An Introduction to Programming C-64 Demos

Puterman

`puterman@civitas64.de`

aka Linus Åkerlund

Contents

Introduction

What is this Document About?

This is a document about coding demos on the C-64. With the additional help of some references, you should be able to learn how to code demos by reading this document and writing some code.

Why Did I Write This Document?

There were several things that got me started writing this document. First of all, I didn't know of any good tutorial about demo programming on the C-64. There's Coders World, of course, but it may not be thorough enough with the initial details. After all, the hardest problem is to get started at all.

Another reason was that people sometimes ask me things about coding, so in a way this is also some kind of an FAQ. Not that I've been asked about all these things, of course, I've filled in quite a lot of gaps.

The most important reason, though, is that I want people to learn to code demos on the C-64, and if the availability of a document like this can make someone start coding, that'd make me very happy. The C-64 scene is slowly fading into oblivion, with fewer and fewer releases, and I think what's missing the most is a new generation of coders who want to astonish people with amazing effects. And if you want to do that, you have to start somewhere, and that somewhere is, in my opinion, not with 8x8 plasmas, but with the kinds of effects that I go through in this tutorial.

I'm not claiming to be a good coder. Not even an average one. But I know some of the basics, and I'm trying to share that knowledge here.

Contributors

Thanks to BlackJack for reading through the text and suggesting changes.

Why C-64 Demos?

Before we start programming we should probably ask ourselves two questions:

- Why program demos?
- Why use the C-64?

Why Demos?

It's pretty obvious why you'd want to program demos; because it's fun. You get all the usual advantages of doing programming, that you're learning about programming and that it's fun. You also get the added advantage that the resulting programs are cool and nice to look at (if you're good at it). And of course, demos is more fun than most other things you can program.

Another good thing about programming demos is that you'll learn about many different aspects of the computer, at a very low level. To get good performance (which is very important in demos), you need to write efficient code, and the only way to do that is to bang directly on the hardware. You don't use any kind of libraries or abstractions when you program demos, you do it all the hard way. So you need to understand how you make graphics appear on the screen, music to be heard from the speakers and how to load stuff from disk.

You also need to use interrupts, which is something you really need to know about if you're ever going to write an operating system. It is also one of the things that is considered to be difficult to understand. Don't let that scare you, though, it's a pretty easy principle to understand, and as soon as you've gotten it to work once, you can probably do it much more easily the next time.

All in all, in coding demos you learn how a computer works at a very fundamental level, and understanding that makes lots of other things in programming easier. I'll just take one example: novice C programmers often have lots of problems with pointers. Well, if you've written some machine code programs you know what pointers are and that they're not dangerous[1].

Why C-64

The second question I posed above was why you'd want to program your demos on a C-64. There are numerous very good reasons why you should use a C-64, some of which are applicable to some other hardware platforms as well, and some which aren't. I'll list some of my favourite reasons here.

The CPU

The CPU in the C-64 is a 6510, which is a variant of the 6502. It's a very nice processor to program. It has a simple instruction set, so it's easy to get started. It has been studied a lot and there's lots of documentation available. It uses memory mapped I/O, which means that performing I/O (like showing stuff on the screen) is no different than putting a value at some arbitrary memory location.

Of course, the 6502 only has three registers (not counting the PC, SP and Status register), which can make it a pain in the ass to implement complicated algorithms on, but on the other hand, it sure makes you appreciate a modern RISC processor with dozens of general-purpose registers.

The VIC

The VIC (the graphics chip) is very nice. The VIC has a lot of interesting bugs (or features, depending on how you look at it), which means you can create amazing effects that are unique to the C-64. It's all about learning about the hardware and learning how to use it to fit your needs, not just blindly following a specification document.

Most of these effects are achieved through well-timed modifications of VIC registers. That's one of the things are most important in demo program: to use perfectly timed code. Creating raster bars on the Amiga is easy: just put some values in the copper list. On the C-64, on the other hand, you have to change $d020 and/or $d021 at exactly the right time, or you'll get flickering bars.

It's Old and Slow

That the C-64 is old and slow might not sound like much of an advantage, but when it comes to demo programming, it is. Why? It means that it's difficult to do stuff that requires lots of computations. You can't just throw CPU cycles at it, and write inefficient code. You have to think, to make it efficient. And doing really difficult things (or rather, impossible things) is what demo programming is all about. A good programmer can write a program that does something that seems very hard to do. A good demo programmer can write code that does something that is completely impossible.

That's why it's meaningless to start writing demos on a modern computer: it's too easy to make stuff that looks impressive, it's too easy to fool people into thinking that you've done something impressive, so you really don't have the right sort of motivation to actually write good code. Instead you're pushed in another direction, towards writing large programs, which will require that you use abstraction, which probably means that you'll use some high level language, and then you're not even close to programming demos.

High level languages are nice, they make it easy to create utility programs, and make them secure and robust. But if you're writing a program that displays graphics effects, that are written in a high level language, don't try to fool anyone that you're writing a demo. A demo is never written in a high level language. [2]

Why am I making such a strong point about this? Because a demo isn't a **program**, it's something else. In what way does a demo differ from a program? A demo has to take over the computer and bang directly on the hardware, it does not run under an operating system. This also means that it will have to take care of the things that the operating system would take care of, if you were writing a normal program, like interrupt handling and I/O.

Anyway, I shouldn't spend too much time on this, just believe me, it's much better to learn to program the C-64, before you move on to more modern[3] architectures. I'm not putting them down, a workstation is good for lots of things, and I spend quite a lot of my time writing code in high level languages, but you can never run a demo under a real operating system. Okay, let's move on to some actual programming.

Learning Machine Language

First of all, you need to learn 6502 assembly language. I'm not going to include a tutorial on assembler programming here. I'll just give a few examples and give pointers to some other good documents.

I'd suggest you do the following to learn assembler: get a tutorial, and try to grasp the general structure of it, learn some of the more important instructions, and then get a good reference, like ``Programmer's Reference Guide''. You shouldn't read the whole book, as most of it explains how to program the C-64 in BASIC. But the summary of the opcodes is nice, so that's a good reason to keep the book.

There's a pretty nice assembler tutorial in the first couple of issues of Commodore Hacking.

When you've learned some assembler, you need to know which memory addresses to poke values into to make interesting things happen. As the C-64 uses memory mapped I/O, you perform I/O operations just by putting values into memory locations, or reading from memory locations. What you need is a memory map, that tells you which addresses are special, and which ones are just RAM. ``Mapping the C-64'' is a nice and detailed memory map.

If you think you're really smart, you shouldn't even need a tutorial on assembler, all you need is a reference document and some examples. So let's look at some really simple examples.

Flashing the Border

A very simple effect, that shows how you can achieve some things on the C-64, is to make the border flash in different colours. Here's the code to do it[4]:

```
            * = $1000

loop:       inc $d020 ; increment $d020
            jmp loop  ; jump to label loop
```

That was a short program, wasn't it? If you type in and run this program[5], you will see the border flashing wildly. If you're using an assembler like Turbo Assembler or AssBlaster, you should just have to tap the RESTORE key to get back to the assembler.

Let's explain exactly what the program does. On the first line, we simply tell the assembler that the machine code it creates should start at memory address $1000 (1000 hexadecimal, which is the same as 4096 in the decimal number system). This is Turbo Assembler syntax. If you're using some other assembler, consult the documentation.

The second line has a label, `loop`, which is just a way to name that memory location, ie. the one where the instruction `inc $d020` starts. As we have told the assembler to assemble the code to address `$1000`, we know that what the label `loop` represents is actually `$1000`.

After the label comes an instruction, `inc $d020`, which tells means ``increment whatever value is at the address $d020 in place''. This is the same thing as loading the value from $d020 (using the instruction `lda $d020`), adding 1 to it (`adc #$01`) and then storing it in the same address (`sta $d020`). Why do we do this? Because the C-64 has memory mapped I/O, and the adress $d020 actually represents the border colour. So by putting different values in $d020, we change the colour of the border. So if it was initially black (value 0), it will be white (value 1) after executing the instruction `inc $d020`.

Then we come to the last line, which says `jmp loop`. When the program flow reaches this line, it jumps to the adress corresponding to the label `loop` and keeps executing instructions there. In our case, it will jump back to the previous line and execute `inc $d020` again. Then it will again reach the `jmp loop` line, go back, increment the border colour etc., forever. Or rather, until we press RESTORE, which will take us back to the assembler, or if we reset the computer or whatever.

So, now you know how to flash the border. If you use some other address than $d020, you can change other things. Some addresses will just give weird results, others will give sane results. If you use $d021 instead of $d020, the main screen will flash instead.

Clearing the Screen

Let's just take another simple example, that shows some other instructions, and actually does something useful. What we'll do is to clear the screen. That may not sound very interesting, but it's something you'll need to do at the beginning of most demo parts you write, so it's actually useful. And of course, after you have cleared the screen, you can just jump to some other routine, like eg. the one that flashes the border.

Here's the code to clear the screen:

```
            * = $1000

            lda #$00       ; Put the value 0 in accumulator
            sta $d020      ; Put value of acc in $d020
            sta $d021      ; Put value of acc in $d021
            tax            ; Put value of acc in x reg
            lda $20        ; Put the value $20 in acc
```

```
clrloop:    sta $0400,x    ; Put value of acc in $0400 + value in x reg
            sta $0500,x
            sta $0600,x
            sta $0700,x
            dex             ; Decrement value in x reg
            bne clrloop     ; If not zero, branch to clrloop
```

So, how does the program above clear the screen? The comments should explain most of what goes on, but here's an overview:

- We start by setting the border and background colours to black. The value 0 means black, and, as you probably remember from the last example, $d020 and $d021 are the addresses that control the colour of the border and the main screen.
- Now we want to remove all the characters from screen memory. Screen memory is by default located at $0400, and is $400 bytes long[6], so we need a loop to clear it all. We clear it by setting every character position to $20, which is the character code for the space character.
- Note how cleverly we handle the loop index, in the X register. We start by setting it to 0. Then we decrement for each iteration and then compare it so 0. The first time we decrement it, it will wrap around to $ff, and then it will go all the way down to 0 again.

If there's anything about this test program that you don't understand, look up the instructions and the various addressing modes in your 6502 assembler reference.

Note that this program won't behave very well if you just type it in and run it, because it doesn't really end. After the loop has finished, it will just happily get whatever is at the address after the loop and try to execute it as an instruction. That will probably not work very well, and something weird will happen. You will have to figure out yourself how to make it behave. One idea is to end this program by appending the border flashing code from the previous example.

Tools

Before we jump straight to demo programming, I need to introduce some tools. The necessary tools are a monitor and/or an assembler. You can choose to do your programming either in a monitor or an assembler. Which choice is the best is up to you. People have different opinions, but these days most people would probably say that an assembler is the best choice, because it makes programming more easy, as you can move stuff around, insert code into already existing code, add comments and use labels. On the other hand, if you program in a monitor, you know exactly where in memory your code is and you know where the page borders are, something which is really important when it comes to writing code that needs perfect timing.

Anyway, I'll try to describe how you can do some stuff with a monitor and with an assembler. All the example code in this text is in Turbo Assembler format, so maybe it's easiest for you to use Turbo Assembler to start with. The reason why I use Turbo Assembler format is that most people in the scene use Turbo Assembler, and I use it myself.

Using a Monitor

The monitor I'll describe here is a pretty standardized one, and the commands described should work on most monitors, like the ones in the Action Replay and Final Cartridge cartridges. There are some exceptions, like S-Mon, and if that's the only monitor you have, you'll have to find out how it works on your own.

To write new code, you type in a line like

```
.a 1000 lda #$00
```

What this does is that it assembles the instruction `lda #$00` and puts the resulting machine code at the address $1000. You can now simply type the next line of code, as the monitor itself should now give you a line that looks like this:

```
.a 1002
```

That is, the monitor itself calculates where you want to put the next instruction, so if you want to type in a whole program, you only have to issue the `a` command once yourself.

To start a program, you simply jump directly it, using the `g` command, eg.:

```
.g 1000
```

This will jump directly to the address $1000. To look at the code in memory, eg. the code you've already written, use the `d` command, like this:

```
.d 1000
```

This will disassemble the code in memory, starting at $1000. Different monitors will handle this differently, but most monitors should let you scroll through the code, using the cursor keys.

There are lots of more monitor commands, but this isn't a manual for machine code monitors, so I'll leave the rest to you.

Using an Assembler

Learning to use an assembler may be a bit more difficult than learning to use a monitor, but most people would probably think that it's easier to use an assembler than a monitor. When you've loaded and started Turbo Assembler (sys $9000), you end up in the editor, in which you write code. The syntax is just like the exemples in this text.

Most commands in Turbo Assembler are invoked by pressing ←, followed by some other key. To assemble a program, use ← - `3`. This will assemble the program, and if you press `s` just after the assembly, the program is started automatically. When you're in the program, you can get back to Turbo Assembler by pressing RESTORE[7]. If that doesn't work, you can reset and type `sys $9000`, which should bring you back, unless you've managed to overwrite the memory of Turbo Assembler.

Some other nice commands are:

- ← - 5 - assemble to object file on disk
- ← - s - save source to disk
- ← - l - load source from disk
- ← - 1 - exit to basic

The rest you'll have to figure out for yourself. There are documents available on the web about how to use Turbo Assembler. There are also lots of different versions floating around, some of which support macros and some of which support undocumented opcodes. You can try different versions to see which one you like best, but for now it shouldn't matter which version you use.

Cross Assemblers and Emulators

You can also use a cross assembler, which is a program that does the assembly for you on some other computer, like an Amiga or a PC, and then you can transfer the machine code to your C-64 and run it. I don't know too much about cross assemblers, so I won't go into that here.

You can also run your code in an emulator (preferably VICE), instead of on a real C-64. This isn't anything I'd recommend, as you can never be sure that the emulator will be completely compatible with the real thing.

Demo Programming

Finally, we get to the interesting stuff, ie. demo programming. Now, there are a lot of things you need to know about if you're going to write a demo, so we'll have to decide where to start. A demo consists of graphics, sound and code. Of course, in this document we'll concentrate on producing the code, but you'll still need to know how to display graphics and play sounds.

The sound in most demos is music, and playing music produced with some music editor, like DMC or JCH's editor, is very easy. You don't really need to make the music yourself either, you can always use someone else's music, as long as you give credit to the person who did the music. I'll show you what to do to play a tune in an example below, but first there are some basic concepts we need to cover, like synchronizing things with the screen refresh.

As playing music is so easy (unless you're going to do something really fancy), we'll concentrate on coding graphics effects, because that's really everything you need in a demo.

Of course, if you make a multi-part demo that can't be squeezed into memory, you'll need to load parts from disk. You don't have to write a loader yourself, though, as there are some very good loaders around already.

$d012

Cryptic header, right? $d012 might be the most important address of them all, when it comes to demo programming on the C-64. $d012 has two different functions:

- When read, it returns the number of the current raster line.
- When written, it is used to set the number of the line where the next raster interrupt will occur.

We'll get back to raster interrupts later. You need to know about $d012 to understand them, so pay attention to the stuff in this section! The first item above is interesting, but it may not be obvious why it is interesting.

The current raster line is the line that is currently being redrawn on your screen. The whole screen is redrawn 50 times per second[8]. Each time it is redrawn from top to bottom, from the left to the right. So, if you want something to happen 50 times per second, all you have to do is to check the current value of $d012, and when it reaches a certain value, call the routine that performs the desired task. When finished, go back to checking $d012.

Now, there are 318 raster lines on a PAL machine, and one register can only store values between 0 and 255, so you need another bit to represent all 318 lines. That bit is bit 7 of $d011. So, bit 7 of $d011 is really bit 8 of $d012. Does that sound confusing? In that case, read this paragraph a couple of more times. So, if bit 7 on $d011 is set, $d012 represents raster lines greater than 255 ($ff), otherwise the lines between 0 and 255.

If you want to take the easy way out, and you don't need to synchronize to some specific part of the screen, use a $d012 value greater than $40, because $ff + $40 = $13f = 319. So you'll never get a value greater than $3f if bit 7 of $d011 is set. I'll show how you can use $d012 with an example:

```
           * = $0801

           lda #$00
           tax
           tay
           jsr $1000 ; initialize music

mainloop:  lda $d012    ; load $d012
           cmp #$80     ; is it equal to #$80?
           bne mainloop ; if not, keep checking

           inc $d020    ; inc border colour
           jsr $1003    ; jump to music play routine
```

```
        dec $d020    ; dec border colour
        jmp mainloop ; keep looping
```

Now, before you try this program out, you'll need to have a tune loaded into memory at $1000. Most tunes used in demos, intros and what have you are located at $1000, so you just need to rip a tune, or get it some other way. The code before the `mainloop` just initializes the tune, ie. sets all registers to 0, and then jumps to the initialization routine. You don't have to know what the music routine does, but what it actually does is that it resets the registers of the SID chip.

After the initialization we move directly into the main loop. What the main loop does is that it loads the value from $d012, compares it to $80 (128 decimal, somewhere in the middle of the screen), and if it's not $80, it jumps back to where it began, ie. to loading the value from $d012.

Now, as soon as the raster beam reaches line $80, the inner loop is exited, and we end up at the line that says `inc $d020`. What we do is that we increment the border colour, jump to the music play routine, which executes and returns, and then we decrement the border colour and jump back to the main loop. Why do we change the border colour? Because it's a way to visualize how much time the music player routine takes. You'll actually not only see how much time it takes (in raster lines), but also where on the screen you're playing it, ie. where the raster beam is when you're in the music player routine.

In this example we've learned two things:

- how to use $d012 to synchronize things with the screen refresh, and
- how to play music.

We will later show that this music player routine is in no way perfect, but it's a good illustration of some concepts, and it works pretty well. You should now be able to figure out for yourself how to do raster bars. The trick is that you wait for a certain raster line, then change the background colour (if you're on the normal screen, you have to change both $d020 and $d021, if you're in the upper or lower border, you only have to change $d020). You need pretty good timing to do that, but with a hardcoded delay between the colour changes, it should work well with code similar to the above example.

Graphics

Now it's time to move on to some more interesting stuff, making things happen on the screen. Of course, things have been happening with the screen in all the above examples, but not in a very controlled way.

To get things to happen on the screen, you need to know which VIC registers to put which values into. There's also the screen memory, which is normally located at $0400, and the colour memory, which is located at $d800, and works like the screen memory, but doesn't contain character codes, but colour values of the characters on the screen. But the VIC is the most important thing, so we'll start with some VIC basics.

Sprites

Sprite Basics

Sprites can be used for lots of cool things. What's nice about them is that they can be moved around on the screen, without affecting the background. As you probably know already, the C-64 has 8 hardware sprites, so you can have 8 things moving around simultaneously, or synchronize their movement to make bigger objects move around the screen.

You probably know that there are ways to get more than 8 sprites on the screen, but we'll go through the basics before we reveal that trick. The basics of sprites consist of the following:

- displaying them,
- setting their position,

- changing their shape and
- changing their colour.

There are a number of things you need to do in order to make a sprite appear on the screen. You need to point it at a chunk of data, that defines what it should look like. Then you need to set the x and y coordinates. Of course, you probably also want to set the colour of the sprite. And, last but not least, you need to turn the sprite on. This is all the stuff that is listed above, and this is everything you need to be able to do to do some basic sprite effects, like moving them around on the screen, animating them, etc.

Sprite Pointers

The sprite pointers, ie. the adresses that contain the pointers to where the graphics data that is displayed on the screen as sprites, are located after the end of the screen memory, which is usually located at $0400. The screen memory takes up exactly 1000 bytes (40 times 25 characters), so the last 24 bytes of the memory from $0400 to $0800 are free. The last 8 ones of these are the sprite pointers, and the 16 bytes before the sprite pointers aren't used at all, unless I'm completely mistaken.

So, the byte at $07f8 is the sprite pointer for the first sprite. Now, how do we fit a pointer into one byte? To represent an address in memory, we need two bytes. But sprite pointers are handled in a special way. First of all, you only point to sprite data that is located in the same VIC bank as the one where you keep the screen memory. There are four VIC banks, $0000-$3fff is the first one, the one that's used by default. A sprite takes up 63 bytes of data, but have to be aligned to 64 bytes, so you can put sprite data starting at adresses $0000, $0040, $0080, $00c0, $0100 etc.

So, how many sprites can you fit into one VIC bank? 256. And how many numbers can a byte represent? 256. Aha! Now we can understand how the sprite pointers work: if you set a sprite pointer to 0, it will point to the address $0000, in the current VIC bank. If you set it to 1, it will point to $0040. If you set it to $80, it will point to $2000, and if you set it to $c0, it will point to $3000. Easy, isn't it? So, let's say you want to do a simple animation, consisting of 8 sprites, with the sprite data placed at $2000 (up to $21ff). You start by putting the value $80 into $07f8, then change it to $81, then $82, and so on up to $87, then you'd start over from $80.

Important Addresses

Now we move on to how you set coordinates and colours and turn on sprites. Turning the sprites on is simple. The sprites are numbered from 0 up to 7. The sprite whose pointer you set when you poke values into $07f8 is 0, the one at $07f9 is 1, and so on. You turn on the sprites by setting the corresponding bits in $d015. So, to turn on sprite number 0, you poke the value 1 into $d015. To turn on sprites 0 and 1, put 3 in $d015.

Setting the coordinates of the sprites are equally simple, with one small catch. The x coordinate registers are $d000 (sprite 0), $d002 (sprite 1) and so on, and the y coordinate registers are $d001, $d003 etc. As these registers can only store one byte each, you can only set coordinates between 0 and 255. But the visible portion of the screen is 320 pixels wide! The solution is that there's an 8th bit for each x register in $d010. So, if bit 0 in $d010 is set, sprite 0's x coordinate is 256, plus the value in $d000. This might sound a bit complicated, but you'll get used to it pretty quickly.

Setting the colours is the easiest part. For single colour sprites, you just poke the colour value of each sprite into the addresses $d027, $d028 etc. Read your favourite VIC reference for information on other stuff that you can do with sprites...

Sprite Example

Here's an example that moves a sprite around on the screen:

```
* = $0801

lda #$01
sta $d015    ; Turn sprite 0 on
```

```
            sta $d027    ; Make it white
            lda #$40
            sta $d000    ; set x coordinate to 40
            sta $d001    ; set y coordinate to 40
            lda #$80
            sta $07f8    ; set pointer: sprite data at $2000

mainloop
            lda $d012
            cmp #$ff     ; raster beam at line $ff?
            bne mainloop ; no: go to mainloop

            lda dir      ; in which direction are we moving?
            beq down     ; if 0, down

                         ; moving up
            ldx coord    ; get coord
            dex          ; decrement it
            stx coord    ; store it
            stx $d000    ; set sprite coords
            stx $d001
            cpx #$40     ; if it's not equal to $40...
            bne mainloop ; just go back to the mainloop

            lda #$00     ; otherwise, change direction
            sta dir
            jmp mainloop

down
            ldx coord    ; this should be familiar
            inx
            stx coord
            stx $d000
            stx $d001
            cpx #$e0
            bne mainloop

            lda #$01
            sta dir
            jmp mainloop

coord
            .byte $40   ; current x and y coordinate
dir
            .byte 0     ; direction: 0 = down-right, 1 = up-left
```

That's a bit more code than in the previous examples, but it shouldn't be hard to understand. Just read the code and comments and you should be able to understand it.

If you type in and run the example program (you should), you'll notice that the movement isn't very smooth. It's not like anyone would accept a jerky movement like that in a real demo, so you'll have to fix it. The question to ask is why it's moving so jerky. The thing is that we haven't turned off timer interrupts. This means that sometimes we'll be in the kernal interrupt handler while we're passing raster line $ff, which means we'll miss it, and the sprite movement won't be updated until the next frame. You'll learn more about interrupts later.

A good exercise, to get your hands dirty with some more interesting sprite programming, would be to display several sprites at once, and make them move according to a sine table. With the help of your friend $d012, you should also be able to make a multiplexer, if you're really ambitious. You've got all the tools, now all you have to do is to think it up and express it in code.

Character Graphics

Sprites isn't the only way to display graphics. Our next topic is character graphics. You might have tried to draw something using the built-in character set. That's not very easy, and the results usually aren't all that good. Of course, you don't have to use the built-in character set. Using a character set editor, you can draw your own character set, and more effects than you can imagine are implemented using character graphics. They don't even have to look like characters, if the trick is (as it often is) to update the character set in realtime.

Before we get into those kinds of tricks, we'll do what we usually do and go through the basics first.

Screen and Colour RAM

As you should already know (from reading the section on learning ML), you change characters on the screen by putting the character values into screen memory. Screen memory is $03e8 bytes long and is located at $0400, unless you move it somewhere else. The screen codes aren't the same as the PETSCII codes you use for normal text, so if you want to prepare text that should be put on the screen, make sure you save it as screen codes. Most demo related text editors and such already do that, so there's nothing to worry about.

Now, putting characters on the screen is cool, but it gets even cooler if you can give each character its own colour. You can do this by poking the colour values into colour RAM, which is always located at $d800 (you can't move it).

So, let's say you want the letter A to appear in the upper left corner of the screen, and you want it to be yellow. What do you do? You poke the screen code for A into screen memory, ie. put the value 1 into memory position $0400, and put the value for yellow, which is 7, in $d800.

Changing Character Set

The built-in character set isn't so hot, so if you're going to display text, you should usually load your own character set. Fortunately, this is very easy, and you can control the locations of both the character set and the screen memory with one VIC register: $d018. You put the upper parts of two addresses into this register: the four most significant bits of the 14 bit[9] address of the screen memory in the four high bits, and the four most significant bits of the 14 bit address of the character set in the lower four bits. Confused? Let's take an example...

Screen memory is $0400 bytes long, and it has to start on an address that is a multiple of $0400, so it can start at $0000, $0400, $0800, $0c00 etc. As I said, it's located at $0400 by default. A character set is $0800 bytes long, and has to start on an address that is a multiple of $0800, so it can be located at $0000, $0800, $1000, $1800 etc. Now, let's say we want screen memory to be where it usually is, at $0400, and we have a character set at $2000. We want the four most significant bits of the 14 bit address $0400, which is 1 (in binary, $0400 is 0000 0100 0000 0000, and if you count 14 bits from the right, you get 00 0100 0000 0000, of which 0001 are the four most significant bits). We also want the four most significant bits of the 14 bit address $2000 (the character set). In binary, $2000 is 0010 0000 0000 0000, which means the four most significant bits of the 14 bit address is 1000, which is 8.

So, what do we actually store in $d018? We want 1 in the four most significant bits and 8 in the four least significant bits, ie. $18.

Now, I may have made this a bit too complicated. It's really not that difficult. Here's a trick: to find out which value to store in $d018, just count the number of $0400 chunks until the start of the address you're interested in. If your screen memory should start at $1800, you count $0000, $0400, $0800, $0c00, $1000, $1400. That's 6, so you use 6 in the upper four bits of $d018. Use the same values for the character set, but put it in the lower four bits of $d018.

Altering the Character Set in Realtime

I mention above that you can make interesting effects by altering the character set in realtime. How do you do this? Well, if you've specified that you want your character set at $2000, you can start poking values in the addresses from $2000 to $2800, and see what happens. As you're altering the character set that's used for displaying what's on the screen, the things on the screen will change immediately. You don't have to change anything in screen memory, just in the character set.

A classical example of an effect that you can do by poking data into a character set is a DYCP (Different Y Character Positions). A DYCP is a scroll text where the letters move up and down, independent of each other. You can figure out how to do it yourself, or read the article about it in Commodore Hacking.

An easier exercise than writing a DYCP would be to create some animated graphics by changing the character data. You can also use this to scroll text without using $d016, by ROLing the char data through character memory...

Here's a simple example program, that doesn't really do much, except illustrate what I've just written. It sets the position of the character set to $2000, and then alters it in realtime. Note that I'm using $d012, as always.

```
           * = $0801

           lda #$00     ; black
           sta $d020    ; border background colour
           sta $d021    ; screen background colour

           tax          ; set X to 0 too

clrscreen               ; set all char codes to 0
           sta $0400,x  ; on the screen
           sta $0500,x
           sta $0600,x
           sta $0700,x
           sta $2000,x  ; and set charset data to 0
           dex
           bne clrscreen

           lda #$18     ; screen at $0400, chars at $2000
           sta $d018

mainloop
           lda $d012
           cmp #$ff     ; on raster line $ff?
           bne mainloop ; no, go to mainloop

           ldx counter  ; get offset value
           inx          ; increment it
           cpx #$28     ; if it's $28, start over
           bne juststx
           ldx #$00
juststx
           stx counter

           lda $2000,x  ; get byte nr x from chardata
           eor #$ff     ; invert it
           sta $2000,x  ; store it back

           jmp mainloop ; keep going...

counter
           .byte 8      ; initial value for counter
```

Some small things in the example might have to be explained. The reason why I'm counting to $28, and not just 8 (which would be sufficient for inverting every byte that make up the character 0), is so that it won't be too fast to see what's going on. The reason why we're poking stuff into $2000 and up is that we're using character number 0. The data for char 0 starts at $2000, the data for char 1 starts at $2008, the data for char 2 at $2010 and so on.

Okay, that's all I'm going to say about character graphics right now. You know the basics, now all you have to do is to experiment and think up cool things to do with character graphics.

Note that I haven't mentioned multicolour character graphics. Using multicolour, you can get your characters to have 3 colours each, one of which is set in colour RAM, and two which are set in $d022 and $d023. The colours in colour RAM are of course different for every character, but $d022 and $d023 can't be changed for each character.

Bitmap Graphics

Character sets are good for a lot of things, but if you want a picture that covers the whole screen, a character set won't do, because you can't set every pixel on the screen, using only $0800 bytes. There are 320 x 200 pixels on the screen in hires mode, and 160 x 200 in colour mode, so you need about $2000 bytes to have total control of the graphics on the screen.

However, if the picture you want to display is ``simple'' enough, ie. if large parts of it are empty, or if it's highly repetitive, so that it could be drawn with a single customized character set, you can of course just use a charset to display it, and you'll save yourself $1800 bytes of memory. There are tools to convert a bitmap to a character set + a screen memory, which of course requires that the information contained in the bitmap will fit into a charset.

If you've understood how character graphics work, you shouldn't have any problems understanding how to display bitmap graphics. I'll describe how you can display hires graphics and multicolour graphics in the Koala Paint format.

The Concepts of Bitmap Graphics

When you display character graphics, you have to set $d018 to point to screen RAM and character set. If you want to display a bitmap picture, you have to do almost the same thing. You have to set the location of screen RAM (which is used for two sets of colours for each character (yes, in a way you're still using character)) and of the bitmap. The bitmap takes up almost $2000 bytes, and it has to be properly aligned, so you can't put a bitmap at eg. $2800, it has to be at $2000, $4000, $6000 etc.

Before we go into the theory of how you display a bitmap, I'll give you an example, and then I'll explain what it does. The example requires that you have a Koala picture loaded at $2000. Koala pictures are usually loaded to $6000, so go into your monitor and type `l ``filename'',08,2000`. That will load your picture to $2000.

```
           * = $0801

           lda #$00
           sta $d020
           sta $d021    ; set border and screen colour to black

           tax
copyloop:
           lda $3f40,x  ; copy colours to screen RAM
           sta $0400,x
           lda $4040,x
           sta $0500,x
           lda $4140,x
           sta $0600,x
           lda $4240,x
           sta $0700,x
           lda $4328,x  ; copy colours to colour RAM
           sta $d800,x
           lda $4428,x
           sta $d800,x
           lda $4528,x
           sta $d800,x
           lda $4628,x
           sta $d800,x
           dex
           bne copyloop
```

```
            lda #$3b      ; bitmap mode
            ldx #$18      ; multi-colour mode
            ldy #$18      ; screen at $0400, bitmap at $2000
            sta $d011
            stx $d016
            sty $d018

mainloop:
            jmp mainloop ; keep going...
```

As I still haven't explained everything about bitmap mode, this might not be all that clear, but it's still a very small example program and it'll display a Koala picture just fine.

We start by making the border and background black. This isn't actually accurate, as the background colour is stored in the Koala format, so we really should have loaded the value from $4711 and put it in $d021, but that's not really important.

Next we start copying stuff. Why do we do that. Well, we'll keep using $0400 as screen RAM, so we'll have to copy the screen RAM, which, in the Koala format, is located at $3f40, to $0400. That's easy, you've seen that one before. The same goes for colour RAM, which starts at $4328, which will have to be copied to $d800.

When we have all the data in the right places, all we need to do is to tell the VIC chip what mode to go into, and where it should read data. We tell it to go into bitmap mode by setting $d011 to $3b, and by setting $d016 to $18 we turn on multi-colour mode. And then we tell it to read screen RAM at $0400 and the bitmap at $2000, by setting $d018 to $18.

The last line just loops forever, so that you get a chance to watch the picture. Hopefully you've found a nice one. :-)

If you don't know where to find a Koala picture, get a program like Amica Paint, Koala Paint or Color-X16, and draw your own pictures. You can of course also rip pictures from other people's productions, but please, don't use ripped graphics in your demos, that's lame.

The Screen Layout

The layout of the actual bitmap is a bit weird, especially if you're used to other architectures, which have nice, linear bitmaps [10]. A C-64 bitmap isn't linearly adressable, ie. you can't just count the pixels from left to right and index into it in an easy way.

The data in a C-64 bitmap are indexed very much like a normal charset. The first byte (in the above example $2000) is the top, leftmost byte. $2001 is the byte below that one, $2002 below that one, and so on, until $2008, which is the one to the right of $2000. So, basically, it's as if you'd read one character from a charset, and put that character in the top left corner of the screen, and then you read the next character and put next to that one, and so on, until you reach character 40, which is below character 0.

It's not really hard to understand, but it does make it a bit difficult to implement efficient bitmap effects, like plotters. Think about it!

Displaying Bitmaps

Displaying a single-colour bitmap is just like displaying a multi-colour bitmap, except that you don't use colour-RAM. Why? Because all you need is a background colour and a foreground colour, and you can use the good old screen RAM for that. For each byte in the screen RAM, you set the upper nybble to one colour value, and the lower nybble to another value (or the same value, but then you won't see anything of the bitmap).

The last paragraph might sound a bit cryptic, but just try it out. Get yourself a nice hires bitmap (use some utility to create one, it should output a file that is 33 blocks long). Load that bitmap to some $2000 aligned address (like $2000), set $d011 to $3b, $d016 to $08 (single colour mode) and $d018 to $18. Then you can start poking values into $0400 etc., to see what happens.

Does that sound too simple? It is simple.

Interrupts

Okay, so far we've learnt a lot of useful things for putting graphics on the screen (and playing music, but that's almost too easy to mention). To sum it all up, we have all the tools we need to write programs that display graphics and play music. Multimedia programs, or something like that. We don't know everything we need to write demos, though. Demos, as I said in the introduction, are quite different from programs that display graphics and play music.

All the example programs above have suffered from one great problem: they have been badly timed, and all movement (as well as the playback of the music) has been jerky and inexact. We need to do something about that. You can't have a scroll text that doesn't move smoothly, or jittering raster bars, because that's the definition of lameness.

So, how do we do something about it? Well, we need to know a bit about interrupts. Of course, as a demo coder, you really need to know *everything* about interrupts, but we're not going to be that complete here. We're going to be complete enough so that you can easily learn the rest by reading your reference material and poking around in your computer yourself, though. Wizardry isn't about knowing *most* of what there is to know, it's about being able to setup interrupt handlers after 20 beers (trust me, that's not easy :-).

Why Interrupts

I suppose we'll have to answer this question, although it might seem stupid. There really isn't anything that keeps you from writing demos that don't use interrupts. If you just know how to turn of the CIA interrupts, you're safe from all the jittering and flickering we've experienced so far. But there are more things you can do with the help of interrupts, and you really need to know about them in order to have control over your computer. And demos are all about control. You'll also need to know about interrupts to be able to use an IRQ-loader, and if you're planning to ever write a demo that's bigger than what will fit in your computer's memory, you'll need to use a loader. And there's no excuse for not using an IRQ-loader these days.

In this text, we'll concentrate on raster interrupts, because that's what you'll use most often in demos. We'll also go into NMIs a bit, because it's very useful to know about them. And of course, if you've written a demo that looks great and that people love, it can feel pretty embarrasing if you can kill it just by pressing RUN/STOP + RESTORE...

What is an Interrupt?

A fair question to ask at this stage is what an interrupt is. They're a very simple, and very efficient way of handling some things. They work pretty much like an alarm clock.

Let's say you're going to bed after a long night of demo coding, and you have to get up at eight in the morning, to go to school or work. What do you do? You set your alarm clock to ring at eight, and then you can safely go to sleep, without having to worry, knowing that the clock will wake you up at eight. Until then, you can sleep all you want.

That's exactly how interrupts work: you set up an interrupt to wake up the CPU at some time and tell it to do something special, but until then the CPU can do its normal things. So, what you do when you set up an interrupt is to define *when* the interrupt should happen and *what* should happen when the interrupt is triggered.

On the C-64, you have some different types of interrupts to watch out for. You have two CIAs, which can both generate timer interrupts (tell the CIA to give you an interrupt after X clock cycles), you have the VIC, which can generate raster interrupts (generate an interrupt when we reach a certain raster line), sprite collision interrupts (generate an interrupt when two sprites collide) etc.

How to Implement Interrupts

Yeah, so how do you set up a raster interrupt? It's actually very easy, so I'll just give you some example code, and then explain it.

```
           * = $1000

           sei          ; turn off interrupts
           lda #$7f
           ldx #$01
           sta $dc0d    ; Turn off CIA 1 interrupts
           sta $dd0d    ; Turn off CIA 2 interrupts
           stx $d01a    ; Turn on raster interrupts

           lda #$1b
           ldx #$08
           ldy #$14
           sta $d011    ; Clear high bit of $d012, set text mode
           stx $d016    ; single-colour
           sty $d018    ; screen at $0400, charset at $2000

           lda #<int    ; low part of address of interrupt handler code
           ldx #>int    ; high part of address of interrupt handler code
           ldy #$80     ; line to trigger interrupt
           sta $0314    ; store in interrupt vector
           stx $0315
           sty $d012

           lda $dc0d    ; ACK CIA 1 interrupts
           lda $dd0d    ; ACK CIA 2 interrupts
           asl $d019    ; ACK VIC interrupts
           cli

loop:
           jmp loop     ; infinite loop

int:
           inc $d020    ; flash border

           asl $d019    ; ACK interrupt (to re-enable it)
           pla
           tay
           pla
           tax
           pla
           rti          ; return from interrupt
```

Now, if you're not used to using interrupts, that might look a bit weird. It works, though, and I'm now going to tell you how. (Well, I'll try, anyway. :-)

We start by turning off all interrupts, with an `SEI` instruction. The reason why we do this is because if we get an interrupt in the middle of setting up interrupts, things can get messed up and the program can crash. These things happen. Ever had a demo crash unpredictably? Sometimes it works, sometimes it doesn't? The interrupt setup might be screwed up. Or there can be some other reason. Read the code and see if you can find it. :-)

So, anyway, after the `SEI`, the action begins. We start by turning off all CIA interrupts. We do this by poking $7f into $dc0d and $dd0d. To find out what the value $7f means, read some reference. At a later stage, you might be interested in finding out what the individual bits of those registers do. We then turn on raster interrupts, by setting the lowest bit of $d01a.

The next thing we do is to setup $d011, $d016 and $d018. You should recognize those by now. We always have to initialize them, and I usually do it at the same time as the interrupt setup. The only bit that's relevant in this context (interrupts) is bit 7 of $d011, which is the 8th bit of $d012 (as we have more than 256 raster lines, but I've already talked about that above).

So, let's move on to the next chunk of instructions, which includes $d012. We set a new address in the interrupt vector, $0314. This address simply specifies where the processor should keep executing code when an interrupt occurs. We point it to our interrupt routine, `int`.

In the interrupt handler code, we don't do much. First we flash the border, by `INC`-inc $d020 (and of course, this happens once every frame), then we ACK the interrupt (set bit 0 of $d019, which I usually do with `ASL $d019`, for some reason (probably because that's how it was done in the example that I learnt about interrupts from). If we forget to do this, the interrupt will be triggered again, right after we return from this interrupt code. So we have to tell the VIC that we've handled this interrupt, so that it is cleared.

Then we have to restore the register contents from the stack (these are stored there when the interrupt occurs), and then execute the instruction `RTI`, which returns from the interrupt. Quite often you won't see exactly this code, but instead of the last six lines you'll see a `JMP` to $ea31 or $ea81. $ea31 is the default interrupt handler, which does this and that that you'll want if you're in BASIC, but you won't need that crap when you're coding demos, so skip it. $ea81 just contains the same code as the last six lines in our program, so exactly the same thing happens, except that we waste three clock cycles (for the `JMP`) and that we save 3 bytes of memory. So it's a trade-off, kind of, but it's not often 3 clock-cycles or 3 bytes will bother you very much, so who cares?

Of course you can't jump to ROM routines if you've switched out kernal ROM, so it's a good idea to use the code above anyway.

There's a modification you can do to the code above, which might be nice to try, and give you some further insight into how interrupts work. Instead of JMPing around in the infinite loop (`JMP loop`), do `RTS`. And add `JMP $ea31` instead of the last 6 lines. If you start the program from BASIC, you'll see that your interrupt routine starts executing, but you're still at the BASIC prompt, and can type in BASIC commands. Cool, huh?

In the rest of the text, I will be a bit more restrictive with explaining details. If you've come this far and understand most of it, and have tried typing in the programs and changing them a bit, as well as making your own routines, you should be able to cope with a bit less detailed explanations. If there's something you don't quite understand, consult your references, write some code and see what happens etc. And send me an email, telling me what you don't understand.

Playing music again

It should be trivial for you to alter the interrupt example program to play music. All you have to do is to init the music before the interrupt initialization, and then do a `JSR` to the play routine in the interrupt handler routine. If you can't remember how to do this, check the example on playing music (without interrupts) above.

Using More Than One Interrupt Per Frame

Now, pretty often (if you're coding real demos), you'll want to have several interrupts occur each frame. Let's say you have some kind of raster bar effect, that's created by an interrupt routine that is triggered at line $20, then you have a scroller that starts at line $e0. You can do this in two ways (or more, but this is just an example): (i) just use one raster interrupt, and do the old $d012 polling to wait for $e0, or (ii) use two interrupts.

There's nothing wrong with the first approach, except that you may want to do something in the main loop as well (outside of the interrupts), and in that case you don't want to sacrifice all the rastertime between the two effects by using them for polling $d012.

So, we'll go for the second approach. Let's say the label for the first interrupt routine is `intraster` and the one for the second one is `intscroll`. In the interrupt setup we'll put the address of `intraster` in the interrupt vector ($0314) and set the value $20 in $d012 (to make the VIC generate an interrupt at line $20).

That's all very well, but if we don't do anything more, the interrupt at $e0 will never happen. What we'll have to do is change the values of $0314, $0315 and $d012 in the interrupt routines. So, at the end of the first interrupt routine (`intscroll`) we set $0314 and $0315 to the address of `intscroll` and $d012 to $e0, before we return from the interrupt. And of course, in the second interrupt routine, we'll have to set the values back, so that they point to the first interrupt routine, and line $20.

If this sounds confusing and you're thinking ``No, I'll just keep polling $d012 instead'', then please read the above paragraphs again. It'll make your life so much easier.

It'd probably be a good idea to try to write a small program, using the ideas explained here. Write two interrupt routines that do something simple, like change the background colour, and check so that you can make them both work. To make it more interesting, change the values of $d012 each frame (by adding or subtracting one from the value). That way, you can make your rasterbars move up and down.

Stable Raster Interrupts

If you've tried out the above, especially changing the background colour on different lines, you've probably noticed that the interrupts aren't stable; if the colour change happens somewhere in the visible area of the screen, you'll see that it jitters a bit. It might even jump a whole line up and down, if you're on a Bad Line.

The reason is that when an interrupt is triggered, an instruction is executing in the CPU, and that instruction will have to finish before the jump to the interrupt handler takes place. As you can understand, there's a variable delay here, eg. a `NOP` only takes two cycles, and some memory manipulating instruction in an indexed addressing mode might take 7 cycles.

This can be pretty irritating, and it can be pretty difficult to time some routines so that they look nice. On the other hand, you can do pretty much, although your interrupts aren't stable. For now, it should be sufficient to know that there are ways to make the raster completely stable. You'll need a stable raster to do some advanced effects, like splits (raster bars that change colour in the middle of a line) and opening the side border. However, you can do stuff like opening the top and bottom border, sprite multiplexers, raster bars, scrollers, displaying FLI pictures, FLD and all kinds of stuff without having a stable raster.

Anyway, there are different ways to get a stable raster, and you'll want to learn it eventually, unless you want to be doomed to code simple effects or ugly-looking chunky crap effects (which sometimes don't seem to use any timing at all) for the rest of your life. These different techniques all have their fans, so I'll go through them quickly here:

- Double interrupts. This one uses two interrupts: the first one sets up the second one and makes it happen when executing a `NOP`, so that the jitter is reduced to one cycle, and then a simple branch instruction at exactly the right time, to remove the last cycle of jitter. This technique seems to be very popular.
- Sync with sprite. This technique utilizes the way sprite data is fetched. What you do is that you turn on at least one sprite, and at the right time, you execute some memory manipulating instruction, which is stalled by the sprite fetch, which makes it end at exactly the same time, a fixed number of cycles after the sprite data fetch is complete. This might sound complicated, but it's a lot simpler than the double interrupts method, and all you need to do is to turn on a sprite and do an `INC` at the right time.
- Triggering a Bad Line. This one might have some undesirable side effects, but it's very simple. All you do is trigger a Bad Line (eg. by executing `INC $d011`), and voila!, you have a stable interrupt. It might

produce some garbage on the screen, but it's really simple. I discovered this one while coding a FLI viewer, and this is pretty much the basis for FLI.

- Syncing with a Bad Line. I haven't tried this one myself, I just read about it in an IRC log I found somewhere on the web. Crossbow/Crest mentioned a method of getting a stable interrupt by using an `INC` at the right moment on a Bad Line. Try it out yourself.

My personal favorite is to sync with a sprite. I find it to be a very clean and nice way of getting a stable raster, without all the hassle of using the double interrupts method. Triggering a Bad Line probably isn't very good at all, but it simple and it works, although it can have side-effects that aren't very nice. Syncing with a Bad Line is probably a very good method, as Crossbow uses it. :-)

NMIs

I won't go very deep into NMIs here, I'll mostly explain how you can get rid of the problem with the RESTORE key. You can't just turn them off, because you can't do that (that's why they're called non-maskable interrupts). What you'll have to do to disable NMIs is the trigger an NMI and then refrain from ACKing it. That way, no new NMIs can occur. The simplest way to do this is to setup a timer interrupt on CIA2 (which is the CIA that can generate NMIs), and just do an `RTI` in the NMI handler routine.

You set up an NMI handler just like an interrupt handler, but instead of setting $0314-$0315, you set the address in $0318-$0319, which is the NMI vector. Just set it to point to a routine that returns, and does nothing more. It's as simple as that.

Interesting Effects

In the previous sections I have explained most of the basics that you need to know in order to code your own demos. You might not understand exactly how everything works, but if you've written some code, using the stuff I've explained above, you should have a fairly good grip on the basics.

However, although you now have most of the tools you need to create lots of effects, you might not have figured out exactly how they work. In this section I'll give some examples of cool effects. You can try implementing them yourself, or, even better, use the same concepts to implement something different and more original. But of course, there are some things that everyone who wants to code demos have to do at least once, and most of the below examples should be such things.

I won't give you source code to write all these things. I don't think that should be necessary. You'll also learn a lot by figuring our the details yourself. You can find source code and more details about some of these effects, and others as well, in early issues of Commodore Hacking and all three issues of Coders World (see the References section below).

I'll try to sort these effects in order of increasing complexity, so the first ones should be easy, and the later ones harder. But of course, that's individual. Anyway, text scrollers and raster bars should be easier than displaying a FLI picture or opening the side border.

Text Scroller

I'll start with this one, not just because it's a simple effect, but because it's so fundamental in demo coding. If you can do a simple text scroller, you can then work on making it more interesting, by adding colours, different sizes of fonts, animated fonts, Y-moving scrollers etc.

Doing the basic scroller is easy. Of course you start by setting up an interrupt that occurs a few lines before the actual text that should be scrolled. There you read a variable, where you've stored the current value of $d016, which is the X-scroll register. The three lower bits of $d016 control the X-scroll, so you have 8 steps of scrolling, enough to scroll one whole character. With the value 0 in the three lowest bits, you'll have your normal screen, and with the value 7, it will be scrolled 7 pixels to the right.

So, what you'll have to do is to start at the value 7, and then decrement it by 1 (or 2, 3, 4 etc., if you want a faster scroller) every frame. When you already have an X-scroll value of 0, you can't decrement it, you'll have to reset it to 7 again. At the same time you'll have to move the characters in screen memory one step to the left, and insert a new character in the character at the right end of the screen. Then you just keep going.

As I've said, a text scroller is a simple effect, but writing your first one isn't necessarily easy. So here are some suggestions from someone who can still remember the hassle of getting a simple text scroller to work:

- Start by putting some characters on the screen and make the $d016 fiddling work. The characters should move to the left for 8 frames and then flip back to their original position and start moving to the left again. This happens pretty fast, so it's not easy to see that everything is really correct, but if you get this effect, you're on the right way.
- When you've got that working, add a loop that copies characters from teh right to the left on one line of the screen, eg. the first one, which starts at $0400 and ends at $0427. That is, copy $0401 to $0400, $0402 to $0401 etc. And of course, add a new value in $0427 each time. You should now have a basic scroll.
- Add a routine that reads from an area of memory where you've stored your scroll text. Make sure it wraps at the end. You can do this by eg. storing the memory position of the last character somewhere, or set an end mark, eg. $ff, at the end of the text.

When you have a basic scroller going, you can add some colours to it (by changing the values in colour RAM, $d800 to $dbe7). One thing that has been used a lot is to have the outer characters have a darker colour than the central ones, which can make it look like the characters are fading onto the screen.

You can of course to other things with colours, like move the colours with the characters, so that each character has its own colour. Or you can have different colours cycling through the scroller.

Then, of course, there are all sorts of things you can do with your scroller to make it look more cool:

- Use your own character set, and it'll look a lot nicer (the ROM charset isn't very hot).
- Use a different sized character set, like 1x2 or 2x2.
- Make the scroller move at different speeds (by adding different values to $d016). Just make sure you move the characters at the right time, so that it doesn't jitter.
- Use $d011 to move it up and down while it's scrolling from the left to the right. You can use the same concept there, copying characters, to make it scroll more than 7 pixels in the Y direction.
- Use multi-colour characters.
- Use your imagination...

Raster Bars

I've already explained the basics of raster bars above: change $d020 and/or $d021 on every line. There are some issues you have to pay attention to to make it work well. You don't actually need a stable raster to make it work, all you need is the correct delay values.

The easiest thing is to do it in the upper or lower border. Then you'll only need to change $d020, and there are no Bad Lines that mess things up for you. All you need to do then is to write a loop that takes 63 cycles to execute (65 if you have an NTSC C-64). In the loop you read a colour value from a table (indexed by the x or y register), increment your index value, change $d020, do a delay, check if your index value has reached a certain value, otherwise branch back to the top. You'll also need to add some delay before the loop, to make it start at the right time.

It might seem a bit complicated to have to count the cycles that different instructions take, but you'll learn it quickly, and it definitely pays off. Of course you'll need some trial and error to get it to work the first times...

Some things can screw up the timing and make instructions take different amounts of cycles. If you're using sprites, they'll steal a few cycles on each line where they're displayed, so don't move sprites over your raster bars, at least not if they're changing Y coordinates. That requires much more timing.

You'll also have to watch the page boundaries. A branch that crosses a page border takes one cycle more to execute than normally. The same goes for indexed addressing modes. Your usual `LDA addr,X` instruction usually takes 4 cycles. But if `addr` is eg. $2080, and X is greater than $80, you'll cross the page boundary, and it'll take 5 cycles instead. In some cases it's much better to use a monitor, instead of an assembler, as you know exactly where the page borders are in your code.

By using different registers to count the times you do the loop and to index into the colour table, you can make the colours in the raster bar roll. Or use the same register, and alter the colour table itself each frame.

Opening the Top and Bottom Border

This is a simple trick, but it can be pretty cool to be able to put stuff in the upper and lower border. Of course it's even more cool to open the side borders, and conceptually that's equally simple, but it requires good timing.

To open the upper and lower border, you have the screen set up to display 25 lines of text (by setting bit 3 of $d011). Then, on the last line of the screen (on raster line $f8, $f9 or $fa), turn off that bit. The VIC will be fooled into not checking where to turn on the border, as it ``thinks'' it's already on, and it won't turn it on. You have to set the bit again, so that the VIC will be fooled again the next frame, but you'll have to wait a couple of lines first, or you'll fool yourself instead. :-)

You can display sprites in the upper and lower border, but not normal graphics. The same goes for the side border. However, the last address in the VIC page ($3fff in the default case, otherwise $7fff, $bfff and $ffff) will also be displayed, in black. So if you have another background colour than black, and the value in this address is not 0, you'll see a garbage pattern in the border area. You can actually use this for some effects... Use your imagination.

FLD

FLD stands for Flexible Line Distance, and was first introduced in the Think Twice demo by The Judges, in 1986, I think. With this routine, you can delay the display of the next character line for an arbitrary amount of lines. Everything that is to be displayed below is delayed, so you can scroll the whole screen down.

What you have to do is to delay the next Bad Line. A Bad Line is triggered when the three lowest bits of $d011 match the three lowest bits of $d012. So what you do is that you take the value from $d012, add a value to it, eg. 2, set the upper five bits to 0 (by using `AND #$07`), set the upper bits appropriately (`ORA #$18` for text mode) and `STA $d011`.

This means that $d012 and $d011 won't match on the next line. On the next line, you do the same thing again. And then you keep going until you think it's time to display the next text line. Of course you use a loop for this, and if you vary the number or times you go through the loop each frame, you can make text lines scroll up and down really easily. Combine it with a scroll text, and you'll have a scroll text that moves up and down. Nice, eh?

And of course, you can use this routine just to get rid of Bad Lines, which can be handy if you want to do eg. colour bars on the main screen. Just do an FLD and change $d020 and $d021 every frame, and you don't have to worry about Bad Lines.

Sprite Multiplexer

The principle of sprite multiplexers is very simple, but it can be a hassle to get a working implementation sometimes. What you do is that you change the Y coordinates of the sprites while they're being displayed, and as soon as the sprite is finished, it'll get the new Y coordinate, which is a couple of lines below. Note that the new Y coordinate has to be at least $15 greater than the previous value, or it won't be displayed at all the second time.

It gets even cooler if you change the sprite pointers too, but that has to be done at exactly the right time, because you can change the sprite data in the middle of a sprite's display, and then it'll just look weird, unless you really know what you're doing (it can be used for some advanced effects).

Use a sine table for the Y values, and find our where you have to change the coordinates.

Tech-tech

You might already have figured out that you can do a simple effect by changing the three lower bits of $d016 (the X-scroll) on every line. If you haven't tried it, you should at least be able to understand what happens: each line will have its own X-scroll value. By changing the X-scroll value in an ordered manner, you can make the graphics wave back and forth. But you don't get more than 7 pixels worth of difference between the lines, which is a pity...

You've probably seen real tech-techs (they're not so common these days as they were around 1990), and they definitely have a greater difference than 7 pixels. You might think that you can do that by moving the characters in screen RAM, but you can't, because screen RAM is only read each 8th line. What you have to do is to change the charset (using the four lower bits of $d018), to a new charset, which is shifted one character to the right. You can fit 7 charsets (you need space for the screen RAM too) into one VIC page (16 kB), which means you can make a 56 pixels wide tech-tech. Not bad, huh? It can be a bit tricky to get it to work the first time, though...

You can of course also change VIC bank (with $dd00), which means you can do the effect with graphics, not just with a charset logo, or just get a wider tech-tech.

FLI, AFLI, IFLI etc.

FLI is a graphics mode that gives you more colours per character than a normal multi-colour pixel. To achieve this, you need 8 different screen RAMs, with different colour values. The effect is achieved by triggering a Bad Line on every line ($d011) and changing $d018 to point to a new screen RAM (but the same bitmap) each line. You do this for 8 lines, then you start over again.

A normal FLI picture starts at $3c00. The area between $3c00 and $3fff is colour RAM, which is copied to $d800 in the initialization. The area between $4000 and $5fff is taken up by the 8 different screen RAMs, and $6000 to $7fff is the bitmap. So the values you'll change between in $d018 are $08, $18, $28 and so on up to $78.

AFLI is exactly the same thing except that it uses hires mode. As hires doesn't use colour RAM, an AFLI picture takes up $4000 bytes, and is laid out in the same way as a FLI picture.

IFLI is just like FLI, except that you have two FLI pictures, which you switch between every frame, as well as changing X scroll by 1 pixel. This is to give it the appearance of more colours and higher resolution, by interlacing it.

In all of these formats you'll see the so called FLI bug, which means that first three characters on every line are unusable. There's no way to get around this. In FLI and IFLI, you just have to set the values for these characters to 0, and they'll be black. In AFLI, they'll always be light grey, so if you want to have some other background colour, you'll have to cover these three lines with sprites.

The FLI effect also has the side effect of giving you a stable raster, which is always nice, of course.

Opening the Side Border

If you've got a stable raster, you can open the side border in exactly the same way as you open the top and bottom borders, but instead of $d011, you use $d016, and instead of once every frame, you have to do it on every raster line, where you want the border to be opened. This isn't the first effect you should try to code, but if you've got a stable raster, give it a try.

A hint: use `DEC $d016` and then `INC $d016`. To see that you're in the right place (just by the right border), use $d021 instead of $d016 when timing it.

Plasmas, Bumpmappers, 3d effects

As you might have noticed, I haven't explained stuff that's seen in many modern demos, like plasmas, bumpmappers, 3d effects etc. I'm not very much into this type of effects, although they can look nice sometimes. Not if they're displayed in 4x4 or 8x8, though! (At least very seldom...) These aren't specific to the C-64, and you can find information about them in lots of places. If you can do a plasma on some other platform, you can do a plasma on the C-64.

The easiest (and ugliest) way to display it is to use 8x8, ie. a normal text screen, where you change the colours of the characters. Another way you can do is to fill a bitmap with a pattern of the values $55 and $aa, so that you get a mixture of two colours in each 8x8 block.

4x4 is often used, and it uses a graphics mode called half-FLI, which forces a Bad Line each 4th raster line. Combined with the right char data and two screen RAMs, this gives you a screen with 80x50 resolution. It doesn't look very nice, but it can be a nice programming exercise... And it's popular, although chunky graphics modes seem to be going out of fashion. It's about time.

To do 3d effects with vector graphics, all you really need to know is how to put a pixel on a bitmap screen and draw a line. Once you've done that, you just need to optimize your code to make it smooth and nice.

A trick that is often used, to optimize all sorts of plotter routines, is to use a 16x16 chars area, where the first column has the chars 0, 1, 2, 3 etc. up to 15, the second column has 16, 17, 18 etc. In this way, you get linearly indexable Y coordinates (0 to 127). The reason for using a 16x16 chars area is that it all fits into one character set. You can make the area larger by shifting charsets in the middle, but that'll make the plotter routine a bit more complicated.

The Rest

The rest you'll have to figure our yourself, or read about in other places. Of course, as I've suggested above, variations of these basic effects can be made, and combining two or more of them can give very nice results. You have to be a bit creative to make interesting demos.

Of course, there might also be VIC tricks that haven't been discovered yet, although it's getting less and less likely.

An Exercise

As I've stated before, you won't learn demo programming by just reading about it. You need to write demos yourself. Before you start trying to use IRQ loaders and writing really advanced effects, try to write a part that consists of the following things:

- A logo at the top of the screen. You can draw it yourself, using some graphics program. Either use multi-colour mode or single-colour hires.

- 8 sprites moving around the screen in some sine pattern. This might be a bit tricky the first time you try to do it, but all you really have to do is to set up two tables, one for the X coordinates and one for the Y coordinates, and assign each sprite an offset into the tables. Then change these offsets each frame, and just read the values from the tables and poke them into $d000, $d001 etc. The easiest thing is of course to use tables that are $100 bytes long.
- A scroll text at the bottom of the screen.
- Raster bars in the top and bottom border. You don't have to open the border to do this, you just have to change $d020.
- A tune playing. You don't have to go through the hassle of learning to use a music editor, just rip a tune from some other demo. As stated earlier, they're usually located at $1000, and are often shorter than $1000 bytes, so often all you have to do is to start the demo, reset, go into the monitor and save the area from $1000 to $1fff.

You should of course familiarize yourself with some of the tools, so don't forget to pack and crunch this demo, so that you can just load it and type `RUN` to start it. See the Tools section below.

If you manage to put together a demo like this, you've come a long way. And if you do, please send it to me! It'd be really cool if people could actually learn to code demos from this document!

Tools

In this section, which maybe should have been called an appendix, I'll go through some tools you'll need to use when putting together demos. There are lots of free tools available on the FTP sites and the web, and if you don't have Internet access, you can get them from your contacts. If you don't have any contacts either, send me a couple of floppies and you'll get them back, filled with nice tools. My address is at the end of this document.

Cartridges

You need one of these. Without a good cartridge, your life is going to be a lot harder. The most popular carts are the Action Replay series, in particular version VI. Those can be a bit hard to come by, though, so you might have to stick to a Final Cartridge, or some other cart. The most important features are a fast loader an a monitor. If you have those, the cart is pretty okay. You need the monitor to move things around in memory, debug code etc. And well, you really can't live without a fastloader...

Packers and Crunchers

Before you release something, you want it load as quickly as possible, and to take as little disk space as possible. To achieve this, you use a packer or a cruncher. For the best possible result, you use both.

Packers use run-length encoding (RLE), which is a simple compression technique. What it does is to check for continuous sequences of the same character, and replace those sequences with that character and the count (marked up by a special markup character).

Crunchers are more advanced, slower and give better results, because they use a more advanced algorithm, which checks for common sequences in the data, and replace those by references to earlier identical sequences. If you think crunching takes too long time, try getting an REU (Ram Expansion Unit) and a cruncher that uses it, as that can improve crunching speed a lot.

If you have a means to transfer stuff between your C-64 and a PC or Amiga, you might be interested in checking out Pasi Ojala's PuCrunch, which crunches your C-64 files on the PC or Amiga side. Of course it's very fast, and it also gives very good results.

Some packers (eg. Sledgehammer) let you link all the files that you want to pack. This can be nice, although I prefer the method of saving the whole memory area that I use and pack that. It's a matter of taste, I guess.

IRQ loaders

An IRQ loader is a loader that does the loading in the main loop, without disturbing your interrupt routines. So it can load while you're displaying scrollers, rasterbars and sprites on the screen, and the people who watch the demo won't even notice, unless they have noisy drives or pay attention to the drive LEDs. Of course, the more raster time that is available, the faster your files will be loaded.

Most IRQ loaders work the same way, but some of them don't work on all drives, and some of they are faster. Some also offer decrunching on the fly, which means you won't have to take care of decrunching yourself, after the file is loaded. Just load it and use the data right away.

You usually have an init routine, which just sets up the drive, and which can then be overwritten (to save memory), and then you have the loader routine, which you'll have to save until you've loaded the final part. There's usually some documentation, which will tell you the details of how to use it.

Sprite, Char, Graphics editors

Well, if you're going to make your own graphics, you need editors to produce them. Everything you need is available. If you have the opportunity, get your hands on everything you can find and try everything out. After a while you'll have found some good editors, that you can use. Of course, this goes for all kinds of tools.

The basic stuff you'll need is editors for multi- and single-colour bitmaps, fonts sprites etc. There are also all sorts of conversion programs, to convert between different formats. And as if that wasn't enough, there are also editors for special purpose formats, like combinations of bitmaps and sprites. Some of them can even give you ideas for new effects...

Sine editors etc.

You'll need a sine editor to produce nice sine tables, to use for sprite movements, scrollers, plasmas and what have you. Most of them offer the same features, so just check out a few and you'll get what you want.

Some people don't use sine editors, but write small BASIC programs instead. That gives you more flexibility, of course. Here's a small example from BlackJack:

```
10 L=256: REM length of table

20 X=10: REM amplitude/2

30 A=1024: REM address

40 FOR I=0 TO 2*π STEP 2*π/L

50 : POKE A,SIN(I)*Y+Y

60 NEXT
```

He also adds: ``It's pretty easy to add stuff between line 40 and 60 to split a larger value into "X div 8" and "X mod 8" for scrolling big logos or similar things." If you want more interesting sine movements, you'll have to change line 50 (the `SIN(I)` part).

Music editors

The most popular music editors still seem to be DMC and the JCH editor. JCH's editor works more like a tracker, while DMC is a more traditional C-64 music editor, where you only see one sequence at a time. The both offer full sound editing capabilities, and the players offer decent rastertime and memory usage.

Some less used editors are Sidwinder by Taki/Natural Beat, which has a very optimized player, that takes up very little raster time, and EMS, by Odie/Cosine, which is, as far as I can tell, just as good as DMC and JCH.

A new editor that is meant to be as simple as possible to use is Odintracker. Unfortunately it doesn't have a very good packer, so the tunes take up huge amounts of memory. This might change, of course.

Text Editors and Scroll Text Writers

In addition to the above, you need text editors, to edit your scroll texts and other texts in your demos. Some text editors let you set the size of the pages, which can be handy, if you're using fonts of non-standard sizes, or only use parts of the screen to display text.

Scroll text writers are used for (surprise) writing scroll texts, and are pretty similar to text editors.

Noters

Noters are used for writing notes to contacts and notes for releases. Personally I think it's cooler to produce a special note for each release, but for small releases you can always use a standard noter. Which one you choose doesn't really matter, as long as it can display text.

Other Tools

There are other kinds of tools as well, but you'll have to discover those for yourself. Generally, if you find that you need a tool for something, it's pretty likely that someone else has already written it, so it's just a matter of finding it...

Common Pitfalls

I put this section near the end of the document, because it's supposed to be more like a reference section. Whenever you get errors that seem mysterious, things that shouldn't happen, you can check here and see any of the things pointed out might be responsible for your problems.

Final Cartridge

I don't know exactly what kinds of problems you might get with a Final Cartridge, but it's always a good idea to try your demos with the cart removed if you're using a Final Cartridge. If you get different results with the cart removed, you know that your problem is cartridge related.

If you have any details to share about Final Cartridge problems, please get in touch!

Graphics at $1000 or $9000

If your sprites (bitmaps, charsets etc.) look like they get their data from the standard CBM charset, you've probably put them in the memory between $1000 and $1fff or $9000 and $9fff. You can't use these areas for graphics, because whatever you do, the VIC will always see the standard charset in these areas. This is probably why the music is almost always placed at $1000...

Garbage in Vertical Border or Behind FLD

If you see black garbage in the top and bottom border or on the screen, while doing an FLD, and don't know how to get rid of it, simply clear the last byte of the current VIC bank, ie. $3fff for the first bank, $7fff for the second, and so on. Be warned though, that doing this without thought might give you other problems. If you have code in that memory position, setting $3fff to 0 might later crash your demo, or just make it misbehave.

Everything Works Until I Pack It!

So you get weird bugs after packing your demo? One possibility is a buggy packer. Another is that you're using un-initialized memory. The situation won't be the same after packing, as the memory positions you're counting on always being set to 0 might now have some other value.

If you think the packer or cruncher you're using is buggy, try another one. If your demo looks different with different packers and crunchers, stick to the ones that others recommend. It's not likely that eg. Sledgehammer II or The Cruncher AB have bugs. People have used these tools in thousands of products, so if they'd had bugs people would have discovered and fixed them.

A Few Words About Optimization

As a general rule of thumb, you want the screen to be updated each frame, even if you do effects that use complex computations. With some effects, this isn't trivial. If you've written a nice effect, but the computations take more raster time than what's available in one frame, you have to optimize your code, or live with an effect that doesn't look as nice as it could have.

What to Optimize

There's a lot to be said about optimization, and I can't cram it all into a document like this. Furthermore, not every optimization is applicable to C-64 code.

If you ask a computer scientist, he or she will tell you that you should first improve your algorithms, and then, if it's still necessary, improve the constant times that each iteration takes. This makes perfect sense, if you want to sort 10000 numbers, you shouldn't spend time optimizing your bubble sort routine, but instead implement some faster algorithm, like quick sort or merge sort.

But the algorithm usually isn't the problem with demo effects. The algorithm itself is most of the time pretty straightforward, and the problem is to get each computation as fast as possible. There's no way to decrease the number of iterations, but instead you have to decrease the time each iteration takes. This is the common case, I'd say, but there are lots of exceptions.

How to Optimize Computations

So, how do you optimize computations written in 6502 machine code? There are lots of different ways, and I'm no expert, but I'll share some simple tricks, and hopefully you can use them and invent your own techniques for making your code run faster.

The basis of optimizing a computation is of course to decrease the time it takes, ie. decrease the number of machine cycles that each computation takes. Each instruction takes a few cycles to execute. You'll need a table with the timing for the different instructions.

Removing Instructions

So the first rule of thumb would be: remove all unnecessary instructions. All instructions take a few machine cycles to execute, so by removing some of them, you're speeding up your code. An example of an instruction that can sometimes be removed without affecting the result is `CLC`. Read through the critical section of your code and try to find instructions to remove.

Substituting Instructions

Sometimes you can rewrite a simple sequence of instructions to execute faster. Imagine that you want to increment a byte in memory, and use this sequence for it:

```
            ldx var
            inx
            stx var
```

This is pretty bad, at least if you're not actually using the value afterwards. The above code takes $4+2+4=10$ cycles. If you'd just used a simple `INC var` instead, you would have saved 4 cycles!

Loop Unrolling

Loop unrolling means that you optimize away the overhead that's inherent in all loops, ie. the code that changes and checks a counter and branches. Let's say you have the following loop in a speed critical section of a program:

```
            ldx #$00      ; 2 cycles
loop
            inc table,x   ; 7 cycles
            dex           ; 2 cycles
            bne loop      ; 3 cycles
```

If you count the cycles in the above code, you'll see each iteration through the loop will take 12 cycles[11]. It's executed 256 times, which means that the total number of cycles used for the above code is $256*12+2=3074$ cycles. That's quite a lot of raster time!

Let's unroll the loop and see if we can get it any faster... The code will look like this:

```
            inc table     ; 6 cycles
            inc table+1   ; 6 cycles
            inc table+2   ; 6 cycles
            inc table+3   ; 6 cycles
            inc table+4   ; 6 cycles

            .
            .
            .

            inc table+253 ; 6 cycles
            inc table+254 ; 6 cycles
            inc table+255 ; 6 cycles
```

It's pretty obvious that this piece of code will need lots of more storage space (three whole pages), but let's say that we have lots of free memory, so that isn't a problem. What about the execution time? Well, each instruction takes 6 cycles, so we'll use $6*256=1536$ cycles. The used raster time is halved!

Of course, you won't gain as much as this in the usual case, but there's a lot of time to spare with this technique.

Use the Zero Page

An `LDA` or `STA` to a normal memory address takes 4 clock cycles, but if you use an address in the zero page, they only take 3 cycles. Need I say more? Maybe, so here goes: use the zero page!

Using Tables

Why do people use pre-calculated sine tables? Because that means they don't have to do the costly sine calculations in real-time. That shouldn't be news to you. But if you think about it for a while, you might realize that you can use tables for lots of stuff. Generally, if you have some kind of complex calculation, you can use lookup tables somewhere. If you're writing complex stuff with lots of shifts here and there, you can probably re-write that code to run in half the time by using a table.

Other Optimizations

The above techniques are just meant as examples of how to optimize code. The general idea should be pretty clear by now: do everything you can to make your code faster. Read through it and try to find ways to make it faster. Use tables, unroll loops, remove crap that's not necessary.

Other Ways to Make Code Faster

There's of course a lot in truth in the rule that you should first optimize your algorithms, and then the constant factors. Don't think that there's no better way of achieving an effect than to use brute force. You can always cheat in some smart way, use data that makes the job easier and the code run in half the time.

Re-using Stuff

In some cases you can reuse results of computations in smart ways. If you realize that your code is, for some reason, computing the same value twice each frame, and the calculation takes some time, store the value in a table and just look it up the next time you need it.

Re-using stuff can even be as simple as plotting the same point in different places on the screen.

Making Assumptions About Data

Don't do the mistake of trying to write nice and general routines. If you think that's the way to go, you probably shouldn't be writing demos. With nice, general routines, you have to take care of special cases and take care of weird data. But it's your code and your data, so your code should know what kinds of data it should expect. Don't make your code more complex than it needs to be. Only take care of the cases that can actually happen.

And of course, if you notice that your data demands a complex routine, think of ways to change the data, so that you can write a faster routine to handle it.

Other Documents

You'll need other documents, besides this little tutorial, or course. Below I list some references and some magazines with articles about demo coding.

Of course you'll also have to learn from the masters. When you watch a demo, try to figure out how the stuff you see on the screen is implemented, eg. what's done with chars and what's done with sprites, or where scrolling is used and where the graphics data is updated in realtime. See the URLs below for places where you can find demos.

References

- Mapping the C-64. This is a great memory map, with thorough explanations of all addresses in the C-64. It should be available at Project 64 (which also has some other useful documents): `http://project64.c64.org/index.html`.

- Programmer's Reference Guide. This one has a nice table of opcodes, with cycle counts, addressing modes etc. Most of the book is pretty useless, though. This one should also be available at Project 64: `http://project64.c64.org/index.html`.

- All About Your C-64. This is a document put together by Ninja/The Dreams, which can be really handy. Especially the HTML version is nice to have available when coding, and you can't remember quite which bit it was that you had to fiddle with to achieve something. This one's available from the The Dreams' home page at `http://www.the-dreams.de/`.

- The MOS 6567/6569 Video Controller (VIC-II) and its Application in the C-64, by Christian Bauer. This article should contain everything that is known about the VIC. There is a text version available at (it's also available in HTML somewhere else): `ftp://ftp.funet.fi/pub/cbm/documents/chipdata/VIC-Article.gz`.

Magazines with Tutorials

- C=Hacking. Great online magazine with articles about everything concerning CBM's 8 bit computers. In some of the early issues there are articles by Pasi ``Albert'' Ojala on demo effects. There's also a multipart Assembler tutorial in the first issues. Available from funet: `ftp://ftp.funet.fi/pub/cbm/magazines/c=hacking/`.

- Discovery. I've only found three issues of this mag, but it has some nice demo-related articles, which you can learn a lot from. Like C=Hacking, the mag consists of ASCII text files that you can download off the Internet. Available from The Fridge: `http://www.ffd2.com/fridge/discovery/`.

- Coders World. This is a disk mag, ie. a mag that you run on your C-64. Three issues were released, and they have a lot of simple code examples and explanations of how to make cool effects. This is a good resource for beginners. When you've learned a bit about the C-64, the info in this mag may seem a bit too basic. You can find the first two issues at eg. funet: `ftp://ftp.funet.fi/pub/cbm/magazines/disk/c64/`. The third one is a bit hard to find. You can get it from me if you can't find it on the net.

URLs

If you've got Internet access, which is getting more and more common, and have a way to transfer stuff between the machine you use to access the Internet and your C-64, you have access to all the demos and tools you need. Below are a list of a couple of FTP sites which, taken together, should offer you everything you need. If you need something else, just do a web search.

- `ftp://c64.rulez.org/pub/c64/`
- `ftp://utopia.hacktic.nl/pub/c64/`
- `ftp://ftp.elysium.pl/`

Contacting the Author

You can contact me by email or snail-mail. The snail-mail address may of course change, so use email if you can. My email adress is `puterman@civitas64.de`. My snail-mail adress is:

Linus Åkerlund

Kantorsgatan 38

S-75424 Uppsala

Sweden

Please contact me and tell me what you think about this document!

Copying this document

I'll try to keep this document as available as possible. For that reason, I'll release it into the public domain. Do what you want with it. Mirror it, copy it, change it, even sell it if you want to (if you can do that and still get a good night's sleep, capitalist bastard). Please feel free to print it and hand out copies to your friends, relatives, pets and random gamer lamers.

I'll make this document available on my home page, which will hopefully be available for a long time at `http://user.tninet.se/~uxm165t/`. That might change, though, and if you decide to upload it somewhere else, please tell me about it, so that I can add a URL here.

The document is written in LaTeX, and I'll make it available in source form (.tex file), as DVI, postscript, HTML and ascii text. Please convert it to other formats if you feel like it.

If you want to contribute changes, please change the text in the source and send the whole document to me, or as a patch (use the command `diff`, which should be available if you're on a Unix system).

About this document ...

An Introduction to Programming C-64 Demos

This document was generated using the **LaTeX**2`HTML` translator Version 99.2beta8 (1.43)

The command line arguments were:
latex2html `-split 0 demo_prog.tex`

The translation was initiated by Linus Åkerlund on 2001-05-22

Footnotes

... dangerous[1]
: A couple of guys, who weren't all that good at C, were writing a toy operating system. They got all sorts of bugs and weird crashes, and didn't know what to do. After a while they came up with a solution: to re-write the whole kernel without using any pointers at all. Don't ask me how they did it, but they did get it to work. Morale of the story: code some C-64 demos, and you'll never run into problems like that. :-)

... language.[2]
: I've heard someone say that some PC demos these days are written in C or C++, but of course, if that's true, then we're not talking about demos anymore.

... modern[3]
: What's modern is another thing that can be discussed forever. A C-64 is about as modern as an IBM PC clone.

... it[4]
: This code should be pretty assembler neutral, ie. you can use whichever assembler you want. You can't type this stuff into a monitor, though, as it uses labels. If you're going to use a monitor, you have to change the `JMP` instructions into jumps to absolute addresses, ie. change `JMP loop` into `JMP $1000`. You also need to remove the line `* = $1000`, which just means that the code should begin at the address $1000.

... program[5]
: See the Tools section on how to do that.

... long[6]
: Actually, if you do your math, you'll realize that it's only $03e8 bytes long. Then follows $10 bytes that you can use for whatever you want, and 8 bytes that are the sprite pointers.

... RESTORE[7]
: This doesn't work in all version of Turbo Assembler. If your version doesn't support this, you can add some code to do it for you.

... second[8]
: On PAL systems, that is, on NTSC systems the screen is redrawn 60 times per second.

... bit[9]
: Why 14 bits? Because the VIC can only address 16 kB of memory, which is what you get with a 14 bit address space.

... bitmaps[10]
: Not as weird as mode X on VGA cards, though.

... cycles[11]
: Except the last one, which will only take 11 cycles, as the branch isn't performed.

Linus Åkerlund 2001-05-22