte |Switch to 8 Mhz                          |W
$F20000|byte |Turn on high speed ROM option in 16 Mhz  |W
$F30000|byte |Turn off high speed ROM option           |W
$F40000|byte |Unknown                                  |W
$F50000|byte |Turn off cache while in 16 Mhz           |W
       |     |   >> Write 0 to an address to set it. <<   |
-------+-----+-----------------------------------------+----------
###############IDE Controller (Falcon, ST-Book, IDE cards) ###########
-------+-----+-----------------------------------------+----------
$F00000|long |Data Register                            |R/W
$F00005|byte |Error Register                           |R
$F00009|byte |Sector Count Register                    |W
$F0000D|byte |Sector Number Register                   |W
$F00011|byte |Cylinder Low Register                    |W
$F00015|byte |Cylinder High Register                   |W
$F00019|byte |Drive Head Register                      |W
$F0001D|byte |Status Register                          |R
$F0001D|byte |Command Register                         |W
$F00039|byte |Alternate Status Register                |R
$F00039|byte |Data Output Register                     |W
-------+-----+-----------------------------------------+----------
```

```
##############ST MMU Controller                          ###########
-------+-----+-----------------------------------------+----------
$FF8001|byte |MMU memory configuration         BIT 3 2 1 0|R/W
       |     |Bank 0                              | | | ||
       |     |00 - 128k --------------------------+-+ | ||
       |     |01 - 512k --------------------------+-+ | ||
       |     |10 - 2m ----------------------------+-+ | ||
       |     |11 - reserved ----------------------+-' | ||
       |     |Bank 1                                  | ||
       |     |00 - 128k --------------------------------+-+|
       |     |01 - 512k --------------------------------+-+|
       |     |10 - 2m ----------------------------------+-+|
       |     |11 - reserved ----------------------------+-'|
-------+-----+-----------------------------------------+----------
##############Falcon030 Processor Control               ###########
-------+-----+-----------------------------------------+----------
$FF8007|byte |Falcon Bus Control           BIT 5 . . 2 . 0|R/W (F030)
       |     |STe Bus Emulation (0 - on) ------'     |   ||
       |     |Blitter (0 - 8mhz, 1 - 16mhz) ---------'   ||
       |     |68030 (0 - 8mhz, 1 - 16mhz) ---------------'|
-------+-----+-----------------------------------------+----------
##############SHIFTER Video Controller                  ###########
-------+-----+-----------------------------------------+----------
$FF8201|byte |Video screen memory position (high Byte)    |R/W
$FF8203|byte |Video screen memory position (mid Byte)     |R/W
$FF820D|byte |Video screen memory position (low Byte)     |R/W  (STe)
$FF8205|byte |Video address pointer (high Byte)           |R
$FF8207|byte |Video address pointer (mid Byte)            |R
$FF8209|byte |Video address pointer (low Byte)            |R
$FF820E|word |Offset to next line                         |R/W (F030)
$FF820F|byte |Width of a scanline (width in words-1)      |R/W  (STe)
$FF8210|word |Width of a scanline (width in words)        |R/W (F030)
$FF8265|byte |Horizontal scroll register (0-15)           |R/W  (STe)
-------+-----+-----------------------------------------+----------
$FF820A|byte |Video synchronization mode           BIT 1 0|R/W
       |     |                     0 - 60hz, 1 - 50hz -' ||
       |     |         0- internal, 1 - external sync ---'|
       |     +-------------------------------------------+
       |     |NOTE: On the TT these bits are reversed, and|
       |     |50/60hz is inoperative on the TT! Check what|
       |     |machine  you are running on before twiddling|
       |     |with these bits.                            |
-------+-----+-----------------------------------------+----------
       |     |                       BIT 11111198 76543210|
       |     |                           543210           |
       |     |    ST color value         .....RRR .GGG.BBB|
       |     |STe/TT Color value         ....rRRR gGGGbBBB|
$FF8240|word |Video palette register 0                    |R/W
    :  |  :  |  :      :       :     :                    | :
$FF825E|word |Video palette register 15                   |R/W
-------+-----+-----------------------------------------+----------
$FF8260|byte |Shifter resolution                  BIT 1 0|R/W
       |     |00 320x200x4 bitplanes (16 colors) ------+-+|
       |     |01 640x200x2 bitplanes (4 colors) -------+-+|
       |     |10 640x400x1 bitplane  (1 colors) -------+-'|
$FF8262|word |TT Shifter resolution             BIT 2 1 0|R/W   (TT)
       |     |000  320x200x4 bitplanes (16 colors) --+-+-+|
       |     |001  640x200x2 bitplanes (4 colors) ---+-+-+|
       |     |010  640x400x1 bitplane  (2 colors) ---+-+-+|
       |     |100  640x480x4 bitplanes (16 colors) --+-+-+|
       |     |110 1280x960x1 bitplane  (1 color) ----+-+-+|
       |     |111  320x480x8 bitplanes (256 colors) -+-+-'|
-------+-----+-----------------------------------------+----------
$FF827E|???? |STACY Display Driver                       |???(STACY)
-------+-----+-----------------------------------------+----------
$FF8400|word |TT Palette  0                              |R/W   (TT)
    :  |  :  | :    :     :                              | :      :
$FF85FE|word |TT Palette 255                             |R/W   (TT)
-------+-----+-----------------------------------------+----------
```

```
##############Falcon030 VIDEL Video Controller            ###########
-------+-----+-----------------------------------------+----------
$FF8006|byte |Monitor Type                          BIT 1 0|R   (F030)
       |     |00 - Monochrome (SM124) -----------------+-+|
       |     |01 - Color (SC1224) --------------------+-+|
       |     |10 - VGA Color -------------------------+-+|
       |     |11 - Television ------------------------+-'|
$FF820E|word |Offset to next line                       |R/W (F030)
$FF8210|word |VWRAP - Linewidth in words                |R/W (F030)
$FF8266|word |SPSHIFT           BIT 10 . 8 . 6 5 4 . . . .|R/W (F030)
       |     |2-colour mode ---------'   |   | | |        |
       |     |Truecolour mode -----------'   | | |        |
       |     |Use external hsync ------------' | |        |
       |     |Use external vsync --------------' |        |
       |     |Bitplane mode ---------------------'        |
       |     +-----------------------------------------+
       |     |      Horizontal Control Registers    (9bit)|
$FF8280|word |HHC - Horizontal Hold Counter               |R   (F030)
$FF8282|word |HHT - Horizontal Hold Timer                 |R/W (F030)
$FF8284|word |HBB - Horizontal Border Begin               |R/W (F030)
$FF8286|word |HBE - Horizontal Border End                 |R/W (F030)
$FF8288|word |HDB - Horizontal Display Begin              |R/W (F030)
$FF828A|word |HDE - Horizontal Display End                |R/W (F030)
$FF828C|word |HSS - Horizontal SS                         |R/W (F030)
$FF828E|word |HFS - Horizontal FS                         |R/W (F030)
$FF8290|word |HEE - Horizontal EE                         |R/W (F030)
       |     +-----------------------------------------+
       |     |      Vertical Control Registers     (10bit)|
$FF82A0|word |VFC - Vertcial Frequency Counter            |R   (F030)
$FF82A2|word |VFT - Vertical Frequency Timer              |R/W (F030)
$FF82A4|word |VBB - Vertical Border Begin                 |R/W (F030)
$FF82A6|word |VBE - Vertical Border End  (count 1/2 lines)|R/W (F030)
$FF82A8|word |VDB - Vertical Display Begin                |R/W (F030)
$FF82AA|word |VDE - Vertical Display End                  |R/W (F030)
$FF82AC|word |VSS - Vertical SS                          |R/W (F030)
       |     +-----------------------------------------+
$FF82C2|word |VCO - Video Control              BIT 3 2 1 0|R/W (F030)
       |     |Quarter pixel width (4x pixels) -----' | | ||
       |     |Half pixel width (double pixels) ------' | ||
       |     |Skip line (interlace) -------------------' ||
       |     |Line doubling ------------------------------'|
-------+-----+-----------------------------------------+----------
##############DMA/WD1772 Disk controller                 ###########
-------+-----+-----------------------------------------+----------
$FF8600|     |Reserved                                  |
$FF8602|     |Reserved                                  |
$FF8604|word |FDC access/sector count                   |R/W
$FF8606|word |DMA mode/status                    BIT 2 1 0|R
       |     |Condition of FDC DATA REQUEST signal --' | ||
       |     |0 - sector count null,1 - not null ------' ||
       |     |0 - no error, 1 - DMA error ---------------'|
$FF8606|word |DMA mode/status        BIT 8 7 6 . 4 3 2 1 .|W
       |     |0 - read FDC/HDC,1 - write ' | | | | | | |  |
       |     |0 - HDC access,1 - FDC access' | | | | | |  |
       |     |0 - DMA on,1 - no DMA ---------' | | | | |  |
       |     |reserved ------------------------' | | | |  |
       |     |0 - FDC reg,1 - sector count reg --' | | |  |
       |     |0 - FDC access,1 - HDC access -------' | |  |
       |     |0 - pin A1 low, 1 - pin A1 high -------' |  |
       |     |0 - pin A0 low, 1 - pin A0 high ---------'  |
$FF8609|byte |DMA base and counter (High byte)            |R/W
$FF860B|byte |DMA base and counter (Mid byte)             |R/W
$FF860D|byte |DMA base and counter (Low byte)             |R/W
-------+-----+-----------------------------------------+----------
##############TT-SCSI DMA Controller                     ###########
-------+-----+-----------------------------------------+----------
$FF8701|byte |TT-SCSI-DMA Address Pointer (Highest byte)  |R/W   (TT)
$FF8703|byte |TT-SCSI-DMA Address Pointer (High byte)     |R/W   (TT)
$FF8705|byte |TT-SCSI-DMA Address Pointer (Low byte)      |R/W   (TT)
```

```
$FF8707|byte |TT-SCSI-DMA Address Pointer (Lowest byte)   |R/W   (TT)
$FF8709|byte |TT-SCSI-DMA Address Counter (Highest byte)  |???   (TT)
$FF870B|byte |TT-SCSI-DMA Address Counter (High byte)     |???   (TT)
$FF870D|byte |TT-SCSI-DMA Address Counter (Low byte)      |???   (TT)
$FF870F|byte |TT-SCSI-DMA Address Counter (Lowest byte)   |???   (TT)
$FF8710|???? |TT-SCSI-DMA Continue Data Register High Word|R/W   (TT)
$FF8712|???? |TT-SCSI-DMA Continue Data Register Low Word |R/W   (TT)
$FF8714|???? |TT-SCSI-DMA Control register                |R/W   (TT)
-------+-----+-------------------------------------------+----------
###############TT-SCSI Drive Controller 5380              ############
-------+-----+-------------------------------------------+----------
$FF8781|byte |Contents of SCSI-Data buses                 |R/W   (TT)
$FF8783|byte |Init-Command Register                       |R/W   (TT)
$FF8785|byte |Transfer Start Register                     |R/W   (TT)
$FF8787|byte |Target-Command Register                     |R/W   (TT)
$FF8789|byte |Bus Status Register                         |R/W   (TT)
$FF878B|byte |Status Register                             |R/W   (TT)
$FF878D|byte |Command Data from SCSI-Bus                  |R/W   (TT)
$FF878F|byte |Reset DMA/Parity error/begin DMA transfer   |R/W   (TT)
-------+-----+-------------------------------------------+----------
###############YM2149 Sound Chip                          ############
-------+-----+-------------------------------------------+----------
$FF8800|byte |Read data/Register select                   |R/W
       |     |Port A (register 14)     BIT 7 6 5 4 3 2 1 0|
       |     |IDE Drive On/OFF ------------' | | | | | | ||    (F030)
       |     |Monitor jack GPO pin ----------+ | | | | | ||
       |     |Internal Speaker On/Off -------' | | | | | ||    (F030)
       |     |Centronics strobe ---------------' | | | | ||
       |     |RS-232 DTR output -----------------' | | | ||
       |     |RS-232 RTS output -------------------' | | ||
       |     |Drive select 1 ------------------------' | ||
       |     |Drive select 0 --------------------------' ||
       |     |Drive side select -------------------------'|
       |     |Port B - (register 15) Parallel port        |
$FF8802|byte |Write data                                  |W
       |     +-------------------------------------------+
       |     |Note:   PSG Registers are now fixed at these|
       |     |addresses.  All  other  addresses are masked|
       |     |out on the Falcon.  Any writes to the shadow|
       |     |registers $8804-$8900 will cause a bus error|
-------+-----+-------------------------------------------+----------
###############DMA Sound System                           ############
-------+-----+-------------------------------------------+----------
$FF8900|byte |Buffer interrupts               BIT 3 2 1 0|R/W (F030)
       |     |TimerA-Int at end of record buffer --' | | ||
       |     |TimerA-Int at end of replay buffer ----' | ||
       |     |MFP-15-Int (I7) at end of record buffer -' ||
       |     |MFP-15-Int (I7) at end of replay buffer ---'|
-------+-----+-------------------------------------------+----------
$FF8901|byte |DMA Control Register     BIT 7 . 5 4 . . 1 0|R/W
       |     |1 - select record register --+   | |     | ||    (F030)
       |     |0 - select replay register --'   | |     | ||    (F030)
       |     |Loop record buffer --------------' |     | ||    (F030)
       |     |DMA Record on --------------------'     | ||    (F030)
       |     |Loop replay buffer ----------------------' ||     (STe)
       |     |DMA Replay on -----------------------------'|     (STe)
-------+-----+-------------------------------------------+----------
$FF8903|byte |Frame start address (high byte)             |R/W  (STe)
$FF8905|byte |Frame start address (mid byte)              |R/W  (STe)
$FF8907|byte |Frame start address (low byte)              |R/W  (STe)
$FF8909|byte |Frame address counter (high byte)           |R    (STe)
$FF890B|byte |Frame address counter (mid byte)            |R    (STe)
$FF890D|byte |Frame address counter (low byte)            |R    (STe)
$FF890F|byte |Frame end address (high byte)               |R/W  (STe)
$FF8911|byte |Frame end address (mid byte)                |R/W  (STe)
$FF8913|byte |Frame end address (low byte)                |R/W  (STe)
-------+-----+-------------------------------------------+----------
$FF8920|byte |DMA Track Control            BIT 5 4 . . 1 0|R/W (F030)
       |     |00 - Set DAC to Track 0 ---------+-+     | ||
```

```
       |     |01 - Set DAC to Track 1 ---------+-+     | ||
       |     |10 - Set DAC to Track 2 ---------+-+     | ||
       |     |11 - Set DAC to Track 3 ---------+-'     | ||
       |     |00 - Play 1 Track ----------------------+-+|
       |     |01 - Play 2 Tracks ---------------------+-+|
       |     |10 - Play 3 Tracks ---------------------+-+|
       |     |11 - Play 4 Tracks ---------------------+-'|
-------+-----+-------------------------------------------+----------
$FF8921|byte |Sound mode control       BIT 7 6 . . . . 1 0|R/W  (STe)
       |     |0 - Stereo, 1 - Mono --------' |         | ||
       |     |0 - 8bit ----------------------+         | ||
       |     |1 - 16bit (F030 only) ---------'         | ||    (F030)
       |     |Frequency control bits                   | ||
       |     |00 - Off (F030 only) --------------------+-+|    (F030)
       |     |00 - 6258hz frequency (STe only) --------+-+|
       |     |01 - 12517hz frequency -------------------+-+|
       |     |10 - 25033hz frequency -------------------+-+|
       |     |11 - 50066hz frequency -------------------+-'|
       |     |Samples are always signed. In stereo mode,  |
       |     |data is arranged in pairs with high pair the|
       |     |left channel, low pair right channel. Sample|
       |     |length must ALWAYS be even in either mono or|
       |     |stereo mode.                                |
       |     |Example: 8 bit Stereo : LRLRLRLRLRLR        |
       |     |        16 bit Stereo : LLRRLLRRLLRR (F030) |
       |     |2 track 16 bit stereo : LLRRllrrLLRR (F030) |
-------+-----+-------------------------------------------+----------
###############STe Microwire Controller (STe only!)       ############
-------+-----+-------------------------------------------+----------
$FF8922|byte |Microwire data register                    |R/W  (STe)
$FF8924|byte |Microwire mask register                    |R/W  (STe)
       |     +-------------------------------------------+
       |     |Volume/tone controller commands(Address %10)|
       |     |Master Volume                 10 011 DDDDDD|
       |     |Left Volume                   10 101 .DDDDD|
       |     |Right Volume                  10 100 .DDDDD|
       |     |Treble                        10 010 ..DDDD|
       |     |Bass                          10 001 ..DDDD|
       |     |Mixer                         10 000 ....DD|
       |     +-------------------------------------------+
       |     |Volume/tone controller values              |
       |     |Master Volume     : 0-40   (0 -80dB, 40=0dB)|
       |     |Left/Right Volume : 0-20    (0 80dB, 20=0dB)|
       |     |Treble/bass       : 0-12 (0 -12dB, 12 +12dB)|
       |     |Mixer             : 0-3 (0 -12dB, 1 mix PSG)|
       |     |                    (2 don't mix,3 reserved)|
       |     +-------------------------------------------+
       |     |Procedure: Set mask register to $7ff. Read  |
       |     |data register and save original value. Write|
       |     |data register. Compare data register with   |
       |     |original value, repeat until data register  |
       |     |returns to original value to ensure data has|
       |     |been sent over the interface.               |
       |     +-------------------------------------------+
       |     |Interrupts: Timer A can be set to interrupt |
       |     |at the end of a frame. Alternatively, the   |
       |     |GPI7 (MFP mono detect) can be used to       |
       |     |generate interrupts thereby freeing up Timer|
       |     |A. In this case, the active edge $FFFA03    |
       |     |must be set by or-ing the active edge       |
       |     |$FFFA03 with the contents of $FF8260:       |
       |     |$FF8260 - 2 (mono)     or.b  #$80 with edge |
       |     |$FF8260 - 0,1 (colour) and.b #$7F with edge |
       |     |This will generate an interrupt at the START|
       |     |of a frame, instead of at the end as with   |
       |     |Timer A. To generate an interrupt at the END|
       |     |of a frame, simply reverse the edge values. |
-------+-----+-------------------------------------------+----------
###############Falcon030 DMA/DSP Controllers               ############
```

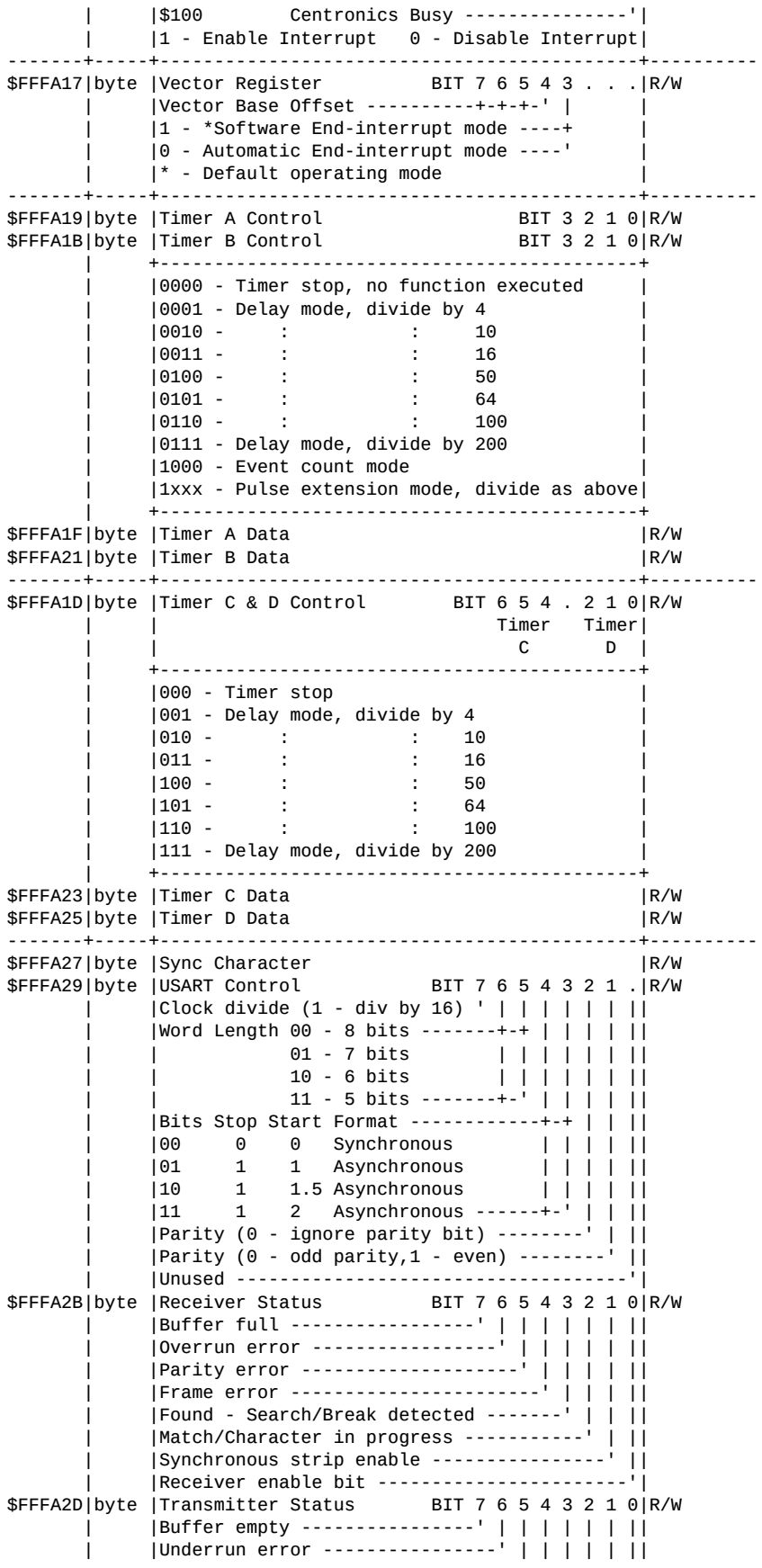

$FF8930|word |Crossbar Source Controller |R/W (F030)
Source: External Input BIT 3 2 1 0
0 - DSP IN, 1 - All others
00 - 25.175Mhz clock
01 - External clock
10 - 32Mhz clock
0 - Handshake on, 1 - Handshake off
Source: A/D Convertor BIT 7 6 5 4
1 - Connect, 0 - disconnect
00 - 25.175Mhz clock
01 - External clock
10 - 32Mhz clock (Don't use)
0 - Handshake on, 1 - Handshake off
Source: DMA-PLAYBACK BIT 11 10 9 8
0 - Handshaking on, dest DSP-REC
1 - Destination is not DSP-REC
00 - 25.175Mhz clock
01 - External clock
10 - 32Mhz clock
0 - Handshake on, 1 - Handshake off
Source: DSP-XMIT Bit 15 14 13 12
0 - Tristate and disconnect DSP
(Only for external SSI use)
1 - Connect DSP to multiplexer
00 - 25.175Mhz clock
01 - External clock
10 - 32Mhz clock
0 - Handshake on, 1 - Handshake off
$FF8932|word |Crossbar Destination Controller |R/W (F030)
Destination: External Output BIT 3 2 1 0
0 - DSP out, 1 - All others
00 - Source DMA-PLAYBACK
01 - Source DSP-XMIT
10 - Source External Input
11 - Source A/D Convertor
0 - Handshake on, 1 - Handshake off
Destination: D/A Convertor BIT 7 6 5 4
1 - Connect, 0 - Disconnect
00 - Source DMA-PLAYBACK
01 - Source DSP-XMIT
10 - Source External Input
11 - Source A/D Convertor
0 - Handshake on, 1 - Handshake off
Destination: DMA-RECORD BIT 11 10 9 8
0 - Handshaking on, src DSP-XMIT
1 - Source is not DSP-XMIT
00 - Source DMA-PLAYBACK
01 - Source DSP-XMIT
10 - Source External Input
11 - Source A/D Convertor
0 - Handshake on, 1 - Handshake off
Destination: DSP-RECORD BIT 15 14 13 12
0 - Tristate and disconnect DSP
(Only for external SSI use)
1 - Connect DSP to multiplexer
00 - Source DMA-PLAYBACK
01 - Source DSP-XMIT
10 - Source External Input
11 - Source A/D Convertor

```
       |     |0 - Handshake on, 1 - Handshake off -------'|
-------+-----+--------------------------------------------+----------
$FF8934|byte |Frequency Divider External Clock BIT 3 2 1 0|R/W (F030)
       |     |0000 - STe-Compatible mode                  |
       |     |0001 - 1111  Divide by 256 and then number  |
-------+-----+--------------------------------------------+----------
$FF8935|byte |Frequency Divider Internal Sync  BIT 3 2 1 0|R/W (F030)
       |     |0000 - STe-Compatible mode   1000 - 10927Hz*|
       |     |0001 - 49170Hz               1001 -  9834Hz |
       |     |0010 - 32780Hz               1010 -  8940Hz*|
       |     |0011 - 24585Hz               1011 -  8195Hz |
       |     |0100 - 19668Hz               1100 -  7565Hz*|
       |     |0101 - 16390Hz               1101 -  7024Hz*|
       |     |0110 - 14049Hz*              1110 -  6556Hz*|
       |     |0111 - 12292Hz               1111 -  6146Hz*|
       |     |          * - Invalid for CODEC             |
-------+-----+--------------------------------------------+----------
$FF8936|byte |Record Tracks Select                BIT 1 0|R/W (F030)
       |     |00 - Record 1 Track ---------------------+-+|
       |     |01 - Record 2 Tracks ---------------------+-+|
       |     |10 - Record 3 Tracks ---------------------+-+|
       |     |11 - Record 4 Tracks ---------------------+-'|
-------+-----+--------------------------------------------+----------
$FF8937|byte |CODEC Input Source from 16bit adder  BIT 1 0|R/W (F030)
       |     |Source: Multiplexer ----------------------' ||
       |     |Source: A/D Convertor -----------------------'|
-------+-----+--------------------------------------------+----------
$FF8938|byte |CODEC ADC-Input for L+R Channel      BIT 1 0|R/W (F030)
       |     |0 - Microphone, 1 - Soundchip            L R|
-------+-----+--------------------------------------------+----------
$FF8939|byte |Channel amplification          BIT LLLL RRRR|R/W (F030)
       |     |      Amplification is in +1.5dB steps      |
-------+-----+--------------------------------------------+----------
$FF893A|word |Channel attenuation       BIT LLLL RRRR ....|R/W (F030)
       |     |       Attenuation is in -1.5dB steps       |
-------+-----+--------------------------------------------+----------
$FF893C|byte |CODEC-Status                        BIT 1 0|R/W (F030)
       |     |Left Channel Overflow -------------------' ||
       |     |Right Channel Overflow ---------------------'|
-------+-----+--------------------------------------------+----------
$FF8941|byte |GPx Data Direction                BIT 2 1 0|R/W (F030)
       |     |0 - In, 1 - Out ----------------------+-+-'|
       |     | For the GP0-GP2 pins on the DSP connector  |
-------+-----+--------------------------------------------+----------
$FF8943|byte |GPx Data Port                     BIT 2 1 0|R/W (F030)
-------+-----+--------------------------------------------+----------
###############TT Clock Chip                              ############
-------+-----+--------------------------------------------+----------
$FF8961|byte |Register select                             |????  (TT)
$FF8963|byte |Data of selected clock chip registers       |????  (TT)
-------+-----+--------------------------------------------+----------
###############Blitter (Not present on a TT!)             ############
-------+-----+--------------------------------------------+----------
$FF8A00|word |Halftone-RAM, Word 0                        |R/W (Blit)
    :  |  :  |    :     :     :  :                        | :     :
$FF8A1E|word |Halftone-RAM, Word 15                       |R/W (Blit)
$FF8A20|word |Source X Increment            (signed,even)|R/W (Blit)
$FF8A22|word |Source Y Increment            (signed,even)|R/W (Blit)
$FF8A24|long |Source Address Register       (24 bit,even)|R/W (Blit)
$FF8A28|word |Endmask 1           (First write of a line)|R/W (Blit)
$FF8A2A|word |Endmask 2           (All other line writes)|R/W (Blit)
$FF8A2C|word |Endmask 3            (Last write of a line)|R/W (Blit)
$FF8A2E|word |Destination X Increment       (signed,even)|R/W (Blit)
$FF8A30|word |Destination Y Increment       (signed,even)|R/W (Blit)
$FF8A32|long |Destination Address Register  (24 bit,even)|R/W (Blit)
$FF8A36|word |Words per Line in Bit-Block       (0=65536)|R/W (Blit)
$FF8A38|word |Lines per Bit-Block              (0=65536)|R/W (Blit)
$FF8A3A|byte |Halftone Operation Register        BIT 1 0|R/W (Blit)
       |     |00 - All ones --------------------------+-+|
```

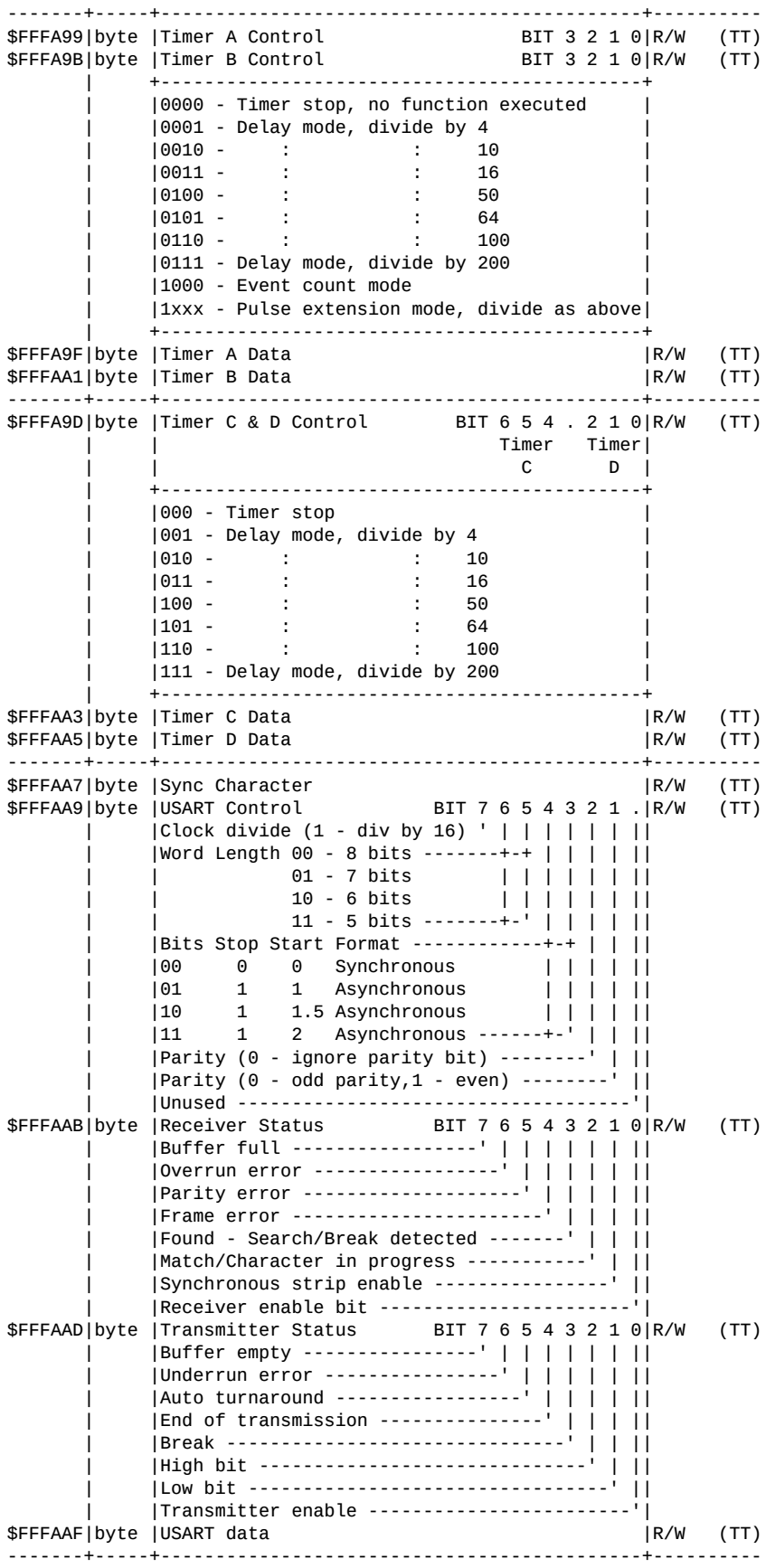
```
       |     |01 - Halftone --------------------------+-+|
       |     |10 - Source ----------------------------+-+|
       |     |11 - Source AND Halftone ----------------+-'|
$FF8A3B|byte |Logical Operation Register       BIT 3 2 1 0|R/W (Blit)
       |     |0000 All zeros ---------------------+-+-+-+|
       |     |0001 Source AND destination ---------+-+-+-+|
       |     |0010 Source AND NOT destination -----+-+-+-+|
       |     |0011 Source -------------------------+-+-+-+|
       |     |0100 NOT source AND destination -----+-+-+-+|
       |     |0101 Destination --------------------+-+-+-+|
       |     |0110 Source XOR destination ---------+-+-+-+|
       |     |0111 Source OR destination ----------+-+-+-+|
       |     |1000 NOT source AND NOT destination -+-+-+-+|
       |     |1001 NOT source XOR destination -----+-+-+-+|
       |     |1010 NOT destination ----------------+-+-+-+|
       |     |1011 Source OR NOT destination ------+-+-+-+|
       |     |1100 NOT source ---------------------+-+-+-+|
       |     |1101 NOT source OR destination ------+-+-+-+|
       |     |1110 NOT source OR NOT destination --+-+-+-+|
       |     |1111 All ones -----------------------+-+-+-'|
$FF8A3C|byte |Line Number Register     BIT 7 6 5 . 3 2 1 0|R/W (Blit)
       |     |BUSY ------------------------' | |   | | | ||
       |     |0 - Share bus, 1 - Hog bus ----' |   | | | ||
       |     |SMUDGE mode ---------------------'   | | | ||
       |     |Halftone line number ----------------+-+-+-'|
$FF8A3D|byte |SKEW Register            BIT 7 6 . . 3 2 1 0|R/W (Blit)
       |     |Force eXtra Source Read -----' |     | | | ||
       |     |No Final Source Read ----------'     | | | ||
       |     |Source skew --------------------------+-+-+-'|
-------+-----+-------------------------------------------+----------
###############SCC-DMA (TT Only!)                        ############
-------+-----+-------------------------------------------+----------
$FF8C01|byte |DMA-Address Pointer (Highest Byte)          |R/W   (TT)
$FF8C03|byte |DMA-Address Pointer (High Byte)             |R/W   (TT)
$FF8C05|byte |DMA-Address Pointer (Low Byte)              |R/W   (TT)
$FF8C07|byte |DMA-Address Pointer (Lowest Byte)           |R/W   (TT)
$FF8C09|byte |DMA-Address Counter (Highest-Byte)          |R/W   (TT)
$FF8C0B|byte |DMA-Address Counter (High-Byte)             |R/W   (TT)
$FF8C0D|byte |DMA-Address Counter (Low-Byte)              |R/W   (TT)
$FF8C0F|byte |DMA-Address Counter (Lowest-Byte)           |R/W   (TT)
$FF8C10|byte |Continue Data Register (High-Word)          |R/W   (TT)
$FF8C12|byte |Continue Data register (Low-Word)           |R/W   (TT)
$FF8C14|byte |Control register                           |R/W   (TT)
-------+-----+-------------------------------------------+----------
###############SCC Z8530 SCC (MSTe/TT/F030)               ############
-------+-----+-------------------------------------------+----------
$FF8C81|byte |Channel A - Control-Register               |R/W  (SCC)
$FF8C83|byte |Channel A - Data-Register                  |R/W  (SCC)
$FF8C85|byte |Channel B - Control-Register               |R/W  (SCC)
$FF8C87|byte |Channel B - Data-Register                  |R/W  (SCC)
-------+-----+-------------------------------------------+----------
###############MSTe/TT VME Bus                            ############
-------+-----+-------------------------------------------+----------
$FF8E01|byte |VME sys_mask             BIT 7 6 5 4 . 2 1 .|R/W  (VME)
$FF8E03|byte |VME sys_stat             BIT 7 6 5 4 . 2 1 .|R/W  (VME)
       |     |_SYSFAIL in VMEBUS ----------' | | |   | |  |program
       |     |MFP --------------------------' | |   | |  |autovec
       |     |SCC ------------------------------' |   | |  |autovec
       |     |VSYNC ------------------------------'   | |  |program
       |     |HSYNC ----------------------------------' |  |program
       |     |System software INT --------------------'  |program
       |     +-------------------------------------------+
       |     |Reading sys_mask resets pending int-bits in |
       |     |sys_stat, so read sys_stat first.           |
-------+-----+-------------------------------------------+----------
$FF8E05|byte |VME sys_int                          BIT 0|R/W  (VME)
       |     |Setting bit 0 to 1 forces an INT of level 1 |Vector $64
       |     |INT must be enabled in sys_mask to use it   |
-------+-----+-------------------------------------------+----------
```

```
$FF8E0D|byte |VME vme_mask             BIT 7 6 5 4 3 2 1 .|R/W  (VME)
$FF8E0F|byte |VME vme_stat             BIT 7 6 5 4 3 2 1 .|R/W  (VME)
       |     |_IRQ7 from VMEBUS -----------' | | | | | |  |program
       |     |_IRQ6 from VMEBUS/MFP ---------' | | | | |  |program
       |     |_IRQ5 from VMEBUS/SCC -----------' | | | |  |program
       |     |_IRQ4 from VMEBUS -----------------' | | |  |program
       |     |_IRQ3 from VMEBUS/soft --------------' | |  |prog/autov
       |     |_IRQ2 from VMEBUS ---------------------' |  |program
       |     |_IRQ1 from VMEBUS -----------------------'  |program
       |     +-------------------------------------------+
       |     |MFP-int and SCC-int are hardwired to the    |
       |     |VME-BUS-ints (or'ed)                        |
       |     |Reading vme_mask resets pending int-bits in |
       |     |vme_stat, so read vme_stat first.           |
-------+-----+--------------------------------------------+----------
$FF8E07|byte |VME vme_int                          BIT 0|R/W   (TT)
       |     |Setting bit 0 to 1 forces an INT of level 3 |Vector $6C
       |     |INT must be enabled in vme_mask to use it   |
-------+-----+--------------------------------------------+----------
$FF8E09|byte |General purpose register - does nothing     |R/W   (TT)
$FF8E0B|byte |General purpose register - does nothing     |R/W   (TT)
-------+-----+--------------------------------------------+----------
###############Mega STe Cache/Processor Control           ############
-------+-----+--------------------------------------------+----------
$FF8E21|byte |Mega STe Cache/Processor Control            |R/W (MSTe)
-------+-----+--------------------------------------------+----------
###############STe Extended Joystick/Lightpen Ports       ############
-------+-----+--------------------------------------------+----------
$FF9200|???? |Fire buttons 1-4                            |R    (STe)
$FF9202|???? |Joysticks 1-4                               |R    (STe)
$FF9210|???? |Paddle 0 Position                           |R    (STe)
$FF9212|???? |Paddle 1 Position                           |R    (STe)
$FF9214|???? |Paddle 2 Position                           |R    (STe)
$FF9216|???? |Paddle 3 Position                           |R    (STe)
$FF9220|???? |Lightpen X-Position                         |R    (STe)
$FF9222|???? |Lightpen Y-Position                         |R    (STe)
-------+-----+--------------------------------------------+----------
###############Falcon VIDEL Palette Registers             ############
-------+-----+--------------------------------------------+----------
       |     |     BIT 33222222 22221111 11111198 76543210|
       |     |         10987654 32109876 543210           |
       |     |         RRRRRR.. GGGGGG.. ........ BBBBBB..|
$FF9800|long |Palette Register  0                         |R/W (F030)
   :   |  :  |   :        :     :                         | :     :
$FF98FC|long |Palette Register 255                        |R/W (F030)
-------+-----+--------------------------------------------+----------
###############Falcon DSP Host Interface                  ############
-------+-----+--------------------------------------------+----------
$FFA200|byte |Interrupt Ctrl Register  BIT 7 6 5 4 3 . 1 0|R/W (F030)
X:$FFE9|     |INIT bit --------------------' | | | |   | ||
       |     |00 - Interupt mode (DMA off) --+-+ | |   | ||
       |     |01 - 24-bit DMA mode ----------+-+ | |   | ||
       |     |10 - 16-bit DMA mode ----------+-+ | |   | ||
       |     |11 - 8-bit DMA mode -----------+-' | |   | ||
       |     |Host Flag 1 ----------------------' |   | ||
       |     |Host Flag 0 -------------------------'   | ||
       |     |         Host mode Data transfers:       | ||
       |     |              Interrupt mode             | ||
       |     |00 - No interrupts (Polling) ------------+-+|
       |     |01 - RXDF Request (Interrupt) -----------+-+|
       |     |10 - TXDE Request (Interrupt) -----------+-+|
       |     |11 - RXDF and TXDE Request (Interrupts) -+-+|
       |     |                 DMA Mode                | ||
       |     |00 - No DMA ----------------------------+-+|
       |     |01 - DSP to Host Request (RX) -----------+-+|
       |     |10 - Host to DSP Request (TX) -----------+-+|
       |     |11 - Undefined (Illegal) ----------------+-'|
$FFA201|byte |Command Vector Register                     |R/W (F030)
X:$FFE9|     |                         BIT 7 . . 4 3 2 1 0|
```

```
       |     |Host Command Bit (Handshake)-'     | | | | ||
       |     |Host Vector (0-31) ----------------+-+-+-+-'|
$FFA202|byte |Interrupt Status Reg     BIT 7 6 . 4 3 2 1 0|R   (F030)
X:$FFE8|     |ISR Host Request ------------' |   | | | | ||
       |     |ISR DMA Status ----------------'   | | | | ||
       |     |Host Flag 3 ----------------------' | | | ||
       |     |Host Flag 2 ------------------------' | | ||
       |     |ISR Transmitter Ready (TRDY) ----------' | ||
       |     |ISR Transmit Data Register Empty (TXDE) -' ||
       |     |ISR Receive Data Register Full (RXDF) -----'|
$FFA203|byte |Interrupt Vector Register                   |R/W (F030)
$FFA204|byte |Unused                                      |    (F030)
$FFA205|byte |DSP-Word High                               |R/W (F030)
X:$FFEB|     |                                            |
$FFA206|byte |DSP-Word Mid                                |R/W (F030)
X:$FFEB|     |                                            |
$FFA207|byte |DSP-Word Low                                |R/W (F030)
X:$FFEB|     |                                            |
-------+-----+--------------------------------------------+----------
###############MFP 68901 - Multi Function Peripheral Chip  ###########
-------+-----+--------------------------------------------+----------
       |     |MFP Master Clock is 2,457,600 cycles/second |
-------+-----+--------------------------------------------+----------
$FFFA01|byte |Parallel Port Data Register                 |R/W
-------+-----+--------------------------------------------+----------
$FFFA03|byte |Active Edge Register     BIT 7 6 5 4 . 2 1 0|R/W
       |     |Monochrome monitor detect ---' | | | | | ||
       |     |RS-232 Ring indicator ---------' | | | | ||
       |     |FDC/HDC interrupt ---------------' | | | ||
       |     |Keyboard/MIDI interrupt -----------' | | ||
       |     |Reserved ----------------------------' | ||
       |     |RS-232 CTS (input) --------------------' ||
       |     |RS-232 DCD (input) ----------------------' ||
       |     |Centronics busy ---------------------------'|
       |     +-------------------------------------+
       |     |   When port bits are used for input only:  |
       |     |0 - Interrupt on pin high-low conversion    |
       |     |1 - Interrupt on pin low-high conversion    |
-------+-----+--------------------------------------------+----------
$FFFA05|byte |Data Direction           BIT 7 6 5 4 3 2 1 0|R/W
       |     |0 - In, 1 - Out -------------+-+-+-+-+-+-+-'|
-------+-----+--------------------------------------------+----------
$FFFA07|byte |Interrupt Enable A       BIT 7 6 5 4 3 2 1 0|R/W
$FFFA0B|byte |Interrupt Pending A      BIT 7 6 5 4 3 2 1 0|R/W
$FFFA0F|byte |Interrupt In-service A   BIT 7 6 5 4 3 2 1 0|R/W
$FFFA13|byte |Interrupt Mask A         BIT 7 6 5 4 3 2 1 0|R/W
       |     |MFP Address                 | | | | | | | ||
       |     |$13C GPI7-Monochrome Detect -' | | | | | ||
       |     |$138   RS-232 Ring Detector ---' | | | | ||
       |     |$134 (STe sound)    Timer A -----' | | | ||
       |     |$130    Receive buffer full -------' | | ||
       |     |$12C          Receive error ---------' | ||
       |     |$128      Send buffer empty -----------' ||
       |     |$124            Send error -------------' ||
       |     |$120 (HBL)          Timer B ---------------'|
       |     |1 - Enable Interrupt   0 - Disable Interrupt|
-------+-----+--------------------------------------------+----------
$FFFA09|byte |Interrupt Enable B       BIT 7 6 5 4 3 2 1 0|R/W
$FFFA0D|byte |Interrupt Pending B      BIT 7 6 5 4 3 2 1 0|R/W
$FFFA11|byte |Interrupt In-service B   BIT 7 6 5 4 3 2 1 0|R/W
$FFFA15|byte |Interrupt Mask B         BIT 7 6 5 4 3 2 1 0|R/W
       |     |MFP Address                 | | | | | | | ||
       |     |$11C               FDC/HDC -' | | | | | | ||
       |     |$118            Keyboard/MIDI ---' | | | | ||
       |     |$114 (200hz clock)  Timer C -----' | | | | ||
       |     |$110 (USART timer)  Timer D -------' | | | ||
       |     |$10C           Blitter done ---------' | | ||
       |     |$108     RS-232 CTS - input -----------' | ||
       |     |$104     RS-232 DCD - input -------------' ||
```


$100 Centronics Busy
1 - Enable Interrupt 0 - Disable Interrupt
$FFFA17|byte |Vector Register BIT 7 6 5 4 3 . . .|R/W
Vector Base Offset
1 - *Software End-interrupt mode
0 - Automatic End-interrupt mode
* - Default operating mode
$FFFA19|byte |Timer A Control BIT 3 2 1 0|R/W
$FFFA1B|byte |Timer B Control BIT 3 2 1 0|R/W
0000 - Timer stop, no function executed
0001 - Delay mode, divide by 4
0010 - : : 10
0011 - : : 16
0100 - : : 50
0101 - : : 64
0110 - : : 100
0111 - Delay mode, divide by 200
1000 - Event count mode
1xxx - Pulse extension mode, divide as above
$FFFA1F|byte |Timer A Data |R/W
$FFFA21|byte |Timer B Data |R/W
$FFFA1D|byte |Timer C & D Control BIT 6 5 4 . 2 1 0|R/W
Timer C
Timer D
000 - Timer stop
001 - Delay mode, divide by 4
010 - : : 10
011 - : : 16
100 - : : 50
101 - : : 64
110 - : : 100
111 - Delay mode, divide by 200
$FFFA23|byte |Timer C Data |R/W
$FFFA25|byte |Timer D Data |R/W
$FFFA27|byte |Sync Character |R/W
$FFFA29|byte |USART Control BIT 7 6 5 4 3 2 1 .|R/W
Clock divide (1 - div by 16)
Word Length 00 - 8 bits
01 - 7 bits
10 - 6 bits
11 - 5 bits
Bits Stop Start Format
00 0 0 Synchronous
01 1 1 Asynchronous
10 1 1.5 Asynchronous
11 1 2 Asynchronous
Parity (0 - ignore parity bit)
Parity (0 - odd parity,1 - even)
Unused
$FFFA2B|byte |Receiver Status BIT 7 6 5 4 3 2 1 0|R/W
Buffer full
Overrun error
Parity error
Frame error
Found - Search/Break detected
Match/Character in progress
Synchronous strip enable
Receiver enable bit
$FFFA2D|byte |Transmitter Status BIT 7 6 5 4 3 2 1 0|R/W
Buffer empty
Underrun error

```
       |     |Auto turnaround -----------------' | | | | ||
       |     |End of transmission ---------------' | | | ||
       |     |Break -------------------------------' | | ||
       |     |High bit -----------------------------' | ||
       |     |Low bit -------------------------------' ||
       |     |Transmitter enable ------------------------'|
$FFFA2F|byte |USART data                                  |R/W
-------+-----+--------------------------------------------+----------
#############Floating Point Coprocessor (in MSTe)        ###########
-------+-----+--------------------------------------------+----------
$FFFA40|???? |FP_Stat    Response-Register                |??? (MSTe)
$FFFA42|???? |FP_Ctl     Control-Register                 |??? (MSTe)
$FFFA44|???? |FP_Save    Save-Register                    |??? (MSTe)
$FFFA46|???? |FP_Restor  Restore-Register                 |??? (MSTe)
$FFFA4A|???? |FP_Cmd     Command-Register                 |??? (MSTe)
$FFFA4E|???? |FP_Ccr     Condition-Code-Register          |??? (MSTe)
$FFFA50|???? |FP_Op      Operanden-Register               |??? (MSTe)
$FFFA54|???? |FP_Selct   Register Select                  |??? (MSTe)
$FFFA58|???? |FP_Iadr    Instruction Address              |??? (MSTe)
-------+-----+--------------------------------------------+----------
###############MFP 68901 #2 (MFP2) - TT Only              ###########
-------+-----+--------------------------------------------+----------
$FFFA81|byte |Parallel Port Data Register                  |R/W   (TT)
-------+-----+--------------------------------------------+----------
$FFFA83|byte |Active Edge Register     BIT 7 6 5 4 3 2 1 0|R/W   (TT)
       |     +--------------------------------------------+
       |     |   When port bits are used for input only:  |
       |     |0 - Interrupt on pin high-low conversion    |
       |     |1 - Interrupt on pin low-high conversion    |
-------+-----+--------------------------------------------+----------
$FFFA85|byte |Data Direction           BIT 7 6 5 4 3 2 1 0|R/W   (TT)
       |     |0 - Pin becomes an input                    |
       |     |1 - Pin becomes an output                   |
-------+-----+--------------------------------------------+----------
$FFFA87|byte |Interrupt Enable A       BIT 7 6 5 4 3 2 1 0|R/W   (TT)
$FFFA8B|byte |Interrupt Pending A      BIT 7 6 5 4 3 2 1 0|R/W   (TT)
$FFFA8F|byte |Interrupt In-service A   BIT 7 6 5 4 3 2 1 0|R/W   (TT)
$FFFA93|byte |Interrupt Mask A         BIT 7 6 5 4 3 2 1 0|R/W   (TT)
       |     |MFP Address                  | | | | | | | ||
       |     |$17C  General Purpose Input -' | | | | | | ||
       |     |$178  General Purpose Input ---' | | | | | ||
       |     |$174               Timer A -----' | | | | ||
       |     |$170    Receive buffer full -------' | | | ||
       |     |$16C          Receive error ---------' | | ||
       |     |$168      Send buffer empty -----------' | ||
       |     |$164             Send error -------------' ||
       |     |$160               Timer B ---------------'|
       |     |1 - Enable Interrupt   0 - Disable Interrupt|
-------+-----+--------------------------------------------+----------
$FFFA89|byte |Interrupt Enable B       BIT 7 6 5 4 3 2 1 0|R/W   (TT)
$FFFA8D|byte |Interrupt Pending B      BIT 7 6 5 4 3 2 1 0|R/W   (TT)
$FFFA91|byte |Interrupt In-service B   BIT 7 6 5 4 3 2 1 0|R/W   (TT)
$FFFA95|byte |Interrupt Mask B         BIT 7 6 5 4 3 2 1 0|R/W   (TT)
       |     |MFP Address                  | | | | | | | ||
       |     |$15C  General Purpose Input -' | | | | | | ||
       |     |$158  General Purpose Input ---' | | | | | ||
       |     |$154               Timer C -----' | | | | ||
       |     |$150               Timer D -------' | | | ||
       |     |$14C  General Purpose Input ---------' | | ||
       |     |$148  General Purpose Input -----------' | ||
       |     |$144  General Purpose Input -------------' ||
       |     |$140  General Purpose Input ---------------'|
       |     |1 - Enable Interrupt   0 - Disable Interrupt|
-------+-----+--------------------------------------------+----------
$FFFA97|byte |Vector Register          BIT 7 6 5 4 3 . . .|R/W   (TT)
       |     |Vector Base Offset ----------+-+-+-' |      |
       |     |1 - *Software End-interrupt mode ----+      |
       |     |0 - Automatic End-interrupt mode ----'      |
       |     |* - Default operating mode                  |
```


$FFFA99|byte |Timer A Control BIT 3 2 1 0|R/W (TT)
$FFFA9B|byte |Timer B Control BIT 3 2 1 0|R/W (TT)
|0000 - Timer stop, no function executed
|0001 - Delay mode, divide by 4
|0010 - : : 10
|0011 - : : 16
|0100 - : : 50
|0101 - : : 64
|0110 - : : 100
|0111 - Delay mode, divide by 200
|1000 - Event count mode
|1xxx - Pulse extension mode, divide as above
$FFFA9F|byte |Timer A Data |R/W (TT)
$FFFAA1|byte |Timer B Data |R/W (TT)
$FFFA9D|byte |Timer C & D Control BIT 6 5 4 . 2 1 0|R/W (TT)
Timer C
Timer D
|000 - Timer stop
|001 - Delay mode, divide by 4
|010 - : : 10
|011 - : : 16
|100 - : : 50
|101 - : : 64
|110 - : : 100
|111 - Delay mode, divide by 200
$FFFAA3|byte |Timer C Data |R/W (TT)
$FFFAA5|byte |Timer D Data |R/W (TT)
$FFFAA7|byte |Sync Character |R/W (TT)
$FFFAA9|byte |USART Control BIT 7 6 5 4 3 2 1 .|R/W (TT)
|Clock divide (1 - div by 16)
|Word Length 00 - 8 bits
01 - 7 bits
10 - 6 bits
11 - 5 bits
|Bits Stop Start Format
|00 0 0 Synchronous
|01 1 1 Asynchronous
|10 1 1.5 Asynchronous
|11 1 2 Asynchronous
|Parity (0 - ignore parity bit)
|Parity (0 - odd parity,1 - even)
|Unused
$FFFAAB|byte |Receiver Status BIT 7 6 5 4 3 2 1 0|R/W (TT)
|Buffer full
|Overrun error
|Parity error
|Frame error
|Found - Search/Break detected
|Match/Character in progress
|Synchronous strip enable
|Receiver enable bit
$FFFAAD|byte |Transmitter Status BIT 7 6 5 4 3 2 1 0|R/W (TT)
|Buffer empty
|Underrun error
|Auto turnaround
|End of transmission
|Break
|High bit
|Low bit
|Transmitter enable
$FFFAAF|byte |USART data |R/W (TT)

```
##############6850 ACIA I/O Chips                          ###########
-------+-----+-------------------------------------------+----------
$FFFC00|byte |Keyboard ACIA control                       |R/W
$FFFC02|byte |Keyboard ACIA data                          |R/W
$FFFC04|byte |MIDI ACIA control                           |R/W
$FFFC06|byte |MIDI ACIA data                              |R/W
-------+-----+-------------------------------------------+----------
##############Realtime Clock                               ###########
-------+-----+-------------------------------------------+----------
$FFFC21|byte |S_Units                                     |???
$FFFC23|byte |S_Tens                                      |???
$FFFC25|byte |M_Units                                     |???
$FFFC27|byte |M_Tens                                      |???
$FFFC29|byte |H_Units                                     |???
$FFFC2B|byte |H_Tens                                      |???
$FFFC2D|byte |Weekday                                     |???
$FFFC2F|byte |Day_Units                                   |???
$FFFC31|byte |Day_Tens                                    |???
$FFFC33|byte |Mon_Units                                   |???
$FFFC35|byte |Mon_Tens                                    |???
$FFFC37|byte |Yr_Units                                    |???
$FFFC39|byte |Yr_Tens                                     |???
$FFFC3B|byte |Cl_Mod                                      |???
$FFFC3D|byte |Cl_Test                                     |???
$FFFC3F|byte |Cl_Reset                                    |???
-------+-----+-------------------------------------------+----------
$FA0000|     |                                           |
    :  |     |128K ROM expansion cartridge port          |R
$FBFFFF|     |                                           |
-------+-----+-------------------------------------------+----------
$FC0000|     |                                           |
    :  |     |192K System ROM                            |R
$FEFFFF|     |                                           |
-------+-----+-------------------------------------------+----------
```

Cookie Jar
Atari "Official" Cookies

This section is preliminary. I can't think of a better format yet for this section, so it will be ugly for a while.

```
Cookie  Description
-------+-----------------------------------------------------------------
_CPU   | CPU Type
       | Processor type is represented in decimal in the lowest
       | byte. (0 - 68000, 40 - 68040)
_VDO   | Video Yype
       | Shifter Type                                        BIT 17 16
       | 0 - ST    1 - STe    2 - TT    3 - Falcon030
_FDC   | Floppy Drive Controller
       | Floppy Format                                       Bit 25 24
       | 0 - DD (Normal floppy interface)
       | 1 - HD (1.44 MB with 3.5")
       | 2 - ED (2.88 MB with 3.5")
       | Controller ID                                       BIT 23-0
       | 0 - No information available
       | 'ATC' - Fully compatible interface built in a way that it
       |         behaves like part of the system.
       | 'DP1' - "DreamPark Development", all other ID's beginning with
       |         "DP" are reserved for Dreampark.
_FLK   | File Locking
       | If present, GEMDOS supports file locking. Value represents the
       | version number of the expansion.
_NET   | Network Type
       | If present, there is GEMDOS network support. Points to 2 longs
       | - The first is the ID of the producer, and the second is the
       | version number.
_SLM   | SLM Driver
```

```
       | Diablo-driver for the SLM laser printer. Value points to a
       | non-documented structure.
_INF   | .INF Patch
       | When present, STEFIX (patch program for TOS 1.06) is active.
_SND   | Sound Type
       | Sound Hardware    BIT 4 3 2 1 0
       | Connection Matrix ----' | | | |
       | DSP56001 ---------------' | | |
       | 16 Bit DMA Sound ---------' | |
       | 8 Bit DMA Sound ------------' |
       | YM2149 -----------------------'
_MCH   | Machine Type
       | Hardware Description High Word + Low word
       | $00000000 - ST/Mega ST
       | $0001xxxx - STe compatible machines
       |      0000 - STe
       |      0001 - ST Book
       |      0010 - Mega STe
       | $00020000 - TT
       | $00030000 - Falcon 030
_SWI   | Configuration Switches
       | State of configuration switches (MSTe/TT only)
_FRB   | Fast Ram Buffer
       | (TT specific) 64k buffer for ACSI DMA
       | 0 - no buffers assigned       >0 - address of FastRam buffer
_FPU   | FPU Type
       |          High word - hardware    Low word - software
       |  0 - No FPU
       |  1 - SFP004 or compatible 68881
       |  2 - 68881 or 68882 unsure which one    3 - plus SFP004
       |  4 - 68881 for sure                     5 - plus SFP004
       |  6 - 68882 for sure                     7 - plus SFP004
       |  8 - 68040's internal FPU               9 - plus SFP004
_OOL   | PoolFix
       | Value corresponds to PoolFix version
_AKP   | Keyboard/Language Configuration
       | Keyboard Configuration                              Low byte
       | 1 - German       2 - French       4 - Spanish    5 - Italian
       | 7 - Swiss French    8 - Swiss German
       | All others - English
       | Language Configuration                           Lowest byte
       | 1,8 - German     2,7 - French     4 - Spanish    5 - Italian
       | All others - English
_IDT   | International Date/Time Format
       | Time Format                                          BIT 12
       | 0 - AM/PM   1 - 24 hours
       | Date Format                                        BIT 9 8
       | 0 - MMDDYY  1 - DDMMYY  2 - YYMMDD  3 - YYDDMM
       | Separator for date                               Lowest Byte
       | ASCII Value (i.e. "." or "/")
MiNT   | MiNT
       | Present if MiNT/MultiTOS is active. Value represents the
       | version number of the MiNT kernel in hex (0x104 = 1.04)
-------+-------------------------------------------------------------
```

68000 Exception Vector Assignments

```
Vector Number   Address   Space   Assignment
--------------+---------+-------+--------------------------------
0              |0/$0     |SP     |Reset:Initial SSP
1              |4/$4     |SP     |Reset:Initial PC
Reset vector (0) requires four words, unlike other vectors which only
require two words, and is located in the supervisor program space.
--------------+---------+-------+--------------------------------
2              |8/$8     |SD     |Bus Error
3              |12/$C    |SD     |Address Error
4              |16/$10   |SD     |Illegal Instruction
5              |20/$14   |SD     |Zero Divide
6              |24/$18   |SD     |CHK, CHK2 Instruction
```

7	|28/$1C	|SD	|cpTRAPcc, TRAPcc, TRAPV Instruction
8	|32/$20	|SD	|Privilege Violation
9	|36/$24	|SD	|Trace
10	|40/$28	|SD	|Line 1010 Emulator (LineA)
11	|44/$2C	|SD	|Line 1111 Emulator (LineF)
12	|48/$30	|SD	|(Unassigned, Reserved)
13 (68030)	|52/$34	|SD	|Coprocessor Protocol Violation
14 (68010)	|56/$38	|SD	|Format Error
15	|60/$3C	|SD	|Uniinitialised Interrupt Vector
16-23	|64/$40	|SD	|(Unassigned, Reserved)
	|95/$5F	|SD	|-
24	|96/$60	|SD	|Spurious Interrupt

Spurious interrupt vector is taken when there is a bus error during interrupt processing.

68000 Auto-Vector Interrupt Table

25	|100/$64	|SD	|Level 1 Int Autovector (TT VME)
26	|104/$68	|SD	|Level 2 Int Autovector (HBL)
27	|108/$6C	|SD	|Level 3 Int Autovector (TT VME)
28	|112/$70	|SD	|Level 4 Int Autovector (VBL)
29	|116/$74	|SD	|Level 5 Int Autovector
30	|120/$78	|SD	|Level 6 Int Autovector (MFP)
31	|124/$7C	|SD	|Level 7 Int Autovector
32-47	|128/$80	|SD	|Trap Instruction Vectors
	|191/$BF	|SD	|(Trap #n = vector number 32+n)

Math Coprocessor Vectors (68020 and higher)

48	|192/$C0	|SD	|FFCP Branch or Set
	|	|	| on Unordered Condition
49	|196/$C4	|SD	|FFCP Inexact Result
50	|200/$C8	|SD	|FFCP Divide by Zero
51	|204/$CC	|SD	|FFCP Underflow
52	|208/$D0	|SD	|FFCP Operand Error
53	|212/$D4	|SD	|FFCP Overflow
54	|216/$D8	|SD	|FFCP Signaling NAN
55	|220/$DC	|SD	|(Unassigned, Reserved)
56	|224/$E0	|SD	|MMU Configuration Error
57	|228/$E4	|SD	|MC68851, not used by MC68030
58	|232/$E8	|SD	|MC68851, not used by MC68030
59-63	|236/$EC	|SD	|(Unassigned, Reserved)
	|255/$FF	|SD	|-
64-254	|256/$100	|SD	|User Defined Interrupt Vectors
	|1019/$3FB	|SD	|-
255	|1020/$3FC	|SD	|DSP-IRQ Vector (F030)

Appendix C ASCII Table, by Stephen McNabb

Below is a table of all the standard ASCII characters and their hexadecimal values, originally by Stephen McNabb.

ASCII = American Standard Code for Information Interchange

		Least significant															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Most significant	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	ST
	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Appendix D List of VT-52 Escape sequences

Below is a table of all the VT-52 Escape sequences, originally by Stephen McNabb.

This is a list of the available escape sequences that can be accessed using Cconout(). To use these sequences you must first output and escape character using Cconout() and then the appropriate characters.

D.1 Example

To clear the screen the following code would be used:

```
move.w    #27,-(sp)          escape character
move.w    #$2,-(sp)          use Cconout() function
trap      #1                 use gemdos
addq.l    #4,sp              tidy up stack

move.w    #'E',-(sp)         use Clear screen sequences
move.w    #$2,-(sp)          use Cconout() function
trap      #1                 use gemdos
addq.l    #4,sp              tidy up stack
```

D.2 List

Char.	Action
A	Moves cursor up one line. If already at the top of the screen then it has no effect.
B	Moves cursor down one line. If already at the bottom of the screen then it has no effect.
C	Moves cursor one character to the right. If already at the right hand side of the screen then it has no effect.
D	Moves cursor one character to the left. If already at the left hand side of the screen then it has no effect.
E	Clears the screen and places the cursor at the top-left hand corner of the screen.
H	Places cursor at the top-left hand corner of the screen without clearing the screen.
I	Moves cursor up one line. If at the top of the screen then all other lines are scrolled down one line.
J	Erases all text from current cursor position to the end of the screen.
K	Erases all text from current cursor position to the end of the current line.
L	Inserts a new line, by scrolling all lines, below and including the current line the cursor is on, down one line. The cursor is positioned at the start of the new blank line.
M	Deletes the current line the cursor is on, scrolls all lines below it up one and inserts a blank line at the bottom. The cursor is positioned at the start of the current line.
Y	Positions cursor on screen. The following two values from Cconout() specify the row and column of the cursor. Values are single byte binary numbers and begin at 1.

Char.	Action
b	Sets the foreground colour index of the text to be displayed. The following value from Cconout() specifies the colour index to be used. The four least significant bits of the value are used.
c	Sets the background colour index of the text to be displayed. The following value from Cconout() specifies the colour index to be used. The four least significant bits of the value are used.
d	Erases all text from the screen from the beginning of the screen to the current cursor position.
e	Makes the cursor visible, usually after using ESC 'f'.
f	Makes the cursor invisible. It can still be moved about the screen.
j	Saves the current cursor position. To be used with ESC 'k'.
k	Restores the cursor position that was saved using ESC 'j'. If the cursor position has not been saved then the cursor is placed at the top-left hand corner of the screen.
l	Erases the current line that the cursor is on. The lines below it are not scrolled up. The cursor is positioned at the start of the line.
o	Erase all the text from the start of the current line to the current cursor position on that line.
p	Enables reverse video text. The background and foreground colours are swapped.
q	Disables reverse video text.
v	Switches text wrap on. Characters written past the end of the line is displayed on the next line.
w	Switches text wrap off. Characters written past the end of the line are written over each other at the right hand side of the screen.

Appendix E Initlib.s

```
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

* save the old palette; old_palette
                    move.l   #old_palette,a0    put backup address in a0
                    movem.l  $ffff8240,d0-7     all palettes in d0-d7
                    movem.l  d0-7,(a0)          move data into old_palette
* end palette save

* saves the old screen adress
                    move.w   #2,-(a7)           get physbase
                    trap     #14
                    addq.l   #2,a7
                    move.l   d0,old_screen      save old screen address
* end screen save

* save the old resolution into old_resolution
* and change resolution to low (0)
                    move.w   #4,-(a7)           get resolution
                    trap     #14
                    addq.l   #2,a7
                    move.w   d0,old_resolution  save resolution

                    move.w   #0,-(a7)           low resolution
                    move.l   #-1,-(a7)          keep physbase
                    move.l   #-1,-(a7)          keep logbase
                    move.w   #5,-(a7)           change screen
                    trap     #14
                    add.l    #12,a7
* end resolution save

                    rts


restore
* restores the old resolution and screen adress
                    move.w   old_resolution,d0  res in d0
                    move.w   d0,-(a7)           push resolution
                    move.l   old_screen,d0      screen in d0
                    move.l   d0,-(a7)           push physbase
                    move.l   d0,-(a7)           push logbase
                    move.w   #5,-(a7)           change screen
                    trap     #14
                    add.l    #12,a7
* end resolution and screen adress restore

* restores the old palette
                    move.l   #old_palette,a0    palette pointer in a0
                    movem.l  (a0),d0-d7         move palette data
                    movem.l  d0-d7,$ffff8240    smack palette in
* end palette 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


old_palette         ds.l      8

old_resolution      dc.w      0

old_stack           dc.l      0

old_screen          dc.l      0
```

Appendix F MC68000 Instruction Execution Times

These are the times for instructions, most of it is self explanatory. On the ST at 8 Mhz you need to round all times to multiples of four. i.e 10 becomes 12. Please note that execution instruction times are generally irrelevant when you have a instruction cache, ie a greater than 68000 processor and i doubt that these numbers will hold true for anything except a 68000 even if you turn the cache off. Also note that it isn't usually worth spending ages trying to optimise your code by using faster instructions. If your code is too slow, then you will probably need to use a different method to achieve you're aims.

F.1 MOVE Instructions

.b.w/.l	dn	an	(an)	(an)+	-(an)	d(an)	d(an.Ri)	abs.s	abs.l
dn	4/4	4/4	8/12	8/12	8/14	12/16	14/18	12/16	16/20
an	4/4	4/4	8/12	8/12	8/14	12/16	14/18	12/16	16/20
(an)	8/12	8/12	12/20	12/20	12/20	16/24	18/26	16/24	20/28
(an)+	8/12	8/12	12/20	12/20	12/20	16/24	18/26	16/24	20/28
-(an)	10/14	10/14	14/22	14/22	14/22	18/26	20/28	18/26	22/30
d(an)	12/16	12/16	16/24	16/24	16/24	20/28	22/30	20/28	24/32
d(an,Ri)	14/18	14/18	18/26	18/26	18/26	22/30	24/32	22/30	26/34
abs.s	12/16	12/16	16/24	16/24	16/24	20/28	22/30	20/28	24/32
abs.l	16/20	16/20	20/28	20/28	20/28	24/32	26/34	24/32	28/36
d(pc)	12/16	12/16	16/24	16/24	16/24	20/28	22/30	20/28	24/32
d(pc,Ri)	14/18	14/18	18/26	18/26	18/26	22/30	24/32	22/30	26/34
Immediate	8/12	8/12	12/20	12/20	12/20	16/24	18/26	16/24	20/28

F.1.1 Time to calculate effective addresses.

	(an)	(an)+	-(an)	d(an)	d(an.Ri)	abs.s	abs.l	d(pc)	d(pc.ri)	Imm
.b.w/.l	4/8	4/8	6/10	8/12	10/14	8/12	12/16	8/12	10/14	4/8

The time taken to calculate the effective address must be added to instructions that affect a memory address.

F.2 Standard Instructions

.b.w/.l	ea,an	ea,dn	dn,mem	
add	8/6(8)	4/6(8)	8/12	(8) time if effective address
and	-	4/6(8)	8/12	is direct.
cmp	6/6	4/6	-	
divs	-	158max	-	Add effective address times
divu	-	140max	-	from above for memory
eor	-	4/8	8/12	addresses.
muls	-	70max	-	
mulu	-	70max	-	
or	-	4/6(8)	8/12	
sub	8/6(8)	4/6(8)	8/12	

F.3 Immediate Instructions

.b.w/.l	#,dn	#,an	#,mem	
addi	8/16	-	12/20	
addq	4/8	8/8	8/12	moveq.l only
andi	8/16	-	12/20	nbcd+tas.b only
cmpi	8/14	8/14	8/12	
eori	8/16	-	12/20	scc false/true
moveq	4	-	-	
ori	8/16	-	12/20	add effective address
subi	8/16	-	12/20	times from above
subq	4/8	8/8	8/12	for mem addresses
clr	4/6	4/6	8/12	single operand
nbcd	6	6	8	instructions
neg	4/6	4/6	8/12	
negx	4/6	4/6	8/12	
not	4/6	4/6	8/12	
scc	4/6	4/6	8/8	
tas	4	4	10	
tst	4/4	4/4	4/4	

F.4 Shift/rotate instructions.

.b.w/.l	dn	an	mem	
asr,asl	6/8	6/8	8	memory is byte only
lsr,lsl	6/8	6/8	8	register add 2x
ror,rol	6/8	6/8	8	shift count
roxr,roxl	6/8	6/8	8	

	(an)	(an)+	-(an)	d(an)	d(an.ri)	abs.s	abs.l	d(pc)	d(pc.ri)
jmp	8	-	-	10	14	10	12	10	14
jsr	16	-	-	18	22	18	20	18	22
lea	4	-	-	8	12	8	12	8	12
pea	12	-	-	16	20	16	20	16	20
movem m>r (t=4)*	12	12	-	16	18	16	20	16	18
movem r>m (t=5)*	8	-	8	12	14	12	16	-	-

* add t x number of registers for .w, 2t x number of registers for .l

F.5 Bit Instructions

.b/.l	register .l only	memory .b only
bchg	8/12	8/12
bclr	10/14	8/12
bset	8/12	8/12
btst	6/10	4/8

F.6 Exception Periods

Address Error	50
Bus Error	50
Interrupt	44
Illegal Instr.	34
Privilege Viol.	34
Trace	34

F.7 Other Instructions

Add effective address times from above for memory addresses

.b.w/.l	dn,dn	m,m	
addx	4/8	18/30	
cmpm	-	12/20	
subx	4/8	18/30	
abcd	6	18	.b only
sbcd	6	18	.b only
Bcc	.b/.w	10/10	8/12
bra	.b/.w	10/10	-
bsr	.b/.w	18/18	-
DBcc	t/f	10	12/14
chk	-	40	max 8
trap	-	34	-
trapv	-	34	4

		reg<>mem
movep	.w/.l	16/24

	Reg	Mem		Reg
andi to ccr	20	-	**move from usp**	4
andi to sr	20	-	**nop**	4
eori to ccr	20	-	**ori to ccr**	20
eori to sr	20	-	**ori to sr**	20
exg	6	-	**reset**	132
ext	4	-	**rte**	20
link	18	-	**rtr**	20
move to ccr	12	12	**rts**	16
move to sr	12	12	**stop**	4
move from sr	6	8	**swap**	4
move to usp	4	-	**unlk**	12

Appendix G Pixel Timings, by Jim Boulton

[Extracted from Jim Boulton's "ST Internals"]

All the following processor timings are based on a bog standard 8MHz MC68000 as found in all standard ST's.

	50Hz	60Hz
Clock cycles per line	512	508
NOPs per scan line	128	127
Scan lines per VBL	313	315
Clock cycles per VBL	160256	
NOPs per VBL	40064	
Pixels per clock cycle (low res)	1	
Pixels per clock cycle (med res)	2	
Pixels per clock cycle (high res)	4	
Pixels per NOP (low res)	4	
Pixels per NOP (med res)	8	
Pixels per NOP (high res)	16	

Appendix H Intelligent Keyboard (IKBD) Protocol

H.1 Introduction

The Atari Corp. Intelligent Keyboard (ikbd) is a general purpose keyboard controller that is flexible enough that it can be used in a variety of products without modification. The keyboard, with its microcontroller, provides a convenient connection point for a mouse and switch-type joysticks.

The ikbd processor also maintains a time-of-day clock with one second resolution.

The ikbd has been designed to be general enough that it can be used with a variety of new computer products. Product variations in a number of keyswitches, mouse resolution, etc. can be accommodated.

The ikbd communicates with the main processor over a high speed bi-directional serial interface. It can function in a variety of modes to facilitate different applications of the keyboard, joysticks, or mouse. Limited use of the controller is possible in applications in which only a unidirectional communications medium is available by carefully designing the default modes.

H.2 Keyboard

The keyboard always returns key make/break scan codes. The ikbd generates keyboard scan codes for each key press and release. The key scan make (key closure) codes start at 1, and are defined in Chapter H.9 Scan Codes (on page 168). For example, the ISO key position in the scan code table should exist even if no keyswitch exists in that position on a particular keyboard. The break code for each key is obtained by ORing 0x80 with the make code.

The special codes 0xF6 through 0xFF are reserved for use as follows:

0xF6	status report
0xF7	absolute mouse position record
0xF8-0xFB	relative mouse position records (lsbs determined by mouse button states)
0xFC	time-of-day
0xFD	joystick report (both sticks)
0xFE	joystick 0 event
0xFF	joystick 1 event

The two shift keys return different scan codes in this mode. The ENTER key and the RETurn key are also distinct.

H.3 Mouse

The mouse port should be capable of supporting a mouse with resolution of approximately 200 counts (phase changes or 'clicks') per inch of travel. The mouse should be scanned at a rate that will permit accurate tracking at velocities up to 10 inches per second.

The ikbd can report mouse motion in three distinctly different ways. It can report relative motion, absolute motion in a coordinate system maintained within the ikbd, or by converting mouse motion into keyboard cursor control key equivalents.

The mouse buttons can be treated as part of the mouse or as additional keyboard keys.

H.3.1 Relative Position Reporting

In relative position mode, the ikbd will return relative mouse position records whenever a mouse event occurs. A mouse event consists of a mouse button being pressed or released, or motion in either axis exceeding a settable threshold of motion. Regardless of the threshold, all bits of resolution are returned to the host computer.

Note that the ikbd may return mouse relative position reports with significantly more than the threshold delta x or y. This may happen since no relative mouse motion events will be generated:

(a) while the keyboard has been 'paused' (the event will be stored until keyboard communications is resumed)

(b) while any event is being transmitted.

The relative mouse position record is a three byte record of the form (regardless of keyboard mode):

```
%111110xy          ; mouse position record flag
                   ; where y is the right button state
                   ; and x is the left button state
X                  ; delta x as twos complement integer
Y                  ; delta y as twos complement integer
```

Note that the value of the button state bits should be valid even if the MOUSE BUTTON ACTION has set the buttons to act like part of the keyboard. If the accumulated motion before the report packet is generated exceeds the +127...-128 range, the motion is broken into multiple packets.

Note that the sign of the delta y reported is a function of the Y origin selected.

H.3.2 Absolute Position reporting

The ikbd can also maintain absolute mouse position. Commands exist for reseting the mouse position, setting X/Y scaling, and interrogating the current mouse position.

H.3.3 Mouse Cursor Key Mode

The ikbd can translate mouse motion into the equivalent cursor keystrokes. The number of mouse clicks per keystroke is independently programmable in each axis. The ikbd internally maintains mouse motion information to the highest resolution available, and merely generates a pair of cursor key events for each multiple of the scale factor.

Mouse motion produces the cursor key make code immediately followed by the break code for the appropriate cursor key. The mouse buttons produce scan codes above those normally assigned for the largest envisioned keyboard (i.e. LEFT=0x74 & RIGHT=0x75).

H.4 Joystick

H.4.1 Joystick Event Reporting

In this mode, the ikbd generates a record whenever the joystick position is changed (i.e. for each opening or closing of a joystick switch or trigger).

The joystick event record is two bytes of the form:

```
1111111x           ; Joystick event marker
                   ; where x is Joystick 0 or 1
%x000yyyy          ; where yyyy is the stick position
                   ; and x is the trigger
```

H.4.2 Joystick Interrogation

The current state of the joystick ports may be interrogated at any time in this mode by sending an 'Interrogate Joystick' command to the ikbd.

The ikbd response to joystick interrogation is a three byte report of the form

```
0xFD              ; joystick report header
%x000yyyy         ; Joystick 0
%x000yyyy         ; Joystick 1
                  ; where x is the trigger
                  ; and yyy is the stick position
```

H.4.3 Joystick Monitoring

A mode is available that devotes nearly all of the keyboard communications time to reporting the state of the joystick ports at a user specifiable rate.

It remains in this mode until reset or commanded into another mode. The PAUSE command in this mode not only stop the output but also temporarily stops scanning the joysticks (samples are not queued).

H.4.4 Fire Button Monitoring

A mode is provided to permit monitoring a single input bit at a high rate. In this mode the ikbd monitors the state of the Joystick 1 fire button at the maximum rate permitted by the serial communication channel. The data is packed 8 bits per byte for transmission to the host. The ikbd remains in this mode until reset or commanded into another mode. The PAUSE command in this mode not only stops the output but also temporarily stops scanning the button (samples are not queued).

H.4.5 Joystick Key Code Mode

The ikbd may be commanded to translate the use of either joystick into the equivalent cursor control keystroke(s). The ikbd provides a single breakpoint velocity joystick cursor.

Joystick events produce the make code, immediately followed by the break code for the appropriate cursor motion keys. The trigger or fire buttons of the joysticks produce pseudo key scan codes above those used by the largest key matrix envisioned (i.e. JOYSTICK0=0x74, JOYSTICK1=0x75).

H.5 Time-of-Day Clock

The ikbd also maintains a time-of-day clock for the system. Commands are available to set and interrogate the timer-of-day clock. Time-keeping is maintained down to a resolution of one second.

H.6 Status Inquiries

The current state of ikbd modes and parameters may be found by sending status inquiry commands that correspond to the ikbd set commands.

H.7 Power-Up Mode

The keyboard controller will perform a simple self-test on power-up to detect major controller faults (ROM checksum and RAM test) and such things as stuck keys. Any keys down at power-up are presumed to be stuck, and their BREAK (sic) code is returned (which without the preceding MAKE code is a flag for a keyboard error). If the controller self-test completes without error, the code 0xF0 is returned. (This code will be used to indicate the version/rlease of the ikbd controller. The first release of the ikbd is version 0xF0, should there be a second release it will be 0xF1, and so on.) The ikbd defaults to a mouse position reporting with threshold of 1 unit in either axis and the Y=0 origin at the top of the screen, and joystick event reporting mode for joystick 1, with both buttons being logically assigned to the mouse. After any joystick command, the ikbd assumes that joysticks are connected to both Joystick0

and Joystick1. Any mouse command (except MOUSE DISABLE) then causes port 0 to again be scanned as if it were a mouse, and both buttons are logically connected to it. If a mouse disable command is received while port 0 is presumed to be a mouse, the button is logically assigned to Joystick1 (until the mouse is reenabled by another mouse command).

H.8 ikbd Command Set

This section contains a list of commands that can be sent to the ikbd. Command codes (such as 0x00) which are not specified should perform no operation (NOPs).

H.8.1 Reset

```
0x80
0x01
```

N.B. The RESET command is the only two byte command understood by the ikbd.

Any byte following an 0x80 command byte other than 0x01 is ignored (and causes the 0x80 to be ignored).

A reset may also be caused by sending a break lasting at least 200ms to the ikbd.

Executing the RESET command returns the keyboard to its default (power-up) mode and parameter settings. It does not affect the time-of-day clock.

The RESET command or function causes the ikbd to perform a simple self-test.

If the test is successful, the ikbd will send the code of 0xF0 within 300mS of receipt of the RESET command (or the end of the break, or power-up). The ikbd will then scan the key matrix for any stuck (closed) keys. Any keys found closed will cause the break scan code to be generated (the break code arriving without being preceded by the make code is a flag for a key matrix error).

H.8.2 Set Mouse Button Action

```
0x07
%00000mss ; mouse button action
          ;   (m is presumed = 1 when in MOUSE KEYCODE mode)
          ; mss=0xy, mouse button press or release causes mouse
          ;   position report
          ;   where y=1, mouse key press causes absolute report
          ;   and x=1, mouse key release causes absolute report
          ; mss=100, mouse buttons act like keys
```

This command sets how the ikbd should treat the buttons on the mouse. The default mouse button action mode is %00000000, the buttons are treated as part of the mouse logically.

When buttons act like keys, LEFT=0x74 & RIGHT=0x75.

H.8.3 Set Relative Mouse Position Reporting

```
0x08
```

Set relative mouse position reporting. (DEFAULT) Mouse position packets are generated asynchronously by the ikbd whenever motion exceeds the setable threshold in either axis (see SET MOUSE THRESHOLD). Depending upon the mouse key mode, mouse position reports may also be generated when either mouse button is pressed or released. Otherwise the mouse buttons behave as if they were keyboard keys.

H.8.4 Set Absolute Mouse Positioning

```
0x09
XMSB      ; X maximum (in scaled mouse clicks)
XLSB
YMSB      ; Y maximum (in scaled mouse clicks)
YLSB
```

Set absolute mouse position maintenance. Resets the ikbd maintained X and Y coordinates. In this mode, the value of the internally maintained coordinates does NOT wrap between 0 and large positive numbers. Excess motion below 0 is ignored. The command sets the maximum positive value that can be attained in the scaled coordinate system. Motion beyond that value is also ignored.

H.8.5 Set Mouse Keycode Mode

```
0x0A
deltax          ; distance in X clicks to return
                (LEFT) or (RIGHT)
deltay          ; distance in Y clicks to return (UP)
                or (DOWN)
```

Set mouse monitoring routines to return cursor motion keycodes instead of either RELATIVE or ABSOLUTE motion records. The ikbd returns the appropriate cursor keycode after mouse travel exceeding the user specified deltas in either axis. When the keyboard is in key scan code mode, mouse motion will cause the make code immediately followed by the break code. Note that this command is not affected by the mouse motion origin.

H.8.6 Set Mouse Threshold

```
0x0B
X               ; x threshold in mouse ticks
                (positive integers)
Y               ; y threshold in mouse ticks
                (positive integers)
```

This command sets the threshold before a mouse event is generated. Note that it does NOT affect the resolution of the data returned to the host. This command is valid only in RELATIVE MOUSE POSITIONING mode. The thresholds default to 1 at RESET (or power-up).

H.8.7 Set Mouse Scale

```
0x0C
X               ; horizontal mouse ticks per internel
                X
Y               ; vertical mouse ticks per internel Y
```

This command sets the scale factor for the ABSOLUTE MOUSE POSITIONING mode.

In this mode, the specified number of mouse phase changes ('clicks') must occur before the internally maintained coordinate is changed by one (independently scaled for each axis). Remember that the mouse position information is available only by interrogating the ikbd in the ABSOLUTE MOUSE POSITIONING mode unless the ikbd has been commanded to report on button press or release (see SET MOSE BUTTON ACTION).

H.8.8 Interrogate Mouse Position

```
0x0D
Returns:
        0xF7            ; absolute mouse position header
BUTTONS
        0000dcba        ; where a is right sbutton down since
                        last interrogation
                        ; b is right button up since last
                        ; c is left button down since last
                        ; d is left button up since last
        XMSB            ; X coordinate
        XLSB
        YMSB            ; Y coordinate
        YLSB
```

The INTERROGATE MOUSE POSITION command is valid when in the ABSOLUTE MOUSE POSITIONING mode, regardless of the setting of the MOUSE BUTTON ACTION.

H.8.9 Load Mouse Position

```
0x0E
```

```
0x00            ; filler
XMSB            ; X coordinate
XLSB            ; (in scaled coordinate system)
YMSB            ; Y coordinate
YLSB
```

This command allows the user to preset the internally maintained absolute mouse position.

H.8.10 Set Y=0 At Bottom

```
0x0F
```

This command makes the origin of the Y axis to be at the bottom of the logical coordinate system internel to the ikbd for all relative or absolute mouse motion. This causes mouse motion toward the user to be negative in sign and away from the user to be positive.

H.8.11 Set Y=0 At Top

```
0x10
```

Makes the origin of the Y axis to be at the top of the logical coordinate system within the ikbd for all relative or absolute mouse motion. (DEFAULT) This causes mouse motion toward the user to be positive in sign and away from the user to be negative.

H.8.12 Resume

```
0x11
```

Resume sending data to the host. Since any command received by the ikbd after its output has been paused also causes an implicit RESUME this command can be thought of as a NO OPERATION command. If this command is received by the ikbd and it is not PAUSED, it is simply ignored.

H.8.13 Disable Mouse

```
0x12
```

All mouse event reporting is disabled (and scanning may be internally disabled). Any valid mouse mode command resumes mouse motion monitoring. (The valid mouse mode commands are SET RELATIVE MOUSE POSITION REPORTING, SET ABSOLUTE MOUSE POSITIONING, and SET MOUSE KEYCODE MODE.) N.B. If the mouse buttons have been commanded to act like keyboard keys, this command DOES affect their actions.

H.8.14 Pause Output

```
0x13
```

Stop sending data to the host until another valid command is received. Key matrix activity is still monitored and scan codes or ASCII characters enqueued (up to the maximum supported by the microcontroller) to be sent when the host allows the output to be resumed. If in the JOYSTICK EVENT REPORTING mode, joystick events are also queued.

Mouse motion should be accumulated while the output is paused. If the ikbd is in RELATIVE MOUSE POSITIONING REPORTING mode, motion is accumulated beyond the normal threshold limits to produce the minimum number of packets necessary for transmission when output is resumed. Pressing or releasing either mouse button causes any accumulated motion to be immediately queued as packets, if the mouse is in RELATIVE MOUSE POSITION REPORTING mode.

Because of the limitations of the microcontroller memory this command should be used sparingly, and the output should not be shut of for more than <tbd> milliseconds at a time.

The output is stopped only at the end of the current 'even'. If the PAUSE OUTPUT command is received in the middle of a multiple byte report, the packet will still be transmitted to conclusion and then the PAUSE will take effect.

When the ikbd is in either the JOYSTICK MONITORING mode or the FIRE BUTTON MONITORING mode, the PAUSE OUTPUT command also temporarily stops the monitoring process (i.e. the samples are not enqueued for transmission).

H.8.15 Set Joystick Event Reporting

```
0x14
```

Enter JOYSTICK EVENT REPORTING mode (DEFAULT). Each opening or closure of a joystick switch or trigger causes a joystick event record to be generated.

H.8.16 Set Joystick Interrogation Mode

```
0x15
```

Disables JOYSTICK EVENT REPORTING. Host must send individual JOYSTICK INTERROGATE commands to sense joystick state.

H.8.17 Joystick Interrogate

```
0x16
```

Return a record indicating the current state of the joysticks. This command is valid in either the JOYSTICK EVENT REPORTING mode or the JOYSTICK INTERROGATION MODE.

H.8.18 Set Joystick Monitoring

```
0x17
rate                        ; time between samples in hundreths
                            of a second
Returns: (in packets of two as long as in mode)
         %000000xy          ; where y is JOYSTICK1 Fire button
                            ; and x is JOYSTICK0 Fire button
         %nnnnmmmm          ; where m is JOYSTICK1 state
                            ; and n is JOYSTICK0 state
```

Sets the ikbd to do nothing but monitor the serial command lne, maintain the time-of-day clock, and monitor the joystick. The rate sets the interval between joystick samples.

N.B. The user should not set the rate higher than the serial communications channel will allow the 2 bytes packets to be transmitted.

H.8.19 Set Fire Button Monitoring

```
0x18
Returns: (as long as in mode)
         %bbbbbbbb          ; state of the JOYSTICK1 fire button
                            packed
                            ; 8 bits per byte, the first sample
                            if the MSB
```

Set the ikbd to do nothing but monitor the serial command line, maintain the time-of-day clock, and monitor the fire button on Joystick 1. The fire button is scanned at a rate that causes 8 samples to be made in the time it takes for the previous byte to be sent to the host (i.e. scan rate = 8/10 * baud rate).

The sample interval should be as constant as possible.

H.8.20 Set Joystick Keycode Mode

```
0x19
RX       ; length of time (in tenths of seconds) until
         ; horizontal velocity breakpoint is reached
```

```
RY      ; length of time (in tenths of seconds) until
        ; vertical velocity breakpoint is reached
TX      ; length (in tenths of seconds) of joystick closure
        ; until horizontal cursor key is generated before RX
        ; has elapsed
TY      ; length (in tenths of seconds) of joystick closure
        ; until vertical cursor key is generated before RY
        ; has elapsed
VX      ; length (in tenths of seconds) of joystick closure
        ; until horizontal cursor keystokes are generated
        ; after RX has elapsed
VY      ; length (in tenths of seconds) of joystick closure
        ; until vertical cursor keystokes are generated
        ; after RY has elapsed
```

In this mode, joystick 0 is scanned in a way that simulates cursor keystrokes.

On initial closure, a keystroke pair (make/break) is generated. Then up to Rn tenths of seconds later, keystroke pairs are generated every Tn tenths of seconds. After the Rn breakpoint is reached, keystroke pairs are generated every Vn tenths of seconds. This provides a velocity (auto-repeat) breakpoint feature.

Note that by setting RX and/or Ry to zero, the velocity feature can be disabled. The values of TX and TY then become meaningless, and the generation of cursor 'keystrokes' is set by VX and VY.

H.8.21 Disable Joysticks

```
0x1A
```

Disable the generation of any joystick events (and scanning may be internally disabled). Any valid joystick mode command resumes joystick monitoring. (The joystick mode commands are SET JOYSTICK EVENT REPORTING, SET JOYSTICK INTERROGATION MODE, SET JOYSTICK MONITORING, SET FIRE BUTTON MONITORING, and SET JOYSTICK KEYCODE MODE.)

H.8.22 Time-Of-Day Clock Set

```
0x1B
YY              ; year (2 least significant digits)
MM              ; month
DD              ; day
hh              ; hour
mm              ; minute
ss              ; second
```

All time-of-day data should be sent to the ikbd in packed BCD format.

Any digit that is not a valid BCD digit should be treated as a 'don't care' and not alter that particular field of the date or time. This permits setting only some subfields of the time-of-day clock.

H.8.23 Interrogate Time-Of-Day Clock

```
0x1C
Returns:
        0xFC            ; time-of-day event header
        YY              ; year (2 least significant digits)
        MM              ; month
        DD              ; day
        hh              ; hour
        mm              ; minute
        ss              ; second
```

All time-of-day is sent in packed BCD format.

H.8.24 Memory Load

```
0x20
ADRMSB            ; address in controller
ADRLSB            ; memory to be loaded
NUM               ; number of bytes (0-128)
{ data }
```

This command permits the host to load arbitrary values into the ikbd controller memory. The time between data bytes must be less than 20ms.

H.8.25 Memory Read

```
0x21
ADRMSB                    ; address in controller
ADRLSB                    ; memory to be read
Returns:
        0xF6              ; status header
        0x20              ; memory access
        { data }          ; 6 data bytes starting at ADR
```

This comand permits the host to read from the ikbd controller memory.

H.8.26 Controller Execute

```
0x22
ADRMSB            ; address of subroutine in
ADRLSB            ; controller memory to be called
```

This command allows the host to command the execution of a subroutine in the ikbd controller memory.

H.8.27 Status Inquiries

Status commands are formed by inclusively ORing 0x80 with the relevant SET command.

Example:

```
0x88 (or 0x89 or 0x8A)      ; request mouse mode
Returns:
        0xF6                ; status response header
        mode                ; 0x08 is RELATIVE
                            ; 0x09 is ABSOLUTE
                            ; 0x0A is KEYCODE
        param1              ; 0 is RELATIVE
                            ; XMSB maximum if ABSOLUTE
                            ; DELTA X is KEYCODE
        param2              ; 0 is RELATIVE
                            ; YMSB maximum if ABSOLUTE
                            ; DELTA Y is KEYCODE
        param3              ; 0 if RELATIVE
                            ; or KEYCODE
                            ; YMSB is ABSOLUTE
        param4              ; 0 if RELATIVE
                            ; or KEYCODE
                            ; YLSB is ABSOLUTE
        0                   ; pad
        0
```

The STATUS INQUIRY commands request the ikbd to return either the current mode or the parameters associated with a given command. All status reports are padded to form 8 byte long return packets. The responses to the status requests are designed so that the host may store them away (after stripping off the status report header byte) and later send them back as commands to ikbd to restore its state. The 0 pad bytes will be treated as NOPs by the ikbd.

Valid STATUS INQUIRY commands are:

```
0x87     mouse button action
0x88     mouse mode
0x89
0x8A
0x8B     mouse threshold
0x8C     mouse scale
0x8F     mouse vertical coordinates
0x90     (returns          0x0F Y=0 at bottom
                           0x10 Y=0 at top)
0x92     mouse enable/disable
         (returns          0x00 enabled)
                           0x12 disabled)
0x94     joystick mode
0x95
0x96
0x9A     joystick enable/disable
         (returns          0x00 enabled
                           0x1A disabled)
```

It is the (host) programmer's responsibility to have only one unanswered inquiry in process at a time.

STATUS INQUIRY commands are not valid if the ikbd is in JOYSTICK MONITORING mode or FIRE BUTTON MONITORING mode.

H.9 Scan Codes

The key scan codes return by the ikbd are chosen to simplify the implementation of GSX.

H.9.1 GSX Standard Keyboard Mapping

Hex	Keytop
01	Esc
02	1
03	2
04	3
05	4
06	5
07	6
08	7
09	8
0A	9
0B	0
0C	-
0D	==
0E	BS
0F	TAB
10	Q
11	W
12	E
13	R
14	T
15	Y
16	U
17	I
18	O
19	P
1A	[
1B	]
1C	RET
1D	CTRL
1E	A
1F	S
20	D
21	F
22	G
23	H
24	J
25	K
26	L
27	;
28	'
29	`
2A	(LEFT) SHIFT

2B	\
2C	Z
2D	X
2E	C
2F	V
30	B
31	N
32	M
33	,
34	.
35	/
36	(RIGHT) SHIFT
37	{ NOT USED }
38	ALT
39	SPACE BAR
3A	CAPS LOCK
3B	F1
3C	F2
3D	F3
3E	F4
3F	F5
40	F6
41	F7
42	F8
43	F9
44	F10
45	{ NOT USED }
46	{ NOT USED }
47	HOME
48	UP ARROW
49	{ NOT USED }
4A	KEYPAD -
4B	LEFT ARROW
4C	{ NOT USED }
4D	RIGHT ARROW
4E	KEYPAD +
4F	{ NOT USED }
50	DOWN ARROW
51	{ NOT USED }
52	INSERT
53	DEL
54	{ NOT USED }
5F	{ NOT USED }
60	ISO KEY
61	UNDO
62	HELP
63	KEYPAD (

64	KEYPAD /
65	KEYPAD *
66	KEYPAD *
67	KEYPAD 7
68	KEYPAD 8
69	KEYPAD 9
6A	KEYPAD 4
6B	KEYPAD 5
6C	KEYPAD 6
6D	KEYPAD 1
6E	KEYPAD 2
6F	KEYPAD 3
70	KEYPAD 0
71	KEYPAD .
72	KEYPAD ENTER