How to generate Color video signals in software using SX chips

by

Rickard Gunée

http://www.rickard.gunee.com

Featuring the games Tetris and Pong
Copyright note
How to generate color video signals in software using SX-chips © Rickard Gunée, 2003. You may distribute this document on in digital or printed form if the purpose is non commercial and the content is not modified in any way.
Table of contents

1. BACKGROUND ................................................................................................................................. 5

2. THE COMPOSITE VIDEO SIGNAL .................................................................................................. 7
   2.1 HOW A STANDARD TV-SET WORKS .......................................................................................... 7
   2.2 DIFFERENT TV STANDARDS .................................................................................................... 8
   2.3 THE INFORMATION IN THE VIDEO SIGNAL ........................................................................... 8
   2.4 THE SCAN-LINE ...................................................................................................................... 9
   2.5 PUTTING THE SCAN-LINES TOGETHER TO AN IMAGE .......................................................... 9
   2.6 VERTICAL SYNCHRONIZATION PULSES ............................................................................. 10
   2.7 COLOR CODING ..................................................................................................................... 10
   2.8 QUADRATURE MODULATION ............................................................................................... 11
   2.9 PUTTING IT ALL TOGETHER ................................................................................................. 12

3. CREATING IT IN SOFTWARE ......................................................................................................... 13
   3.1 THE BASICS FOR COLOR GENERATION ............................................................................. 13
   3.2 MATHEMATICAL TRICKS ....................................................................................................... 13
   3.3 KNOW YOUR HARDWARE ..................................................................................................... 14
   3.4 OUR NEW PARAMETERS ........................................................................................................ 15
   3.5 PHASE ALTERNATING LINE ................................................................................................. 15
   3.6 VIDEO OUTPUT HARDWARE ............................................................................................... 16
   3.7 LIMITATIONS WITH COLORS ............................................................................................. 16
   3.8 USE OF PALETTE .................................................................................................................. 17
   3.9 OUTPUTTING MONOCHROME OBJECTS USING PALETTE .................................................. 18
   3.10 COLORED TEXT LINES ...................................................................................................... 18
   3.11 EMULATORS ......................................................................................................................... 18

4. GAME SYSTEM .............................................................................................................................. 19
   4.1 SCHEMATIC OVERVIEW ....................................................................................................... 19
   4.2 JOYSTICKS ............................................................................................................................ 20
   4.3 THE OSCILLATOR .................................................................................................................. 20
   4.4 TV CONNECTION .................................................................................................................... 21
   4.5 PCB ........................................................................................................................................ 21

5. TETRIS ........................................................................................................................................ 23
   5.1 HOW TO PLAY THE GAME ..................................................................................................... 23
   5.2 THE SOFTWARE ..................................................................................................................... 23

6. PONG ........................................................................................................................................... 25
   6.1 HOW TO PLAY THE GAME ..................................................................................................... 25
   6.2 THE SOFTWARE ..................................................................................................................... 25

7. CONCLUSIONS ............................................................................................................................ 27

APPENDIX A: COLOR TEST1 SOURCE CODE .................................................................................. 29
APPENDIX B: COLOR TEST2 SOURCE CODE .................................................................................. 35
APPENDIX C: TETRIS SOURCE CODE ......................................................................................... 41
APPENDIX D: PONG SOURCE CODE ............................................................................................ 61
APPENDIX E: GAME SYSTEM PCB LAYOUT .................................................................................. 81
1. Background

Back in early 1998 I made some experimenting using a PIC16F84 microcontroller (3MIPS of processor power) to generate composite B&W video signals on the fly in software, with two resistors as the only video hardware. I made the two classical games Pong and Tetris with this technique and published them including source on my homepage. Since then it has been built by several hundreds of people. During the Christmas 1998-1999 I got some equipment from Scenix (nowadays known as Ubicom) and made some experiments to generate color video signals using an SX chip, but before I got any results my programmer broke down, at least that was what I believed, and I stopped developing it. In the early summer of 2001 I was told by people at Parallax that it was the early versions of the SX-chips that had a bug in them so my programmer was just fine, so they gave me some new chips and I continued my work. After some new experiments, calculating and many late hours and a bit of luck I got my TV to lock onto the color signal and by the end of summer I got a Tetris game up and running. During the fall I developed the Pong game, which was finished during the Christmas holidays 2001-2002. I didn’t release the games as there were some details left to take care of. I didn’t want to publish them until they were as perfect as possible due to my bad experience with my PIC-based games that were spread in early bad versions. Now in spring 2003 I decided that I shouldn’t do any more improvements of the games as I don’t have time to work on them and I got to stop sometime. The biggest remaining issue is that it only works good for NTSC, it is much harder to get a correct PAL signal in software, but that is a problem for someone else to solve. Another issue about the games was this text about generating color video signals that I wanted to finish before I released the games, to not get that many questions about video generation that I don’t have time to answer. After reading this document you will hopefully understand how to generate color composite video signals in software. To fully understand this you need mathematical knowledge at university level, some RF-knowledge would also help a lot.
2. The composite video signal.
To understand anything about generating video signals in real-time, one must know how video-signals work in detail, so before we look at any code we'll have to talk about video signals.

2.1 How a standard TV-set works

A standard TV-set is built with a vacuum tube, which has a phosphor screen that an electron cannon shoots at. When the electrons from the cannon hits the screen, light is emitted from the phosphor when the cannon shoots electrons at it, and it also has a short afterglow making each pixel lit until the electron beam hits it again. The electron beam from the electron-cannon can be bent using magnets so it shoots at different parts of the screen. If this is controlled so it draws horizontal lines all over the screen repeatedly, while the intensity of the beam is modulated, an image can be drawn on the screen. The screen is redrawn 25 times per second (on a PAL system), but to reduce flickering the image is interlaced, showing first all odd lines then all even lines, so the image is partially updated 50 times per second. To get color each dot on the screen is divided into three colors: red, green and blue.
2.2 Different TV standards

There are three major analog TV-standards: NTSC, SECAM and PAL as seen on the map above. The NTSC (Short for "National Television System Committee", but back in the early days of TV there was problems with getting the same color over the whole picture so a more evil interpretation of the letters is that it stands for "Never The Same Color" ) is the American TV-standard, it has only 525 scan-linelines, but it has a update frequency of 30Hz. SECAM (Short for "Sequential Color And Memory", but as the French usually want to get their own solution to problems, a more evil interpretation is that it stands for "System Essentially Contrary to the American Method") is the French TV-standard, it has improved color stability and higher intensity resolution but with less color resolution, I don't know much about that standard. The European standard is PAL (Phase Alternating Lines, or as a PAL enthusiast would interpret the letters: "Perfect At Last"), it has 625 lines per frame, 25 frames per second. It is based on NTSC, but the color-coding has been improved by using a phase shift on every other line to remove the color errors that occurred with NTSC.

2.3 The information in the video signal

The image seen on the screen has different intensities. As the electron beam sweeps over the screen, the intensity that should be at the position of the beam, is sent as a voltage level in the video signal.. There is no information in this intensity information about where the electron beam is on the screen. To solve this, a synchronization pulse is sent in the beginning of each line to tell the TV that the current line is finished and move down the electron beam to the next line. (Like the <Enter> key on the keyboard, when writing a text with a computer) The TV must also know when a new image is coming, this is done by making a special synchronization pattern. (Like the "new document" function when writing
a text with a computer) An image that is updated 25 times per second would be quite flickering, so therefore all even lines are drawn first and then all odd, this method shows 50 half images per second, making the picture have less flickering. The information whether the image contains even or odd lines are sent in the vertical synchronization pattern, as different patterns for odd and even images. The video signal has a voltage range 0 to 1V, where 0.3V represents black, and 1.0V is white (gray intensities have voltages between these values). Levels close to zero represent synchronization pulses.

2.4 The scan-line
The image is divided into scan-lines, it is the most important part of the image since it contains the image data. The scan-lines are all 64us long. First a 4us long sync pulse is sent, by setting the signal level to 0V, to tell the TV that a new line is coming. The old TV's was kind of slow, so they needed 8us after the sync-pulse to get the electron beam in position. During this time the signal is kept at black level. The 8us delay is followed by the image data for 52us, drawn on the screen from the left to the right with the intensities obtained from the video signal. Black is represented by 0.3V and as the voltage increases the intensity increases, with the maximum intensity at 1.0v (white). See the image right to see the scan-line. The color information is added as two amplitude modulated sinus waves, we'll get back to that later.

2.5 Putting the scan-lines together to an image
An image is built from 625scanlines, but a TV doesn't show 625 lines. Some of the lines are used for synchronization pulses, and some lines are invisible (I don't know exactly how many) because old TVs needed some time to move the electron beam from the bottom of the screen. (Those invisible lines are nowadays used for other purposes, Text-TV for example).
2.6 Vertical synchronization pulses.
To tell the TV that a new image is coming, a special pattern of synchronization pulses is sent. Since the picture is built from two half pictures, the pattern is different for the odd and even images. The vertical synchronization pulses looks like this:

```
<table>
<thead>
<tr>
<th>Field I</th>
<th>Pre-equalizing pulses</th>
<th>Vertical sync</th>
<th>Post-equalizing pulses</th>
</tr>
</thead>
<tbody>
<tr>
<td>621</td>
<td>622</td>
<td>623</td>
<td>624</td>
</tr>
<tr>
<td>Field II</td>
<td>Pre-equalizing pulses</td>
<td>Vertical sync</td>
<td>Post-equalizing pulses</td>
</tr>
<tr>
<td>309</td>
<td>310</td>
<td>311</td>
<td>312</td>
</tr>
</tbody>
</table>
```

This picture shows the different vertical synchronization pulses for the two half images. The levels are 0v and 0.3v. (Numbers below signals shows scan-line number)

2.7 Color coding.
When color was introduced, it was the same problem as with any change in technology, there is always a demand for backwards compatibility that limited the new technology. For video signals this meant that a color video signal should look very much like a B&W signal so old TVs would still work. The problem was solved by overlaying the color signal with an amplitude modulated carrier on top of the video signal. In average the video signal would still be the same for B&W and it would not be noticed if the carrier had high enough frequency and the modulation also was kept to a low bandwidth.

The intensity of the TV signal is the sum of the Red, Green and Blue parts (weighted with the eyes sensitivity coefficients for those colors) in the video signal, and since that information is already given in the B&W signal then the additional color information only needs to contain two components with color difference. With the intensity sum and the two components G-R and G-B, it is possible to derive the R,B and G values. Humans have higher resolution for light intensity than for color, so using higher bandwidth for intensity than for color variation is very appropriate. Limiting the color information to two components is especially great as it is possible to transfer two signals using quadrature modulation, making it possible to transfer color using only one carrier overlaid on the B&W video signal!
## 2.8 Quadrature modulation

Quadrature modulation is a general method for modulation of a carrier. The idea is to change both amplitude and phase of the carrier to be able to send two signals with one carrier frequency. Each signal has its own carrier, one is \( \sin (2\pi f_c t) \) and one is \( \cos (2\pi f_c t) \), which makes it possible to reach all phases and amplitudes by modulating the voltages of the two signals. This method is not only used for TV color modulation, it is widely used, for example this is how stereo information is sent over radio also. It is a clever way to use the bandwidth to the maximum, with standard amplitude modulation only one channel is used, the other is just wasted. In order for this method to work, there must be a “pilot”, a reference signal that makes synchronizes the oscillator in the receiver with the one on the transmitter.

How the quadrature modulation is used differs slightly between PAL and NTSC. One variation is the white level as PAL where developed after NTSC and has hence more accurate coefficients to the newer more luminant phosphors used in modern CRTs. The colors are weighted according to the eye’s sensitivity, so the green color is weighted the most, blue the least and red in the middle. Using RGB-color levels detected by the “video camera”, the luminance is calculated according to:

\[
\begin{align*}
\text{PAL:} & \quad Y = 0.222R + 0.707G + 0.071B \\
\text{NTSC:} & \quad Y = 0.299R + 0.587G + 0.114B \\
\end{align*}
\]

The \( Y,U,V \) component transformation can be described as a matrix, for PAL the matrix looks like the following.

\[
\begin{bmatrix}
  Y \\
  U \\
  V \\
\end{bmatrix} = \begin{bmatrix}
  0.299 & 0.587 & 0.114 \\
  -0.147 & -0.289 & 0.436 \\
  0.615 & -0.515 & -0.100 \\
\end{bmatrix} \begin{bmatrix}
  R \\
  G \\
  B \\
\end{bmatrix}
\]

In NTSC the \( U \) and \( V \) components are rotated 33 degrees to minimize bandwidth for \( Q \), the rotated components are called \( I \) and \( Q \), calculated according this:
\[ I_i = V_i \cos(33^\circ) - U_i \sin(33^\circ) \]
\[ Q_i = V_i \cos(33^\circ) + U_i \sin(33^\circ) \]

For NTSC the \( Y, I, Q \) components can be described using the following conversion matrix.

\[
\begin{bmatrix}
Y \\ I_i \\ Q_i
\end{bmatrix} =
\begin{bmatrix}
0.299 & 0.587 & 0.114 \\
0.596 & -0.274 & -0.322 \\
0.211 & -0.523 & 0.311
\end{bmatrix}
\begin{bmatrix}
R \\ G \\ B
\end{bmatrix}
\]

2.9 Putting it all together

The output is created with quadrature modulating as described before by modulating a cosine and a sine with the \( U \) and \( V \) (I and Q for NTSC) components and sum the result together with the lumination component \( Y \). For PAL there is also a change in sign of the sinus component to compensate for phase error that will take out each other. (That is why it is called Phase Alternating Line). The video signal is calculated according the following.

\[
\text{PAL: } S(t) = Y + U_i \cos(2\pi f_c t) \pm V_i \sin(2\pi f_c t) \\
\text{NTSC: } S(t) = Y + I_i \cos(2\pi f_c t + 33^\circ) + Q_i \sin(2\pi f_c t + 33^\circ)
\]

So the color coding is just as simple as that, but there is one detail left, there must be a pilot signal in order for the quadrature modulating. In most systems using quadrature modulation, the pilot signal is sent constantly as a tone in the signal, for TVs however that would disturb the image too much. If there is an oscillator in the TV that is very stable, it would be enough to send a couple of cycles of the pilot periodically for the oscillator to tune in to, just often enough for the oscillator to keep its phase. In the B&W signal there is a gap of about 7\( \mu \)s between the sync pulse and where the image information starts, so it was an obvious place to put the reference carrier. This is 10-12 cycles of the color carrier (amplitude of 20IRE = 0.15V) and referred to as the “color burst”. The color burst is also shifted \(+45\) degrees on every scan-line for PAL.

![This picture shows the scan-line including color burst.](image)
3. Creating it in software

Generating a B&W signal is not very complicated; it is just hard work as it is a question of counting clock cycles to make the program flow take exactly the same amount of clock cycles all the time. When doing a color signal, this is even more important, if the line is one cycle too long or short (An error of 0.03% in scan line length) the TV can’t lock to the color carrier at all, for a B&W video signal the timing is not this critical, most TVs can compensate for quite large errors in a B&W video signal, so you could make the scan line’s length several tenths of cycles wrong without noticing as the TV compensates for it, but as our goal is to make a color video signal we are not allowed to do any errors at all. To make the job of timing easier I’ve created a general delay macro that delays for a given time using a minimal amount of program memory. I’ve also tried to use a lot of “EQU-constants” to make the code more readable and make the code possible to run for both NTSC and PAL by only changing the constants so the code is the same for both systems.

The first thing the software needs to do is output the vertical sync pulses, to tell the TV that a new frame has started. Then for the following 304 lines (254 for NTSC) it should keep each line 64us long and start each line with a horizontal sync pulse. Later on when doing a color signal a color burst must also follow after the horizontal sync pulse. During the 52us of image time the software needs to vary the voltage of the video signal between 0.3v (black) and 1v (white) as the electron beam sweeps over the screen and try to do draw something as the electron beam sweeps over the screen. This is quite easy with an SX performing 50MIPS, I’ve done B&W games this way using a PIC16F84 performing 3MIPS, so one could do B&W games with quite high resolution using an SX. However, generating color is much more cool, so let’s talk about color generation now.

3.1 The basics for color generation

As you would know after reading the chapter about video signals, the software needs to create modulated sinus and cosines waveforms for color information and sum them together with the intensity waveform. To get a good result the sample rate needs to be much higher than the color carrier frequency, and the software must also be able to do the needed calculations for the waveform which in total would need a very powerful processor if there is no hardware to help. An SX processor performing 50MIPS would not be good enough using this method.

3.2 Mathematical tricks

However, there is fortunately a better way to do it. The color carrier part of the signal is the sum of a sinus and a cosines with the same frequency but different amplitude, this is very fortunate as the cosines could be rewritten as a sinus with it phase shifted 90 degrees compared to a cosines. Ok so what good is that, well, the sum of two sinuses with same frequency and fixed phase difference but with
varying amplitude could be rewritten as one sinus with alternating phase and amplitude according to:

\[ f(x) = a \sin(x) + b \cos(x) = \]
\[ = \sqrt{a^2 + b^2} \left( \frac{a}{\sqrt{a^2 + b^2}} \sin(x) + \frac{b}{\sqrt{a^2 + b^2}} \cos(x) \right) \]

The coefficients preceded \( \cos \) and \( \sin \) describes a point on the unit circle and could be replaced with \( \cos \) and \( \sin \) with the angle \( \alpha \) according to:

\[ \frac{a}{\sqrt{a^2 + b^2}} = \cos(\alpha) \]
\[ \frac{b}{\sqrt{a^2 + b^2}} = \sin(\alpha) \]

This equals a rotation by an angle \( \alpha \) according to:

\[ f(x) = \sqrt{a^2 + b^2} \left( \cos(\alpha) \sin(x) + \sin(\alpha) \cos(x) \right) = \]
\[ = \sqrt{a^2 + b^2} \sin(x + \alpha) \]

Making it possible to express the sum of an amplitude modulated \( \sin \) and \( \cos \) with one \( \sin \) that is both amplitude and phase modulated.

### 3.3 Know your hardware

Ok we got rid of one of the components but still have one sinus that needs to be generated requiring a lot of CPU power. At the input of a TV there is a low-pass filter to limit the signal within a video signals allowed bandwidth of about 5MHz, which is very good because that means that a square wave at the color carrier frequency would look like a sinus to the TV as the high frequency components of the square wave have been filtered away. Now we are down to a square wave with changing phase, amplitude and offset, which is possible to generate in software with an SX@50MHz if the number of phases is limited and the clock frequency is a multiple of the color carrier frequency. In my projects I clock the SX with 12 times the carrier frequency for both PAL and NTSC, which gives 53.156550MHz for PAL and 42.954540 for NTSC, the over clocking of a 50MHz SX chip to 53MHz in the PAL case seems not to be a problem at all.
3.4 Our new parameters

The simplified signal with the square wave works like this: The average voltage of the signal control the lumination, the amplitude of the signal controls the whiteness and phase controls the color. When using 12 times the color carrier it is possible us get 12 different colors with different variation in intensity and whiteness. The first test I made with color generation was to examine the 12 base colors available, this test I shown in the picture to the left below. The source for this test can be found in Appendix A. (This is the only one of my current programs actually performing phase alternation in PAL, so the phase errors for PAL are not visible in this example) All possible variations for the 12 base phases can be seen here to the right below where all possible values for first and second amplitude are shown for all 12 phases and five bits. (There are 25*25*12/2 - 25*5 = 3625 combinations) The source for the later is available in Appendix B.

![The 12 phases, generates these 12 base colors.](image1)

![The available colors for 5bit DA and 12 phases.](image2)

3.5 Phase Alternating Line

is what PAL stands for, and that is a problem, when generating a PAL signal one should switch the phase of the signal 180 degrees on every line (color burst switched 90 degrees), this is not possible with the method I generate color signals. It is possible to produce more simple graphics such as one colored horizontal lines and phase alternate, but when doing more complicated stuff (like text or graphical objects) it becomes a problem as not only is the phase alternated, so is the positions of the graphics as the graphics must be aligned with the color carrier cycles. I chose to solve this by ignoring the phase alternation, with the downside that it makes phase errors visible as they did originally with NTSC where there is no phase alternation. With NTSC this is no longer a problem as the modern TVs have become better and lock to the color carrier much better, which the PAL TVs didn’t have to as their color method compensated for this problem, giving me a problem when I “cheat” when generating my video signals. I have no good solution for the problem with PAL to be software generated; it is up to someone else to figure that one out. (All pictures in this document are from the NTSC versions as they are the only pictures that are good enough to digitize with the TV-card in my computer)
3.6 Video output hardware

To be able to generate the signal we need a DA-converter. To make this simple a resistor based DA is the way to do it. There are two kinds of resistor DAs, $2^N$-nets and R2R-ladders. The $2N$ net is the simplest solution, it looks like this:

![2^N DA converter schematic](image)

The downside with the $2N$-net is that it is very inaccurate; the R2R-ladder requires twice as many resistors but has much higher accuracy, it looks like this:

![R2R DA converter schematic](image)

First I chose 6 bits for the DA as that is the largest number of bits that would be useful using 1% accuracy resistors, later I found that five bits is enough, the extra bit is better off in the DA, so the finished system go five bits for both sound and video. The video bits is bit 1 to 5 in my system as I already had done a optimizations in the code for using the lower 6 bits of portb making it the easiest solution, but when designing I new 5 bit system it is of course better to use bits 0 to 4 instead. Output voltage should be in the range 0 to 1.25V, which sets the values of the resistors to 220$\Omega$ and 440$\Omega$, but as there are no such resistors, it is better to keep the 1:2 ratio and use 221$\Omega$ and 442$\Omega$.

3.7 Limitations with colors

The color bandwidth is very low so it is not possible to change colors fast. In my games I keep the color phase constant within a graphic object and only change lumination level once every color cycle. This gives a maximum resolution of...
2766/12=230 pixels per scan line for PAL and 2233/12=186 pixels per scan line for NTSC. In reality not all pixels can be used as color (phase) changes cost time and thereby color cycles, and then the graphics also has to be calculated to there are not all of these pixels that actually can be used.

3.8 Use of Palette

To save memory a palette is often used in computer graphics cards. A palette is basically a color lookup table. In most cases the palette contains $2^N$ colors, usually 16 or 256 colors to be able to get each color into a nibble or a byte. If a picture only uses 16 different colors, then it needs 6 times less memory compared to if each byte would have been stored as three 8bit values with the RGB-components if a 16 color palette is used. In my games a palette is used to need less data for some of the graphics, a 16 color palette is used, however the lookup table doesn’t store the RGB values, instead it stores high and low period values for the square wave. In other words, my palette only contains info on brightness and whiteness, the color is set by the phase of the square wave which is not stored in the palette. Only one palette is used for both my games and it stars at black level, moves to color with maximum intensity, and then moves to maximum white. (See diagram below.) This palette makes it possible to generate objects with a 3D-feeling as it is possible to make dark shadows and more illuminated parts within the same object, but the object must be “monochrome”. It is possible to generate palettes with a 180 degree phase shift and get the complimentary color, but as the bandwidth is limited it is not possible to mix colors from the two phases in any order, it takes almost one color cycle for the phase change. (If the graphics is carefully planned to get few phase shifts, this could probably be used to do some really cool two colored objects)

The BCW-palette used for monochrome objects in my games
3.9 Outputting monochrome objects using palette

When showing graphics with high resolution (one intensity change per color cycle) it is not possible to calculate the graphics in real-time, so the graphics needs to be pre-calculated and stored in a buffer and then outputted from the buffer. I have created a routine that gets 4-bit graphics from the upper nibble in program ROM, translates it using a palette and store it in a buffer, consuming 31 clocks per pixel. A matching output routine, called memtovideo, which outputs data from the buffer at a speed of one pixel per color cycle (12 clock cycles). During the calculation of the next object it is not possible to show any graphics except for black or different gray colors, so therefore the layout of the graphics is very critical. In my Pong game I use three different graphic buffers, one for each paddle and one for the ball, and the graphics calculation is dynamically changed depending of where the ball is on the screen because the ball position controls where on the screen there are black surfaces that can be used for graphics calculations. In Tetris the graphics for the screws beside the graphics is calculated to the right of the playfield on the line above the one where the graphics is shown, and as both screws are identical only one graphics calculation is needed but it is outputted twice (one time on each side).

3.10 Colored text lines

The texts that appear in my games are generated on the fly; only two ROM-accesses are needed per character. First the character is read from the string stored in program ROM (low 8 bits), then this is used together with the line number to find the graphics from the font that also is stored in program ROM. Each character is 7 pixels wide, and separated by two pixels, originally the separation was three pixels but after unrolling the loop I got it down to two pixels (At the cost of program memory usage). The separation could probably be decreased to one by more unrolling at the cost of more program memory. A font is quite expensive in memory usage, so to save memory I only store the characters I use. The color generation in the text output is done by having a high and a low level for each pixel, the high level is an input parameter and the low level is always black to optimize the routine.

3.11 Emulators

Developing this kind of software is always much easier, but there are unfortunately no emulators available for color composite video signal generation with SX chips. However, there are some interesting open source stuff that might could be used as a good base for developing an SX color video game emulator.
4. Game system

The power supply is standard, a 7805 regulates the voltage to 5v, there is a rectifier at the input to be able to run the system on both AC or DC, the voltage can be 9..15v something. Then there is a bunch of caps on the board to get rid of noise etc.

The video generation is quite simple; it is just a five bit R-2R resistor ladder. It might seem a little bit strange that I connected it to bit 1...5 instead on 0..4, but that is because when I first made the prototype it had six bits for video and four for audio. I chose six bits first as it is the largest number of bits you should use with an R-2R ladder when using 1% tolerance resistors. Later I understood that it was not needed that many bits for video, that last bit would be better off in the audio generation. At the end of the R-2R ladder I have put one 1pF cap to get a a
little bit of filtering if the TV's input has to high bandwidth as my color generation technique generates square wave that needs to be filtered so only the "sinus part" remains. The resistor ladder has an open output as it is supposed to be connected to the TV that has an 75Ohm input that ends the ladder.

The audio part is very similar to the video part, also a five bit DA using an R-2R ladder. The difference is at the end of the ladder, the audio has a 100 Ohm pot to regulate the volume. The 100kΩ pot also ends the ladder as the audio impedance varies a lot between different audio inputs. (1k...20kΩm)

4.2 Joysticks
The joystick inputs are extremely simple, just five pins on the chip connected directly to the joystick inputs. The joystick pins on the SX-chip have their internal pull-up resistors enabled so there is no need for external resistors. There are two joystick inputs, and as with my PIC-based games I used old C64/Amiga/Atari joysticks. If you don't have one you could build one quite easy using the schematic here to the right using five off(on) switches and a 9pin female dsub.

4.3 The Oscillator
One of the more tricky parts is the Oscillator. This should run at 12 times the color carrier of the TV-system. The built in generator in the SX-key programmer is not accurate enough for video generation, so an external oscillator is needed. During the development of the games I used an almost 30 years old frequency generator (as new ones cost a fortune) seen at the picture here to the right, which made the development a lot easier. There chip-oscillators available that can be programmed once just like you can program a microcontroller. See the table below for what frequencies to use.

<table>
<thead>
<tr>
<th>TV System</th>
<th>Carrier Frequency</th>
<th>ClockFreq = 12 x Carrier Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAL</td>
<td>4.4297125 MHz</td>
<td>53.156550 MHz</td>
</tr>
<tr>
<td>NTSC</td>
<td>3.579545 MHz</td>
<td>42.954540 MHz</td>
</tr>
<tr>
<td>PAL-N*</td>
<td>3.575611 MHz</td>
<td>42.907332 MHz</td>
</tr>
<tr>
<td>PAL-M*</td>
<td>3.582056 MHz</td>
<td>42.984672 MHz</td>
</tr>
</tbody>
</table>

*Note: None of the games have been tested and calibrated for PAL-M or PAL-N.*
4.4 TV connection

The system is connected to the TV’s SCART input with a cable with RCA inputs and a SCART contact at the end. These cables are available in most TV stores. The best thing to do is buying a finished cable, building one is more expensive and doesn't give a good result, however if you still want to build one you should follow the schematic here to the right.

4.5 PCB

I’ve made a PCB design for the game system, available in Appendix E. This is quite simple as the game system is very simple, just a one layer. The PCB is stored in scale 1:1 so if you print it directly from this document you will get the size correct. The component placement is also available in Appendix E. Note that for my games you don’t need the expansion memory, I might do games later on that will use it but nothing planned yet. To avoid cracking the programmer I used a 90 degree bent connector for the programmer so it lies flat on the PCB when connected. There is a jumper close to the oscillator and the programmer that selects between programmer and oscillator as both can not be connected at the same time. As mentioned before, the resistors should be 220Ω and 440Ω, but as there are no such resistors, it is better to keep the 1:2 ratio and use 221Ω and 442Ω.
5. Tetris

The first game I made in color using SX-chips was Tetris. Tetris is an old Russian computer game where you should try to fit in block into a play-field, quite simple but really fun. All blocks are built from four bricks (the name Tetris is derived from the ancient greek word for four: "tetra"), there are seven combinations of the four bricks as seen here to the left. This version is using my PIC Game System as platform, generating a video signal in software. The video generating hardware is a 5-bit DA converter built with a few resistors. Usually the video signal is generated in video games is created with a dedicated video chips, reading the image data from a graphics memory. In this project the video signal is calculated in real-time by the microprocessor as the electron beam sweeps over the screen.

5.1 How to play the game.

When the power is turned on the game starts! (was no memory left for a fancy intro screen or similar). The score is shown left of the gamefield, and the next block to come is shown in the upper left corner of the screen. As the blocks fall down, they can be moved sideways by using the joystick (left gameport on hardware), the fall speed can temporary be increased by moving joystick down. The fire-button is used to rotate the blocks. When one horizontal line is full, then it is removed. You get points for full lines and placed blocks. As you get more points the difficult level is increased by increased block falling speed. The musics speed is increased as the game speed increases. You get game over when the playfield level has reached to the top and there is not room for more blocks (See picture here to the right).

5.2 The software

One of the problems for Tetris is the memory required. The size of the playfield is 16x8 bricks, to be able to keep track of thee 7 different block kinds (different color for each kind) and also be able to represent empty area, 3 bits are required for each brick. As one byte is 8 bits I chose to represent each brick as one nibble (4 bits), making the playfield 64 bytes. I chose to organize the memory making to the top 4 banks of the memory and letting each memory bank represent two
columns. The main game variables are placed in the first bank, some less used data such as score and a buffer of the next block and some other misc. stuff are placed in the second bank. The two remaining banks (except for the top four bytes of the fourth bank) are used as graphics buffers when outputting data to the DA. The sound frequency and sample position are stored in the top four bytes of the fourth bank.

The tune Karboschka is stored in program memory as 52 notes and 52 note lengths, where the note refers to a frequency table with frequencies according to the tempered note scale (half notes differs one twelfth root of two in frequency). There is a 32-sample 4-bit sinus wave in program memory that is outputted to the audio DA at the pace of the current note translated through the frequency table. The code outputting the frequency is performed during the horizontal sync pulse, and the tune is updated at the bottom of the screen before the vertical sync. As the number of bits used for music is not very high, it sounds a little bit distorted and not very good, but better than nothing =)

Most of the game data of the game is stored as one big chunk to be able to use the program ROM more efficient. This is done by using all the 12 bits and the iread instruction, which makes it possible to store 50% more data than by using retlw, but at the cost of speed. It is hard to use 12bit data efficiently, but to make it easier I chose to separate the gamedata into one fastmem- and one slowmem-part, where the 8 lower bits of each 12-bit word is the fastmem and the upper 4 bits are the slowmem. Getting one byte from the slowmem requires two iread but the fastmem only requires one. Graphics objects are stored as 4-bit palette values, so is the music, but the font and text strings are all 8-bit values, so it is quite natural to store the 4-bit data in the upper part and the 8-bit data in the lower part.

The software is written to run for both PAL and NTSC with almost the same code, done by making all timing with constants. The constant system selects what TV system to use. In the code I have also prepared timing for PAL-M and PAL-N but they are not tested. It is not possible to generate SECAM color video signals in software with this design, so there is nothing in the code to support it. Note that the frequency which the chip should be clocked depends on your TV-system.
6. Pong

After making the tetris game, it was very easy to make a Pong game. The game Pong was the world's first video game in the early 70's; this is a modern version of it, made with a little bit less hardware than the original version. In my version, the video signal is generated in software. The video generating hardware is a 5-bit DA converter built with a few resistors. Usually the video signal is generated in video games is created with a dedicated video chips, reading the image data from a graphics memory. In this project the video signal is calculated in real-time by the microprocessor as the electron beam sweeps over the screen.

6.1 How to play the game

The first screen is where you select how you want to play by moving the joystick: UP and DOWN to select Human vs. Human, Human vs. Computer or Computer vs. Computer. Start with FIRE. The computer vs. computer game to plays forever or until someone reset the game using the reset switch. You start serving by pressing fire, it is also possible to change direction and speed of the ball using fire. The player who has the serve will get points. If the player with the serve miss the ball, then the serve goes over to the other player. The paddles are moved up and down with the joysticks. It is possible to smash (increase speed) by pressing FIRE, and when doing so it is also possible to steer the ball by moving joystick up or down. When someone wins a game over picture will show and tell who won.

6.2 The software

The game logic is taken care of in the invisible lines at the top of the screen. The ball and the paddles are first generated with the setgrapics routine that loads the bitmap data and converts it using a palette and then writes it to the output buffer. The data in the buffer is outputted to the screen with the memtovideo routine. there is a delay before and after the ball is shown that varies depending on the ball position, the variation is divided into 12cycle steps to keep the phase of the signal correct. The code for the game control is mostly things to keep the ball and
players within the screen, however it is not as easy as one could think as the program must always take exactly the same number of clock cycles or the TV looses its lock to the color carrier. Keeping track of all flow paths and keeping the timing is the largest problem when generating color signals in software.

The sound generation is very simple; there are two sound channels for outputting sound. The sound is called at the beginning of each scan line and outputs a sinus waveform from ROM for each channel to the audio DA, the position is updated according to the speed variable. The speed is changed according the speed change variable and thereby can be pitched up or down. A kind of bounce sound is created by pitching a ton down quickly when the ball bounces. There is also a timer variable to keep track of how many frames the sound should be active.

Most of the game data of the game is stored as one big chunk to be able to use the program ROM more efficient. This is done by using all the 12 bits and the iread instruction, which makes it possible to store 50% more data than by using retlw, but at the cost of speed. It is hard to use 12bit data efficiently, but to make it easier I chose to separate the game data into one fastmem- and one slowmem-part, where the 8 lower bits of each 12-bit word is the fastmem and the upper 4 bits are the slowmem. Getting one byte from the slowmem requires two iread but the fastmem only requires one. Graphics objects are stored as 4-bit palette values, so is the music, but the font and text strings are all 8-bit values, so it is quite natural to store the 4-bit data in the upper part and the 8-bit data in the lower part.

The software is written to run for both PAL and NTSC with almost the same code, done by making all timing with constants. The constant system selects what TV system to use. In the code I have also prepared timing for PAL-M and PAL-N but they are not tested. It is not possible to generate SECAM color video signals in software with this design, so there is nothing in the code to support it. Note that the frequency which the chip should be clocked depends on your TV-system.
7. Conclusions

It is possible to generate composite color video signals in software, but it is a lot of work and it is only possible in some special cases. NTSC is much more easy to do than PAL when doing the signal in software as phaseshifting is better done in hardware. The main reason for doing video in software is doing it for fun and that it is possible =), this form of video generation has very little commercial value as it takes huge amount of time to generate something with very poor result. Doing software based monochrome signal colored with hardware would give better result, but the best result is of course done with memory mapped graphics outputted with dedicated hardware.
Appendix A: Color test1 source code

;********************************************************************************************
;*  SX COLOR TEST1 (C) Rickard Gunee, 2001                                           *
;******************************************************************************************
;*  This is a test that shows the 12 phases as 12 colored lines.                     *
;*  The video signal is not 100% correct, it will not work on all TV:s, so if      *
;*  your TV can't lock on the color signal or you get strange colors on the       *
;*  screen than your TV probably can't run this game.                              *
;*  This is an open source project and you may use this design and software        *
;*  anyway you like as long as it is non commercial and you refer to the          *
;*  original author with name and link to homepage.                               *
;*  Use this at your own risk, don't blame me if you blow up your tv or kill      *
;*  yourself or anyone else with it.                                              *
;*  For more info about project go to: http://www.rickard.guinee.com/projects      *
;******************************************************************************************

DEVICE SX28,TURBO,STACKX_OPTIONX
RESET start ;goto 'start' on reset
NOEXPAND

SYSTEM_PAL= 1
SYSTEM_PAL_N = 2
SYSTEM_PAL_M = 3
SYSTEM_NTSC = 4

SYSTEM = SYSTEM_PAL ;This line selects TV-system timing to use

IF (SYSTEM = SYSTEM_PAL)
FREQ 53156550

TIME_2U84 EQU 128
TIME_4U55 EQU 239
TIME_27USSEQU 1463
TIME_29USEQU 1574
TIME_64U5 EQU 3405
TIME_TOTALSEQ TIME_64US
TIME_PRESYNC EQU 89
TIME_SYNC EQU 250
TIME_PREBURST EQU 48
TIME_BURSTSEQ 144
TIME_POSTBURST EQU 115

TOT_LINES EQU 304
PRE_LINES EQU 35
POST_LINESEQU 13

PHASESHIFT_MASK EQU 2

ENDIF

IF (SYSTEM = SYSTEM_PAL_M)
FREQ 42907332

TIME_2U84 EQU 103
TIME_4U55 EQU 193
TIME_27USSEQU 1181
TIME_29USEQU 1271
TIME_64U5 EQU 2749
TIME_TOTALSEQ TIME_64US
TIME_PRESYNC EQU 47
TIME_SYNC EQU 202
TIME_PREBURST EQU 39
TIME_BURSTSEQ 144
TIME_POSTBURST EQU 5

TOT_LINES EQU 254
PRE_LINES EQU 35
POST_LINESEQU 13

PHASESHIFT_MASK EQU 2

ENDIF

IF (SYSTEM = SYSTEM_PAL_N)
FREQ 42984672

TIME_2U84 EQU 103
TIME_4U55 EQU 193
TIME_27USSEQU 1181
TIME_29USEQU 1271
TIME_64U5 EQU 2749
TIME_TOTALSEQ TIME_64US
TIME_PRESYNC EQU 47
TIME_SYNC EQU 202
TIME_PREBURST EQU 39
TIME_BURSTSEQ 144
TIME_POSTBURST EQU 5

TOT_LINES EQU 304
PRE_LINES EQU 35
POST_LINESEQU 13

PHASESHIFT_MASK EQU 2

Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.guinee.com/projects
ENDIF

IF (SYSTEM = SYSTEM_NTSC)

    FREQ  42954540

    TIME_2US  EQU  103
    TIME_4US  EQU  193
    TIME_7US5EQU  1181
    TIME_9USEQU  1271
    TIME_64US  EQU  2748
    TIME_TOTALUS  TIME_64US
    TIME_PRESYNC  EQU  47
    TIME_SYNC  EQU  202
    TIME_PREBURST  EQU  39
    TIME_BURSTUSQ  144
    TIME_POSTBURST  EQU  5
    TOTAL_LINES  EQU  254
    PRE_LINES  EQU  30
    POST_LINESQ  13
    PHASESHIFT_MASK  EQU  0

ENDIF

delaytimer1  equ  08h
delaytimer2  equ  09h
temp0  equ  08h
temp1  equ  09h
temp2  equ  0Ah
temp3  equ  08h
temp4  equ  00h
temp5  equ  00h
temp6  equ  00h
temp7  equ  00h
stuff  equ  10h
black  equ  14
neutral  equ  14
frame  equ  0
phaseshiftequ  1

video  equ  RB
audio  equ  RC

TIME_HSXNC= (TIME_PRESYNC + TIME_SYNC + TIME_PREBURST + TIME_BURST + TIME_POSTBURST)
TIME_IMAGE= (TIME_TOTAL - TIME_HSXNC)
VISLINES = (TOTAL_LINES - PRE_LINES - POST_LINES)

;****************************** vout macro ******************************
;* This macro outputs a constant to the video DA *
;****************************** vout macro ******************************

vout  MACRO 1
    mov  w, #1
    mov  video, w
ENDM

;****************************** voutr macro ******************************
;* This macro outputs data from a register to the video DA *
;****************************** voutr macro ******************************

voutr  MACRO 1
    mov  w, \
    mov  video, w
ENDM

;****************************** tnop macro ******************************
;* This macro creates a delay of 3 clock cycles only using *
;* one word of program memory. *
;****************************** tnop macro ******************************

tnop  MACRO 1
    jmp  tnopj
:tnopj
ENDM

;****************************** setphase macro ******************************
;* This is a macro for creating delay that depends on the *
;* contents of w, it adds w to the low part of pc, and adds *
;* nops after the jmp instruction, the number of nops is *
;* specified as a parameter to the function *
;****************************** setphase macro ******************************

setphase  MACRO 1
    jmp  pc
    RET
    nop
ENDM
ENDM

;****************************** delay macro ******************************
;* This is a macro for creating delays by calling the delay *
;* functions, it minimizes the number of program words to max *
;* 4 words. For delays times less than 1017 and longer than 9 *
;* it uses the short delay functions at the cost of 2-3 words *
;* for shorter delays it uses the fixed delays at a cost of 1 *
;* to 3 words, longer delays are done by a call to the short *
;* delay functions followed by a long delay call with a total *
;* cost of 4-6 words of program memory. The macro can handle *

30 Generating color composite video signals in software, written by Rickard Gunée
; delays from 0 to 260k cycles.
; WARNING, no guarantee that this really works correctly for
; all delays as it quite complex and I'm too lazy to test it.

; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

delay MACRO 1
:delbase IF (:delbase & $E00) = (delay9 & $E00)
  IF ((\1)/6) =1
    nop ENDIF
  IF ((\1)/3)=2
    nop ENDIF
  IF ((\1)/3) > 0
    REPT ((\1)/3)
    nop
  ENDIF
ENDIF
ENDIF
ENDIF
IF (\1) > 1028
  mov w,\((\1)-7) >> 2
  page delay_short 0
  call delay_short 0 - (\(\1)-7)*3
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDING
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
Appendix B: Color test2 source code

*******************************************************************************
* SX COLOR TEST2 (C) Rickard Gunde, 2001                                 *
*******************************************************************************
* This is a test that shows all available colors on my SX-based gamesystem. *
* The video signal is not 100% correct, it will not work on all TVs, so if *
* your TV can't lock on the color signal or you get strange colors on the *
* screen then your TV probably can't run this game.                         *
* This is an open source project and you may use this design and software   *
* anyway you like as long as it is non comercial and you refer to the       *
* original author with name and link to homepage.                         *
* Use this at your own risk, don't blame me if you blow up your tv or kill *
* yourself or anyone else with it.                                         *
* For more info about project go to: http://www.rickard.gune.com/projects   *
*******************************************************************************

DEVICE SX28,TURBO,STACKX_OPTIONX
RESET start ;goto 'start' on reset

SYSTEM_PAL= 1
SYSTEM_PAL_M = 2
SYSTEM_PAL_M = 3
SYSTEM_NTS= 4
SYSTEM = SYSTEM_NTS;This line selects TV-system timing to use

IF (SYSTEM = SYSTEM_PAL)
FREQ 53156550
TIME_0US4    EQU  128
TIME_4US5    EQU  239
TIME_27USSEQU       1463
TIME_29USSEQU       1574
TIME_64US        EQU  3405
TIME_TOTALEQU TIME_64US
TIME_PRESYNK     EQU  89
TIME_SYNC        EQU  250
TIME_PREBURST   EQU  48
TIME_BURSTEQU   144
TIME_POSTBURST  EQU  114
TOT_LINES      EQU  304
PRE_LINES       EQU  35
POST_LINESEQU  13
ENDIF

IF (SYSTEM = SYSTEM_PAL_M)
FREQ 42907332
TIME_0US4    EQU  103
TIME_4US5    EQU  193
TIME_27USSEQU       1181
TIME_29USSEQU       1271
TIME_64US        EQU  2749
TIME_TOTALEQU TIME_64US
TIME_PRESYNK     EQU  47
TIME_SYNC        EQU  202
TIME_PREBURST   EQU  39
TIME_BURSTEQU   144
TIME_POSTBURST  EQU  5
TOT_LINES      EQU  254
PRE_LINES       EQU  35
POST_LINESEQU  13
ENDIF

IF (SYSTEM = SYSTEM_PAL_M)
FREQ 42984672
TIME_0US4    EQU  103
TIME_4US5    EQU  193
TIME_27USSEQU       1181
TIME_29USSEQU       1271
TIME_64US        EQU  2749
TIME_TOTALEQU TIME_64US
TIME_PRESYNK     EQU  47
TIME_SYNC        EQU  202
TIME_PREBURST   EQU  39
TIME_BURSTEQU   144
TIME_POSTBURST  EQU  5
TOT_LINES      EQU  304
PRE_LINES       EQU  35
POST_LINESEQU  13
ENDIF

IF (SYSTEM = SYSTEM_NTS)
FREQ 42954540
TIME_0US4    EQU  103
TIME_4US5    EQU  193

Generating color composite video signals in software, written by Rickard Gunde
More info available at: http://www.rickard.gune.com/projects
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects

```assembly
DELAY_TIMES1 EQU 08h
DELAY_TIMES2 EQU 09h
 TEMP6 EQU 08h
 TEMP1 EQU 09h
 TEMP2 EQU 0Ah
 TEMP3 EQU 0Bh
 TEMP4 EQU 0Ch
 TEMP5 EQU 0Dh
 TEMP6 EQU 0Eh
 TEMP7 EQU 0Fh
 BLACK EQU 14
 NEUTRAL EQU 14
 FRAME EQU 0
 VIDEO EQU BB
 AUDIO EQU RC

TIME_HSYNC = (TIME_PRESYNC + TIME_SYNC + TIME_PREBURST + TIME_BURST + TIME_POSTBURST)
TIME_TOTAL = (TIME_TOTAL - TIME_HSYNC)
VISTINES = (TOT_LINES - PRE_LINES - POST_LINES)

; *************** vout macro ***************
; This macro outputs a constant to the video DA

vout MACRO 1
mov w,#\1
mov video,w
ENDM

; *************** voutr macro ***************
; This macro outputs data from a register to the video DA

voutr MACRO 1
mov w,#\1
mov video,w
ENDM

; *************** tnop macro ***************
; This macro creates a delay of 3 clock cycles only using

tnop MACRO 1
jmp :tnop
:tnopj
ENDM

; *************** setphase macro ***************
; This is a macro for creating delay that depends on the
; contents of w, it adds w to the low part of pc, and adds
; one word of program memory

setphase MACRO 1
jmp pcr+w
REPT \1
nop
ENDR
ENDM

; *************** delay macro ***************
; This is a macro for creating delays by calling the delay
; functions, it minimizes the number of program words to max
; 4 words. For delaytimes less than 1017 and longer than 9
; it uses the short delay functions at the cost of 2-3 words
; for shorter delays it uses the fixed delays at a cost of 1
; to 3 words, longer delays are done by a call to the short
; delay functions followed by a long delay call with a total
; cost of 4-6 words of program memory. The macro can handle
; delays from 0 to 260k cycles.
;
; WARNING, no guarantee that this really works correctly for
; all delays as it quite complex and I'm too lazy to test it

delay MACRO 1
:delbase
IF (delbase & $E00) = (delay9 & $E00)
IF (((\1)+6)
IF (((\1)//3)=1
nop
ENDIF
IF (((\1)//3)=2
```
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects

37
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects
Appendix C: Tetris source code

Device SX28,TURBO,STACKX_OPTION
Reset jumpstart ; goto 'start' on reset

DEVICE

SYSTEM_PAL = 1
SYSTEM_PAL_N = 2
SYSTEM_PAL_M = 3
SYSTEM_NTSC = 4

SYSTEM = SYSTEM_PAL ; This line selects TV-system timing to use

IF (SYSTEM = SYSTEM_PAL)

FREQ 53156550

TIME_20US EQU 128
TIME_40US EQU 239
TIME_70USSEQ EQU 1463
TIME_100USSEQ EQU 1574
TIME_64US EQU 3405
TIME_TOTALEQU TIME_64US
TIME_FRESYNC EQU 89
TIME_SYNC EQU 250
TIME_PREBURST EQU 48
TIME_POSTBURST EQU 114
TIME_LEFTGFX EQU 80*12
TIME_RIGHTGFX EQU 40*12
LEFTGFX_BASE EQU 12*10
TOTALINES EQU 304
PRE_LINES EQU 35
POST_LINESSEQ 13

BRICK_WIDTH EQU 7
BLINE_PHASE EQU 5
CAP_BASE EQU 70*12
CAP_PHASE EQU 6
CAP_PHASEDIFF EQU -1
LEFTSCREEN_PHASE EQU 7
RIGHTSCREEN_PHASE EQU 0
GAMEFIELD_PHASE EQU 6

SCORE_BASESEQ 12*6
SCORE_PHASE EQU 9
TEXTEXPLAIN_BASE EQU 12*6
TEXTEXPLAIN_PHASE EQU 1
TEXTEXPLAIN_PHASE EQU 1
TEXTEXPLAIN_PHASE EQU 1
NBLOCK_BASE EQU 12*6
NBLOCK_PHASE EQU 6
GAMEOVER_PHASE EQU 6
GAMEOVER_BASE EQU 17*12

STRO_BASE EQU 38*12
STRO_PHASESEQ 11
STRI_BASE EQU 30*12
STRI_PHASESEQ 7

ENDIF

IF (SYSTEM = SYSTEM_PAL_M)

FREQ 42907332

TIME_20US EQU 103
TIME_40US EQU 193
TIME_70USSEQ EQU 1181
TIME_100USSEQ EQU 1271
TIME_64US EQU 2749
TIME_TOTALEQU TIME_64US
TIME_FRESYNC EQU 47
TIME_SYNC EQU 202
TIME_PREBURST EQU 39
TIME_POSTBURST EQU 5

ENDIF

This is the classic computer game tetris, outputting a color video signal.

The video signal is not 100% correct, it will not work on all TVs, so if
your TV can't lock on the color signal or you get strange colors on the
screen then your TV probably can't run this game.

This is an open source project and you may use this design and software
any way you like as long as it is non commerical and you refer to the
original author with name and link to homepage.

Use this at your own risk, don't blame me if you blow up your tv or kill
yourself or anyone else with it.

For more info about project go to: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects

```
TIME_64US EQU 2748
TIME_TOTALSEQU TIME_64US
TIME_PRESYNC EQU 47
TIME_SYNC EQU 202
TIME_PREBURST EQU 39
TIME_BURSTSEQU 144
TIME_POSTBURST EQU 5
TOT_LINES EQU 254
PRE_LINES EQU 25
POST_LINES EQU 13
TIME_LEFTGFX EQU 80*12
TIME_RIGHTGFX EQU 40*12
LEFTGFX_BASE EQU 12*10
BRICK_WIDTH EQU 5
BLINE_PHASE EQU 3
CAP_BASE EQU 70*12
CAP_PHASE EQU 4
CAP_PHASEDIFF EQU 0
LEFTSCREEN_PHASE EQU 5
RIGHTSCREEN_PHASE EQU 9
GAMEFIELD_PHASE EQU 3
SCORE_BASESEQ 12*6
SCORE_PHASE EQU 8
TEXTNEXT_BASE EQU 12*6
TEXTNEXT_PHASE EQU 6
TEXTSCORE_BASE EQU 12*2
TEXTSCORE_PHASE EQU 0
NBLOCK_BASE EQU 12*6
NBLOCK_PHASE EQU 4
GAMEOVER_PHASE EQU 7
GAMEOVER_BASE EQU 10*12
STRO_BASE EQU 20*12
STRO_PHASESEQ 10
STRI_BASE EQU 13*12
STRI_PHASESEQ 5

ENDIF

delaytimer1 equ 08h
delaytimer2 equ 09h
temp0 equ 08h
temp1 equ 09h
temp2 equ 0ah
temp3 equ 08h
temp4 equ 0ch
temp5 equ 00h
temp6 equ 00h
temp7 equ 0fh
joyup equ RB.7
joydown equ RC.5
joyleft equ RC.6
joyright equ RC.7
joybuttonseq equ RB.6
x equ $10
y equ $11
kind equ $12
angle equ $13
nextkind equ $14
faltimer equ $15
oldjoy equ $16
joytimer equ $17
blockbuff equ $18
mixedbits equ $10
rnd equ $11
gfxcnt equ $12
lincnt equ $13
musictimerseq $14
songpos equ $15
stemp0 equ $1b
pos equ $1c
pos_1 equ $1c
pos_h equ $1b
sfreq equ $1e
sfreq_1 equ $1e
sfreq_h equ $1f
joybutton_old equ oldjoy.2
black equ 14
neutral equ 14
JTIME equ 10
frame equ 7
gameoverbit equ 6
video equ RB
audio equ RC
CAP_SEP EQU (((BRICK_WIDTH + 2)*8*12) + (13*12))
```

G e n e r a t i n g  c o l o r  c o m p o s i t e  v i d e o  s i g n a l s  i n  s o f t w a r e ,  w r i t t e n  b y  R i c k a r d  G u n é e
M o r e  i n f o  a v a i l a b l e  a t :  h t t p : / / w w w . r i c k a r d . g u n é e . c o m / p r o j e c t s

43
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects
mov temp1, #((\1) + gamedata) >> 8) ; 2
mov temp3, #((\1) + gamedata & $FF) ; 2
pcall strout ; 2
inc temp4 ; 1
delay TIME_IMAGE-(1*(W-1)*12) + 44 + 1) - (13) + (14) - (2+2+2+1+4+1)
decsz temp7 ; 2(2)
jmp :bots_1 ; 3
delay 2
pcall haync
ENDM

;************** tnop macro **************
/* This macro creates a delay of 3 clock cycles only using */
/* one word of program memory. */
;****************************************

;tnop MACRO
:tnopf
IF (:tnopf & $111000000000) = (:tnopf+1 & $111000000000)
jmp :tnopf + 1
ELSE
nop
nop
nop
ENDIF
ENDM

;************** setphase macro **************
/* This is a macro for creating delay that depends of the */
/* contents of w, it adds w to the low part of pc, and adds */
/* nops after the jmp instruction, the number of nops is */
/* specified as a parameter to the function */
;****************************************

;setphase MACRO 1
jmp pcrw
REPT \1
nop
ENDR
ENDM

;************** delay macro **************
/* This is a macro for creating delays by calling the delay */
/* functions, it minimizes the number of program words to max */
/* 4 words. For delays times less than 1017 and longer than 9 */
/* it uses the short delay functions at the cost of 2-3 words */
/* for shorter delays it uses the fixed delays at a cost of 1 */
/* to 3 words, longer delays are done by a call to the short */
/* delay functions followed by a long delay call with a total */
/* cost of 4-6 words of program memory. The macro can handle */
/* delays from 0 to 260k cycles. */
/* WARNING, no guarantee that this really works correctly for */
/* all delays as it quite complex and I'm too lazy to test it */
;****************************************

delay MACRO 1
rlenbase
IF (:rlenbase & $E00) = (delay9 & $E00)
IF ((\1)<6)
IF ((\1)//3)=1
nop
ENDIF
IF ((\1)//3)=2
nop
ENDIF
IF ((\1)//3) > 0
REPT ((\1)//3)
tnop
ENDIF
ENDIF
ENDIF
IF ((\1)>=5) AND ((\1)<10)
call delay5 - ((\1)-6)
ENDIF
IF ((\1) > 9) AND ((\1)<1027)
mov w,\1-6)\2
mcall delay_short_0 - ((\1)-6)\3
ENDIF
IF ((\1) > 1026
IF (((\1)-12)//1017)<10
mov w,\1-(\1)-12//1017)\2
mcall delay_short_0 - (((\1)-12)//1017)\3
m mov w,\1-(\1)-12//1017)
ELSE
mov w,\1-(\1)-12//1017)
call delay_short_0 - (((\1)-12)//1017)
mov w,\1-(\1)-12//1017)
ENDIF
mcall delay_long
ENDIF
ELSE
IF ((\1)<7)
IF ((\1)//3)=1
nop
ENDIF
IF ((\1)//3)=2
nop
ENDIF
IF ((\1)//3) > 0
REPT ((\1)//3)
延迟函数

```
delay9     nop    ;1   entrypoint of delay9 that delays 9 clocks
delay8     nop    ;1   entrypoint of delay8 that delays 8 clocks
delay6     retp   ;3   entrypoint of delay6 that delays 6 clocks
delay_short_3 nop    ;1   entrypoint of delay_short_3 that delays 4W + 8
delay_short_2 nop    ;1   entrypoint of delay_short_2 that delays 4W + 7
delay_short_1 nop    ;1   entrypoint of delay_short_1 that delays 4W + 6
delay_short_0 mov temp0, w   ;1   entrypoint of delay_short_0 that delays 4W + 5
delay_long_0 mov temp1, w   ;1   set long time counter from w
delay_long_1 mov w, #251   ;1   set time to delay in short delay
delay_long_3 mov w, #1012  ;1   time to delay in #14/#1017
decz temp0   ;1(2)   decrease counter, mainloop of delay short
jmp delay_short_m   ;3   keep looping until counter is zero
ret   ;3   return back to caller
```

```
readsong mov m, #(MUSIC + gamedata) >> 8);   ;1
mov w, #(MUSIC + gamedata) & SFF   ;2
add w, songpos   ;2
anc   ;2(2)
mov m, #(MUSIC + gamedata) >> 1 + 1   ;1
inc songpos   ;1
ret   ;3
```

```
readfreqtbl mov m, #(FRQ7BL + gamedata) >> 8);   ;1
mov w, #(FRQ7BL + gamedata) & SFF   ;2
add w, temp0   ;2
anc   ;2(2)
mov m, #(FRQ7BL + gamedata) >> 1 + 1   ;1
inc temp0   ;1
ret   ;3
```

```
readjoy1 mov w, BC   ;1
and w, #11110000   ;1
mov temp0,w   ;1
mov w, <RB   ;1
mov w, #00000100   ;1
or w, temp0   ;1
ret   ;3
```

```
readjoy0 mov w, BC   ;1
and w, #11110000   ;1
mov temp0,w   ;1
mov w, <RB   ;1
mov w, #00000100   ;1
or w, temp0   ;1
ret   ;3
```

```
readjoy1    mov w, BC   ;1
and w, #11110000   ;1
mov temp0,w   ;1
mov w, <RB   ;1
mov w, #00000100   ;1
or w, temp0   ;1
ret   ;3
```

```
readjoy0    mov w, BC   ;1
and w, #11110000   ;1
mov temp0,w   ;1
mov w, <RB   ;1
mov w, #00000100   ;1
or w, temp0   ;1
ret   ;3
```

```
; ***************** delay functions ***************
;* Different delay functions to be able to create long delays *
;* using as few bytes of program memory as possible *
;* These functions are required by the delay macro *
;* delays with exact clock count uses no registers *
;* long delays use temp0 and temp1 *
;***************** delay functions *******************
```

```
PLAYCOPl
ENDR
ENDM
```

```
; ********************* playcole **************************************************
;* Reads joy1 bits from RC and RB and combining them to w *
;* temp register 0 used *
;* clocks: 12 unfortunate delays *
;******************************************************
```

```
; ********************* readjoy0 **************************************************
;* Outputs data from memory to video output *
```
; * number of clocks: w'12 + 7 + 1 *
; * temp register 0 used *
;*******************************************************************************

mento video mov temp0, w ;1 set pixelcounter

mv w, ind ;1 get lower level byte from mem
mv video, w ;1 send to video output
setb fsr.5 ;1 select upper bank
mov w, ind ;1 get upper level byte from mem
inc fsr ;1 point at next pixel
clrb fsr.5 ;1 select lower bank
nop ;1
mov video, w ;1 send to video output
decsz temp0 ;1(2) decrease pixel counter
jmp mtv10 ;3 keep looping until all pixels are done
vout BLACK ;2 set black color
rewt ;3 get out here

;*******************************************************************************

; ******************************************************************************
; * outputs data from memory to video output *
; * number of clocks: w'31 + 5 + 1 *
; * temp0 = bitmap rom-pointer bit 0..7 *
; * temp1 = bitmap rom-pointer bit 8..11 *
; * temp2 = palette rom-pointer bit 0..7 *
; * fsr = points to memory where to store graphics *
; * Note: bits 8..11 of palette pointer is in the constant *
; * called PALETTE_PAGE, all palettes should be placed within *
; * this page. far, temp0 and temp1 are modified *
;******************************************************************************

sets graphics mov temp3, w ;1 set pixelcounter

mov w, temp1 ;2 set page
mov w, temp0 ;1 get image pointer
iread ;4 read pixel data from rom
mov w, m ;1 get slowness nibble
add w, temp2 ;1 select palette, assuming all palettes within the same page
mov w, PALETTE_PAGE ;1 select page
iread ;4 read palette
mov ind, w ;1 remember first level
setb fsr.5 ;1 select second level bank
and w, #0FF ;1 mask out two upper bits
mov ind, w ;1 store second level two upper bits
rr ind ;1 move upper bits into correct position (1/2)
rr ind ;1 move upper bits into correct position (2/2)
mov w, m ;1 get second level lower nibble
or ind, w ;1 store second level lower nibble
crb fsr.5 ;1 get back to first level bank
inc fsr ;1 point at next pixel memory position
inc temp0 ;1 point at next nibble
isz ;1(2)
inc temp1 ;1 if page overflow, go to next page
decsz temp3 ;1(2) decrease pixel counter
jmp sg10 ;3 keep looping until all pixels are done
rewt ;3 get out here

; ******************************************************************************

; ******************************************************************************
; * get compressed brick data *
; * clock cycles: 20 *
;******************************************************************************

blocks and w, #91111
add pc, w ;3
rewt S50 ;3
rewt 044 ;3
rewt S05 ;3
rewt S0C ;3
rewt S0D ;3
rewt S0E ;3
rewt S0F ;3
rewt S44 ;3
rewt S00 ;3
rewt S50 ;3
rewt S0C ;3
rewt S0D ;3
rewt S0E ;3
rewt S0F ;3
rewt S6C ;3
rewt S6D ;3
rewt S6E ;3
rewt S6F ;3

; ******************************************************************************

; ******************************************************************************
; * Check if out of bounds, calculate address to brick and *
; * mask to unwanted nibble *
; * clock cycles: 20 *
; * uses temp0..temp3 *
; * temp0 = x-position *
; * temp1 = y-position *
; * returns out of bounds as a nonzero value in temp3 *
; * returns bitmask in w *
;******************************************************************************

;20 clocks brickposcheck mov w, #S9 ;1 get illegal x-positions
and w, temp0 ;1 mask out illegal x-bits for x-position
mov temp3, w ;1 store illegal bits for later
mov w, #FFFF ;1
mov temp1, w ;1 get illegal y-positions
or temp3, w ;1 combine illegal x- and y- bits and store in temp3
mov w, #4S9 ;1 get x-position and swap nibbles
and w, S86 ;1 only keep former bit 1 and 2
add w, temp1 ;1 get y-position
or w, #S9 ;1 set bit 4 and 7 to get correct address
mov fsr, w ;1 set file select register to calculated pointer
mov w, #S9 ;1 get bitmask for left brick
shb temp0, 0 ;1(2) check if x-pos is odd

Generating color composite video signals in software, written by Rickard Gùnee
More info available at: http://www.rickard.gunee.com/projects
mov w,#50F ;1 yes, get bitmask for right brick instead
ret ;3 get out here

;;;;;;;;;; setbrick ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* Sets a brick on the position temp0, temp1 with color temp2
* clock cycles: 34
* uses temp0, temp3
* temp0 = x-position
* temp1 = y-position
* temp2 = color

setbrick call brickposcheck ;20 calc address, check out of bounds and get bitmask
test temp3 ;1 out of bounds ? (1/2)
sz ;1(2) out of bounds ? (2/2)
jmp delay9 ;3 yes, out of bounds, do delayed return
and ind,w ;1 clear wanted nibble
mov w,temp2 ;1 get color
sub temp0,0 ;1 check if x is odd
mov w,<temp2 ;1 yes, get color with swapped nibbles instead
or ind,w ;1 set color
ret ;3 get out here

;;;;;;;;;; checkbrick ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* Checks if there is a brick on the position temp0, temp1
* returns nonzero value of there is a brick and zero if the
* position is clear
* clock cycles: 33
* uses temp0, temp3
* temp0 = x-position
* temp1 = y-position

checkbrick call brickposcheck ;20
mov temp3 ;1 out of bounds ? (1/2)
sz ;1(2) out of bounds ? (2/2)
jmp delay8 ;3 yes, out of bounds, do delayed return
and w,ind ;1 invert bits to get wanted nibble
mov w,ind ;1 get wanted nibble from playfield
ret ;3 get out here

;;;;;;;;;; checksetsel ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* If bit 0 in temp6 is set then the the block in blockbuf
* is drawn in the playfield on position x,y with color temp2
* using 221 clocks
* If bit 0 in temp6 is clear then the the block in blockbuf
* is checked for collisions on position x,y in the playfield
* returning result in temp7 using 217 clocks
* tempregs 0..7 are used
* The reason of combining these two operations is that they
* are very similar, combining them will save program mem
* checkblock calls checksetsel with temp6.0 cleared (221+1)
* setblock calls checksetsel with temp6.0 set (223+1)

checkblocksel temp6.0 ;1 set checking (clear setting)
skip ;2 don't set setting
setblock setb temp6.0 ;1 set setting
checksetsel clr temp7 ;1 clear result
mov temp4,#blockbuf ;2 point temp4 at block buffer
mov temp5,#4 ;2 each block has 4 bricks
setblock_lmov fsr,temp4 ;2 set file select register to block buffer pointer
mov w,x ;1 get block base x-position
add w,ind ;1 add relative brick position
mov temp0,w ;1 store brick x-position
inc fsr ;1 point at next buffer x-position
mov w,y ;1 get block base y-position
add w,ind ;1 add relative brick y-position
mov temp1,w ;1 store brick y-position
sub temp0,0 ;1(2)
call checkbrick;33
shl temp6,0 ;1(2)
call checkbrick ;34
or temp7,w ;1 store result from check
call decsz temp5 ;1(2) decrease brick counter
jmp setblock_l ;3 keep loopin until all 4 bris are out
bank 500
retp ;3 get out here

jumpstart pjmp start

;;;;;;;;;; simplesel ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* outputs w color cycles at (almost) maximum amplitude
* Clocks: w[12 + 2] + 1

simplesel mov temp2,w ;1
mov temp0,#63 ;2
mov temp1,#black ;2 skip ;2

;;;;;;;;;; simplesel ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* outputs w color cycles
* Clocks: w[12 + 5] + 1

simplesel mov temp2,w ;1
mov temp0,#63 ;2
mov temp1,#black ;2 skip ;2

;;;;;;;;;; simplecolor ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* outputs w color cycles
* Clocks: w[12 + 2] + 1

simplecolor mov temp2,w ;1
mov temp0,#63 ;2
mov temp1,#black ;2 skip ;2

simplecolor_l mov temp0 ;2 set first level
delay 4 ;4 delay to get j12cycle loop

Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects
;*********************** updatemusic ***********************
;* Music player, called once per frame, playing "Karboinka" *
;* From rom, each position of the tune is stored as a note *
;* Followed by a length. The length is multiplied with the *
;* gamespeed making it play faster with the speed of the game *
;* Each note is translated to a "frequency" from a table in *
;* Rom, notes are separated with a short (2 frames) pause *
;* Clocks: 130 + 1 *
;***********************

updatemusic
bank $20 ;1
dec sz ;1
jmp musictimer;
if not zero, don't update note info
jmp musicpause

bank $60 ;1
mov w,sfreq_l ;1
or w,sfreq_h ;1
is frequency zero ?
sz ;1
yes, don't make a pause
jmp musicpause;3
no, make a pause (i.e. set freq to zero for two frames)

bank $20 ;1
call readsong ;17 get next nibble of the song from rom, the note (i.e position
in frequency table
bank $60 ;1
mov temp0,w ;1
temp0 = 1*freqtablepos
add temp0,w ;1
temp0 = 2*freqtablepos
add temp0,w ;1
temp0 = 3*freqtablepos

call readsreqtbl ;16 get bit 0..3 from rom
mov sreq_l,w ;1
store in high byte of frequency
inc temp0 ;1
point at next position in rom

inc temp0 ;1
point at next position in rom
call readsreqtbl ;16 get bit 4..7 from rom
mov sreq_h,w ;1
store in high byte of frequency

bank $20 ;1
mov temp1,$12 ;1
temp1 = 11
sub temp1,(SCORE+1) ;2
temp1 = 11-speed = note baseline

call readsong ;17 get next nibble of the song from rom, the notelenth
mov temp0,w ;1
put notelenth in temp0 to be able to do tests
test temp0 ;1
update flags according to notelenth
clc ;1
clear carry to prevent a set carrybit pollution of the

speed multiplier
sz ;1
check if notelenth is larger than zero
s1 temp1 ;1
temp1 = (11-speed)*2
shl temp0,1 ;2(4) check if notelenth is 1
add temp1,w ;1
temp1 = speed*2 + speed/2 = speed * 2.5
mov temp1,w ;1
w = temp1 = lengthmultipler (1, 2 or 2.5)
mov musictimer,w ;1
set notelenth to (11-speed)*lengthmultipler

mov w,songpos ;1
get song position
xor w,#104 ;1
xor with songlength

snz ;1
if result is zero then we have reached the end of the song
jmp main ;4 get back to main

musicnochnote
delay 130-12 ; delay to keep timing when no change of frequency

jmp main ;4 get back to main

musicpause
bank $60
delay 3

musicpauseclr
sfreq_l ;1 clear low byte of sfreq
clr sfreq_h ;1 clear high byte of frequency
clr pos_l ;1
clr pos_h ;1

bank $20 ;1
mov musictimer,#2 ;2 pause is for two cycles
delay 130-25 ; delay to keep timing when setting pause

jmp main ;4 get back to main

;*********************** vrealsound ***********************
;* vrealsound is called from vsync and calls realsound every *
;* second vsync cycle as vsync is called at twice the speed *
;* as sync, so vrealsound is dependent of realsound *
;* Clocks: 39 + 1 *
;***********************

vrealsounds
bank $60 ;1

mov temp2.0 ;1(2)
jmp realsound ;135
delay 35-1-3
ret

vrealsoundsdb
mov w,for
bank $60
mov temp0,w
mov m,((INTABLE+4)+2) ;1 point at current page for sintable
add pos, #3
update sintable position according to
speed and pos_h,#31
mov temp0,pos_h

Generating color composite video signals in software, written by Rickard Guneé
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects

53
Generating color composite video signals in software, written by Rickard Guñée
More info available at: http://www.rickard.gunee.com/projects
call brickcolortable ; get phase and amplitude of color
mov ind,w  ; update phase and color in video buffer
inc temp1  ; update real video buffer pointer
inc temp4  ; update real video buffer pointer (again)
add temp0,$220 ; move playfield pointer one step right
decsz temp3  ; decrease field read counter
jmp fieldread_l ; loop until entire line is read
clr fSr  ; as the fSr was manipulated, reset it back first

delay ( ((BRICK_WIDTH + 2) *8*12) +5*1+ 7*1111 ) -163
pcall rightgfx
delay TIME_IMAGE - TIME_RIGHTGFX - TIME_LEFTGFX - ( ((BRICK_WIDTH + 2) *8*12) +5*1+ 7*1111 ) - 2
mov temp6,BRICK_LINES

page

line1
call haync  ;643 11
pcall leftgfx
delay GAMEFIELD_PHASE
bank $20  ;
mov w,temp7  ; get blockline
and w,$81110  ; dont care about even or odd
xor w,$81000  ; check if line is 6 or 7
az  ; (2)
jmp nogameover  ;3 if not don't show gameover
page showgameover  ;1 prepare page for jmp
sub mixbits.gameoverbit  ;(2) check if game is over
jmp showgameover  ;3 if so, show game over
lkip  ;(2)

nogameoverjmp nogameover2  ;3

mov fSr,$VIDEO_BUFFER+8  ;2
mov temp0,BRICK_WIDTH  ;2
mov temp1,#black  ;2
mov w,#8
pcall outputcol ;((BRICK_WIDTH + 2) *8*12)+5+1

showgameoverret
delay GAMEFIELD_PHASE
pcall rightgfx
delay TIME_IMAGE - TIME_RIGHTGFX - TIME_LEFTGFX - ( ((BRICK_WIDTH + 2) *8*12) +5+1+ 7*1111 ) - 4

decsz temp6  ;(2)
jmp line1  ;3
delay 2

page field1

decsz temp7  ;
jmp field1  ;
delay 2

call haync

pjmp nextmain

ILogger

*************** outputcol ***************
* shows w number of colorfields, each BLOCKWIDTH cycles wide *
* (this is the thing that shows the colors in the playfield) *
* number of clocks: (temp0 + 2) * w * 12 + 5 *
* used tempregs: 0..5 *
* input: *
* w = number of clockfields *
* temp0 = fieldlength, must be odd (phase lost even) *
* temp1 = neutral level *
* fSr = pointer to field data (contents is destroyed) *
* output: *
* none *
* local use of tempregs: *
* temp2 = use as color loop counter *
* temp3 = used as field loop counter *
* temp4 = used as temp storage for intensity calculations *
* temp5 = used as temp storage for phase *
* comments: *
* This routine is optimized to get as short gaps between *
* fields as possible, these optimizations assumes some *
* limitations of the input data to be able to get the gap *
* down to only 2 color cycles (24 clocks) *
* Field color is stored as bytes, where each byte has *
* Bit 0..2 as phase bit and bit 3..6 is intensity, note *
* that this means that there is and overlap of phase and *
* intensity *
* Remaining phase = 7-phase = 7-(phase) = 7+(phase+1) = *
* -1+(phase+1) = /phase + /phase xor 7 (assuming 3 bit calc) *
* this calculation (xor by 7) is done in the inner loop as *
* it was the only place where there was free clock cycles *

Generating color composite video signals in software, written by Rickard Gunèe
More info available at: http://www.rickard.gunee.com/projects

55
;; so to get a correct result the inner loop needs to be
;;;; executed an odd number of times, which makes the field-
;;;; length required to be an odd number or else phase will
;;;; be lost.
;;;; The x in the Phase comment field is the phase value read
;;;; it can be values 0..6
;/***********************************************************************************/

outputcol mov temp3,w ;1 5 set field counter

ocoll mov w,ind ;1 10 get phase and intensity and w,$7 ;1 11 mask out phase

setphase7 mov temp5,w ;1 10-x remember phase for later

mov temp2,temp0 ;2 11-x set color loop counter

xor temp5,$7 ;2 3-x invert all bits in phase

mov w,>ind ;1 5-x get color and phase shifted right one step

mov temp4,w ;1 6-x store shifted value in a tempreg

mov w,>$amp ;1 7-x shift value one more step right

mov video,w ;1 8-x and set video output to the intensity
decsz temp2 ;2(2) 9-x decrease color cycle counter

jmp ocolx3 ;3 10-x and loop until all color cycles are done

voutr temp1 ;2 11-x set neutral level

mov w,temp5 ;1 1-x get remaining phase = 7-phase = phase xor 7

setphase7 mov temp3, @ ;3+x 2-x set remaining phase to return to original phase

inc fsr ;1 5-x+a point at next field byte
decsz temp3 ;2(2) 6 decrease field cycle counter

jmp ocolx3 ;3 7 and loop until all fields are done

swip ;3 8 return back home

/***************************************************************************/

leftgfx ****************************

/leftgfx at left side of playfield

***************************************************************************/

leftgfx delay LEFTGFX_BASE

bank $20 ;1 select bank $20 to be able to read linecount

mov w,<linecnt ;1 get line number with swapped nibbles

and w,$FF ;1 mask out most significant nibble of linecount to get section

number

jmp grow ;3 jump to correct section of 16 lines

jmp teextnext_line ;3 text "NEXT"

jmp nextblock_line ;3 bricks preview

jmp nextblock_line ;3 bricks preview

jmp nextblock_line ;3 bricks preview

jmp black_line;3 black lines between preview and points text

jmp teextscore_line ;3 text "POINTS"

jmp showpoints_line ;3 display points

jmp black_line;3 black lines at the bottom

jmp black_line;3 black lines at the bottom

jmp black_line;3 black lines at the bottom

jmp black_line;3 black lines at the bottom

jmp black_line;3 black lines at the bottom

black_line1 delay TIME_LEFTGFX - LEFTSCREEN_PHASE - LEFTGFX_BASE - 9 - 4 - 3 - 3 - 104

showscrew mov fsr,VIDEO_BUFFER ;2

mov w,8 ;1

pcall mentvideo;104 output left screw graphics
delay LEFTSCREEN_PHASE

swip

/***************************************************************************/

rightgfx ****************************

/rightgfx at right side of playfield

***************************************************************************/

rightgfx delay 11-RIGHTGFX_PHASE

mov fsr,VIDEO_BUFFER ;2

mov w,8 ;1

pcall mentvideo;104 output right screw graphics

bank $20 ;1

inc linecnt ;1 update linecounter

mov temp1,(#(gamedata + SCREEN) >> 8) ;2 set page of graphics

mov w,<linecnt ;1 get linenumber with swapped nibbles

and w,6787 ;1 rehouse unwanted bits to get (line88)*16

mov temp0,w ;1 store in temp0
cle ;1 clear carry

rr temp0 ;1 rotate right to get (line88)*8

add temp0,(#(gamedata + SCREEN) & $FF) ;2 add graphics base

inc temp1 ;1(2) check for page overflow

inc temp1 ;1 point at next page

mov temp2,(#(PALETTE_BOW + gamedata2) & $FF) ;2 point at correct palette

mov fsr,VIDEO_BUFFER ;2 point at video buffer position where to store graphics

mov w,8 ;1

pcall setgraphics ;254 translate graphics into
delay TIME_RIGHTGFX + RIGHTSCREEN_PHASE -11 - 104 - 21 - 254 - 4 - 3

swip ;3

/******************************************************************************/

strout ****************************

/output characters from string in rom using a font in rom

/ temp0 used as character temp storage

/ temp2 used as character counter

/ temp3 = pointer to string

/ temp4 = line (0..7) + FONT_BASE

/temp0 = length

/ clocks: 8 * 12 * w + 44 + 1

strout cl

Generating color composite video signals in software, written by Rickard Guneé

More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects

strout_l vout black ;2 pixel three to seven
delay 2 ;2
rr temp0 ;1
snc ;1(2)
mov w,#53 ;1
decsz temp2 ;1(2)
jmp strout_l ;3
strout mov m,temp1 ;2 set character page
vout black ;2 pixel one starts here
mov w,temp3 ;1 get pointer to character
iread ;4 read one character
add w,temp4 ;1 update according to line and fontbase
mov m,((gamedata + FONT) >> 8) ;1 set font-page
mov temp0,w ;1 store character pixels in temp0
mov w,#53 ;1
inc temp3 ;1 (2)
mov video,w ;1
snz ;1(2)
inc temp1 ;1
delay 2
mov temp2,#5 ;2
rr temp0 ;1
snc ;1(2)
mov w,#53 ;1
mov video,w ;1
decsz temp5 ;1(2)
jmp strout_cl ;3
vout black ;2
retp ;3

;*************** inpoints ***************
;/* add one point to score */
;/* clocks: 59+1 */
;*******************************
inpoints mov fsr,#SCORE+3 ;2
mov temp0,#4 ;2
inpoints_l inc ind ;1
mov w,#%1010 ;1
xor sz ;1(2)
jmp nocarry_l ;3
clr ind
dec fsr ;1
decsz temp0 ;1(2)
jmp inpoints_l ;3
delay 7 ;7
retp ;3
nocarry_l delay ?
decsz temp0 ;1(2)
jmp nocarry_l ;3
retp

;*************** tempoints ***************
;/* add ten points to score */
;/* note: this routine requires inpoints */
;*******************************
tempoints mov fsr,#SCORE+2 ;2
mov temp0,#3 ;2
jmp inpoints_l ;3

;*************** textlines ***************
;/* routine to output line with text in leftgraphics field */
;/* clocks: 53+7 */
;*******************************
textlines mov w,linecnt ;1 get line number
and w,#%1010 ;1(2) check bit 2,3
sz ;1 if bits are zero, don't do next test
togg bit 2,3 to check if values are %1
snz ;1(2) if bits are %00 or %11
jmp notext ;3 then lines should be empty
mov temp4,((gamedata + FONT) & $ff) - 4 ;2 set temp4 to fontbase - 4 (compensating
line starting at 4)
mov temp4,#((gamedata + FONT) & $ff) - 4 ;2
add temp4,w ;1 update pointer according to line number
mov temp5,#6 ;2 always output 6 characters
jmp strout ;525 output text
notext delay 526 ;523 delay to keep timing correct if no text
ret ;3 get back to left graphics and show screw

;*************** makevect ***************
;/* create graphics for next block to be used in leftgraphics */
/* clocks: 183 + */

;*******************************************************

xsel and w,#3 ;1 remove unwanted bits
jmp pcw ;3 select correct return value
retw 2 ;3 bit 1
retw 4 ;3 bit 2
retw 8 ;3 bit 3
retw 1 ;3 bit 0

ysel and w,#3
jmp pcw
retw (NEXTGFX & $7F) | $80
retw (NEXTGFX+1 & $7F)
retw (NEXTGFX+2 & $7F)
retw (NEXTGFX+3 & $7F)

makenext bank NEXTGFX ;1
clr NEXTGFX ;1
cir NEXTGFX+1
bank 500 ;1
cir
mov w,<nextkind ;1
pca blocks ;10
mov temp0,w ;1
mov w,<nextkind ;1
or w,#1 ;1
call Blocks ;9
call makenext_l ;1
mov temp1,w ;1
mov temp3,#4 ;2

makenext_lmov w,temp0 ;1
call xsel ;10
mov temp2,w ;1
mov w,temp1 ;10
call ysel ;10
mov fsr,w ;10
sub temp2 ;10
reap temp2 ;10
clr fsr7 ;10(2)
crtemp2 ;10
ot ind,temp2 ;2
rr temp0 ;1
rr temp0 ;1
rr temp1 ;1
rr temp1 ;1
decz temp3 ;1
jmp makenext_l ;3

;********************** textscore_line **********************
/* handles field with text "next" in leftgraphics */
;********************** textnext_line **********************
/* handles field with text "score" in leftgraphics */

;********************** textpoints_line **********************
/* handles field with score in leftgraphics */

;********************** textnextr_line **********************
/* handles field with text "next" in leftgraphics */

;********************** textpoints_line **********************
/* handles field with score in leftgraphics */
Generating color composite video signals in software, written by Rickard Gürnberg
More info available at: http://www.rickard.gunee.com/projects

leavel in w

mov w, #53
;1(2) check if bit set, if not keep black

mov w, video, w
;1 else if bit set set digit intensity

mov selected level to video output
dec temp2
;1(2) decrease pixel counter

jmp stroutp_1
;2 loop until all pixels are done

vout black
;2 set video level to black
dec temp1
;1(2) decrease digit counter

jmp stroutp_1
;3 loop until all digits are done

delay TIME_LEFTFX - LEFTSCREW PHASE - LEFTFX_BASE - 9 - 4 - 3 - 3 - 104 - 3 - 487 -

SCORE_BASE+SCORE PHASE

jmp showcresw
;3 get back to left graphics and show screw

;*********************** nextblock_line ***********************

;* handles fields showing next block

;**************************

nextblock_line delay NBLOCK_BASE-NBLOCK_PHASE
mov w, gfnct
and w, #50C
;j1 get gfnctcounter

jmp NBNEXTFX
;j1 check if line is 0 .. 2 of the 12-line

brick

anz
;j1(2) if not, get on with the brick drawing

jmp nonextg
;j3 if one of the first lines, then it should be black

mov temp0, gfnct
;j2

bank NEXTGF
;j1

mov w, NEXTGF
;j1 get graphics for next block

snb temp0, 5
;j1(2) check if brickline second half

mov w, NEXTGF+1
;j1 yes, it was, get next graphics

mov temp0, 4
;j1 store in temp1

snb temp1
;j1(2) check if brickline is odd

swap temp1
;j1 yes, swap nibbles

bank $00
;j1 set bank for next kind

mov w, +nextkind
;j1 get next kind of block

and w, $#00001111
;j1

pall brickcolorable
;j10 translate kind to phase and amplitude

page next10
;j10 brickcolorable destroy page register, restore it

mov temp2, w
;j1 store phase and amplitude in temp2

mov temp3, #4
;j2 we have 4 brick positions to convert

mov gfr, #VIDEO_BUFFER + $28
;j2 point at beginning of buffer

next10
mov w, temp2
;j1 get phas and amplitude of next block

rr temp1
;j1 rotate block data to get next bit

ac
;j1(2) check if bit was set

mov w, # (black) <2
;j1 if not set black intensity

mov ind, w
;j1 store phase and color in buffer

inc fsr
;j1 point at next buffer position

dec temp3
;j1(2) decrease brick counter

jmp next10
;j3 keep looping until all bricks are done

mov gfr, #VIDEO_BUFFER + $28
;j2 point at beginning of buffer

mov temp0, #1
;j2 each brick is 7 cycles

mov temp1, #black
;j1 we have 4 brick positions

pall outputcol
;j438 output colors to video output

jmp donenextg
;j3

donenextg bank $20
;508 empty line, delay to keep timing

inc gfnct
;j1 increase counter

mov w, gfnct
;j1 get counter value

and w, #3
;j1 check bit 0 and bit 1

az
;j1(2) if not zero upper part was not increased

swap gfnct, 0
;j1 if zero, jump to one by setting bit 0

delay TIME_LEFTFX - LEFTSCREW PHASE - LEFTFX_BASE - 9 - 4 - 3 - 3 - 104 - 3 - 524 -

NBLOCK_BASE+NBLOCK PHASE
;j3 delay to make points section 551 cycles

jmp showcresw
;j3 get back to left graphics and show screw

;******************** showgameover ********************

;* show "GAME OVER" text in playfield

;**************************

showgameover delay GAMEOVER_BASE-GAMEOVER PHASE

mov temp3, # (STR5 + gamedata) & $FF
;j2 set lower pointer to "Game" text

mov temp4, # (STR4 + gamedata) >> 8
;j2 set upper pointer to "Ove" text

mov w, # (STR4 + gamedata) & $FF
;j1 get lower address to "Over"

mov temp7, 0
;j1(2) check if section is odd

mov temp3, #1
;j1 odd line, set lower pointer to "Over"

mov temp4, # (gamedata + FONT -1 +8) & $FF
;j2 set temp4 to fontbase - 4 (compensating

line starting at 4)

mov w, temp6
;j1 get linecounter

and w, #7
;j1 get last significant nibble

(sentence line)

snb temp6, 3
;j1(2) check if linear larger than 7

jmp emptyoverl
;j3 if found, skip it

sub temp4, w
;j1 update pointer according to line number

mov temp5, #4
;j1 always output 4 characters
call strout
delay [(BRICK_WIDTH-2)*8+12]+35-444-26|GAMEOVER PHASE-GAMEOVER_BASE
;j429 output text

p, jmp showgameoverret
;j4 get back to mainloop

emptyoverl delay [(BRICK_WIDTH-2)*8+12]+35-15+6|GAMEOVER PHASE-GAMEOVER_BASE
;j4 get back to mainloop

gameover pcall hsync
bank $00
Appendix D: Pong source code

;****************************************************************************************
;SX-PONG © Rickard Gunned, 2001
;****************************************************************************************
;This is the classical videogame pong, outputing a color video signal in
;software using a couple of resistors.
The video signal is not 100% correct, it will not work on all TVs, so if
your tv can't lock on the color signal or you get strange colors on the
screen than your TV probably can't run this game.
This is an open source project and you may use this design and software
anyway you like as long as it is non comercial and you refer to the
original author with name and link to homepage.
Use this at your own risk, don't blame me if you blow up your tv or kill
yourself or anyone else with it.
For more info about project go to: http://www.rickard.gunee.com/projects
;****************************************************************************************
DEVICE SX28,TURBO,STACKX_OPTIONX
RESET jumpstart ; goto start on reset
NOEXPAND
SYSTEM_PAL= 1
SYSTEM_PAL_N = 2
SYSTEM_PAL_M = 3
SYSTEM_NTSC = 4
SYSTEM = SYSTEM_PAL ; *** This line selects TV-system timing to use ***
delaytimer1 equ 08h
delaytimer2 equ 09h
temp0 equ 08h
temp1 equ 09h
temp2 equ 0Ah
temp3 equ 08h
temp4 equ 0Ch
temp5 equ 00h
temp6 equ 00h
temp7 equ 0Fh
joy equ RC
joyup equ RB.7
joydown equ RC.5
joyleft equ RC.6
joyright equ RC.7
joybutton equ RB.6
joyup equ RA.2
joydown equ RA.3
joyleft equ RA.0
joyright equ RA.1
joybutton equ RC.7
y1 equ $10
y2 equ $11
mixedbits equ $12
ballx equ $13
ballx_l equ $13
ballx_h equ $14
gamekind equ $13
bally equ $15
bally_l equ $15
bally_h equ $16
ballx_speed equ $17
ballx_speed_l equ $17
ballx_speed_h equ $18
bally_speed equ $19
bally_speed_l equ $19
bally_speed_h equ $1A
p1 equ $1B
p2 equ $1C
state equ $1D
oldj1 equ $1E
oldj2 equ $1F
soundtemp equ $10
soundtemp+ equ $11
wave ypos equ $12
wave ypos_l equ $12
wave ypos_h equ $13
wave speed equ $14
wave speed_l equ $14
wave speed_h equ $15
wave speed d equ $16
Generating color composite video signals in software, written by Rickard Gunéé
More info available at: http://www.rickard.gunee.com/projects

wave1speeddif_l equ $16
wave1speeddif_h equ $17
wave1timer equ $18
wave2pos equ $19
wave2pos_leq equ $19
wave2pos_heq equ $1A
wave2speed equ $1B
wave2speed_l equ $1B
wave2speeddif equ $1B
wave2speeddif_l equ $1D
wave2speeddif_h equ $1E
wave2timer equ $1F
black equ 14
neutral equ 14

VIDEO_BUFFER equ $F0

TIME equ 10

frame equ 0
line equ 1
gameoverbit equ 2
video equ RB
audio equ RC
joyup_old equ oldj2.7
joydown_old equ oldj1.5
joyleft_old equ oldj1.6
joyright_old equ oldj1.7
joybutton_old equ oldj2.6
joyup_old equ oldj1.2
joydown_old equ oldj1.3
joyleft_old equ oldj1.1
joyright_old equ oldj1.0
joybutton_old equ oldj1.7

IF (SYSTEM = SYSTEM_PAL)
FREQ 53156550
TIME_2US equ 128
TIME_4US equ 239
TIME_7US equ 1463
TIME_12US equ 1574
TIME_44US equ 3405
TIME_TOTAL equ TIME_44US
TIME_PREFSYNC equ 89
TIME_SYNC equ 250
TIME_PREFRST equ 48
TIME_BURST equ 144
TIME_POST_BURST equ 112
LEFT_SPACE equ 120
RIGHT_SPACE equ 144
TOTAL_LINES equ 304
PRE_LINES equ 35
POST_LINES equ 19
LEFTPAD_PHASE equ 1
RIGHTPAD_PHASE equ 10
BALL_PHASE equ 4
LEFTSCORE equ (12*20)
LEFTSCORE equ 9
RIGHTSCORE equ (12*18)
RIGHTSCORE equ 4
TEXT_BASE equ (12*87)
TEXT_PHASE equ 8
TEXT_BASE equ (12*30)
TEXT_PHASE equ 8
TEXTLINE_PHASE equ 6
PADMID_PHASE equ 1
PAMBEND_PHASE equ (12*65)
WINTEXT_PHASE equ 7
INITTEXT1 equ (12*92)
INITTEXT1 equ 11
INITTEXT2 equ (12*56)
INITTEXT2 equ 11
INITTEXT3 equ (12*57)
INITTEXT3 equ 5
INITTEXT4 equ (12*57)
INITTEXT4 equ 5
INITTEXT5 equ (12*28)
INITTEXT5 equ 8
ENDIF

IF (SYSTEM = SYSTEM_PAL_M)
FREQ 42907332
TIME_2US equ 103
TIME_4US equ 193
TIME_7US equ 1181
TIME_12US equ 1271
TIME_44US equ 2749
TIME_TOTAL equ TIME_44US
TIME_PREFSYNC equ 47
TIME_SYNC equ 202
TIME_PREFRST equ 39
TIME_BURST EQU 144
TIME_POSTBURST EQU 5
TOT_LINES EQU 254
PRE_LINES EQU 35
POST_LINES EQU 19
LEFT_SPACE EQU (12*12)
RIGHT_SPACE EQU (12*7)
LEFT_PAD_PHASE EQU 1
RIGHT_PAD_PHASE EQU 10
BALL_PHASE EQU 4
LEFTSCORE_BASE EQU (12*20)
LEFTSCORE_PHASE EQU 9
RIGHTSCORE_BASE EQU (12*15)
RIGHTSCORE_PHASE EQU 4
TTXT_BASE EQU (12*65)
TTXT_PHASE EQU 8
BTEXT_BASE EQU (12*13)
BTEXT_PHASE EQU 8
TEXTLINE_PHASE EQU 6
PAD MID_PHASE EQU 1
PAD END_PHASE EQU 7
WINT Ext_BASE EQU (12*48)
WINT Ext_PHASE EQU 7
INIT TEXT1_BASE EQU (12*75)
INIT TEXT1_PHASE EQU 11
INIT TEXT2_BASE EQU (12*40)
INIT TEXT2_PHASE EQU 11
INIT TEXT3_BASE EQU (12*40)
INIT TEXT3_PHASE EQU 5
INIT TEXT4_BASE EQU (12*40)
INIT TEXT4_PHASE EQU 5
INIT TEXT5_BASE EQU (12*13)
INIT TEXT5_PHASE EQU 8

ENDIF

IF (SYSTEM = SYSTEM_PAL_N)

FREQ 42984672
TIME_2US4 EQU 103
TIME_4US5 EQU 193
TIME_7US65 EQU 1181
TIME_9US62 EQU 1271
TIME_6US EQU 2749
TIME_TOTALE EQU TIME_6US
TIME_PRESYC EQU 47
TIME_SYNC EQU 202
TIME_PREBURST EQU 39
TIME_BURST EQU 144
TIME_POSTBURST EQU 5
TOT_LINES EQU 304
PRE_LINES EQU 35
POST_LINES EQU 19
LEFT_SPACE EQU (12*12)
RIGHT_SPACE EQU (12*7)
LEFT_PAD_PHASE EQU 1
RIGHT_PAD_PHASE EQU 10
BALL_PHASE EQU 4
LEFTSCORE_BASE EQU (12*20)
LEFTSCORE_PHASE EQU 9
RIGHTSCORE_BASE EQU (12*15)
RIGHTSCORE_PHASE EQU 4
TTXT_BASE EQU (12*65)
TTXT_PHASE EQU 8
BTEXT_BASE EQU (12*13)
BTEXT_PHASE EQU 8
TEXTLINE_PHASE EQU 6
PAD MID_PHASE EQU 1
PAD END_PHASE EQU 7
WINT Ext_BASE EQU (12*48)
WINT Ext_PHASE EQU 7
INIT TEXT1_BASE EQU (12*75)
INIT TEXT1_PHASE EQU 11
INIT TEXT2_BASE EQU (12*40)
INIT TEXT2_PHASE EQU 11
INIT TEXT3_BASE EQU (12*40)
INIT TEXT3_PHASE EQU 5
INIT TEXT4_BASE EQU (12*40)
INIT TEXT4_PHASE EQU 5
INIT TEXT5_BASE EQU (12*13)
INIT TEXT5_PHASE EQU 8

ENDIF

IF (SYSTEM = SYSTEM_NTSC)

FREQ 42954540
TIME_2US4 EQU 103
TIME_4US5 EQU 193
TIME_7US65 EQU 1181
TIME_9US62 EQU 1271
TIME_6US EQU 2748

Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.guinee.com/projects
; add16 macro
; * This is a macro to add two 16 bit numbers, inputs two arguments, each pointing at the
; * the nms register at position arg+1. Results are stored in registers referred to by first arg.
; * clocks: 6
; *
; add16 MACRO 2
; add \(1\), \(2\) ;2
; nrc \(1\) \(2\)
; inc \(1\) + 1 ;1
; add \(1\) + 1, \(2\) + 1 ;2
; ENDM

; sub16 macro
; * This is a macro to subtract two 16 bit numbers, inputs two arguments, each pointing at the
; * the nms register at position arg+1. Results are stored in registers referred to by first arg.
; * clocks: 6
; *
; sub16 MACRO 2
; sub \(1\), \(2\) ;2
; sc \(1\)
; dec \(1\) + 1 ;1
; sub \(1\) + 1, \(2\) + 1 ;2
; ENDM

; neg16 macro
; * This is a macro to negate one 16 bit number, inputs one argument, pointing at the lab register
; * followed by the nms register at position arg+1. Results are stored in registers referred to by first arg.
; * clocks: 5
; *
; neg16 MACRO 1
; not \(1\)
; not \(1\)+1
; inc \(1\)
; nrc \(1\)
; inc \(1\)+1
; ENDM

; add168 macro
; * This is a macro to add one 16 bit register with one 8 bit constant, inputs two arguments, register, constant.
; * register argument pointing at the lab register followed by the nms register at position arg+1. Results are stored in registers referred to by first arg.
; * clocks: 4
; *
; add168 MACRO 2
; add \(1\), #\(2\) ;2
; nrc \(1\)
; inc \(1\)
; ENDM

; sub168 macro
; * This is a macro to subtract one 16 bit constant from one 16 bit register, inputs two arguments, register, constant.
; * register argument pointing at the lab register followed by the nms register at position arg+1. Results are stored in registers referred to by first arg.
; * clocks: 4
; *
; sub168 MACRO 2
; sub \(1\), #\(2\) ;2
; sc \(1\)
; dec \(1\) + 1 ;1
; ENDM

; mov16 macro
; * This is a macro to set one 16 bit register to one 16 bit constant, inputs two parameters, register, constant where.
; * register argument pointing at the lab register followed by the nms register at position arg+1. Results are stored in registers referred to by first arg.
; * clocks: 4
; *
; mov16 MACRO 2
; mov \(1\), #\(2\)<<16
; mov \(1\), #\(2\)>>8
; ENDM

; dosound1 macro
; * This is a macro initials a sound in sound channel one.
; * taking three constant parameters: speed, speeddifference

; More info available at: http://www.rickard.gunee.com/projects
dosound1 MACRO 3
mov16 \wavespeed,\(\|\) \r4 start frequency
mov16 \wavespeeddiff,\(\|\) \r4 frequency change speed
mov \waveltimmer,#\(\|\) \w sound length
ENDM

* This is a macro that initializes a sound in sound channel two
* and length
* clocks: 10

dosound2 MACRO 3
mov16 \wavespeed,\(\|\) \r4 start frequency
mov16 \wavespeeddiff,\(\|\) \r4 frequency change speed
mov \waveltimmer,#\(\|\) \w sound length
ENDM

* This macro does the same as \lcall but in 2 words
* clocks: 4

pjmp MACRO 1
paw \(\|\)
jmp \(\|\)
ENDM

* This macro does the same as \lcall but in 2 words
* clocks: 4

pcall MACRO 1
paw \(\|\)
call \(\|\)
ENDM

* This macro outputs a constant to the video DA
* clocks: 2

vout MACRO 1
mov w,#\(\|\)
mov video,w
ENDM

* This macro outputs data from a register to the video DA
* clocks: 2

voutr MACRO 1
mov w,\(\|\)
mov video,w
ENDM

* This macro creates a delay of 3 clock cycles only using
* one word of program memory.
* clocks: 3

tnop MACRO
jmp :tnopj
:tnopj
ENDM

* This is a macro for creating delay that depends of the
* contents of w, it adds w to the low part of pc, and adds
* nops after the jmp instruction, the number of nops is
* specified as a parameter to the function
* clocks: w+3

setphase MACRO 1
jmp pc+w
REST \1
nop
ENDM
ENDM

* This is a macro for creating delays by calling the delay
* functions, it minimizes the number of program words to max
* 6 words. For delays less than 1017 and longer than 9
* it uses the short delay functions at the cost of 2-3 words
* for shorter delays it uses the fixed delays at a cost of 1
* to 3 words, longer delays are done by a call to the short
* delay functions followed by a long delay call with a total
* cost of 4-6 words of program memory. The macro can handle
* delays from 0 to 260k cycles.
* WARNING, no guarantee that this really works correctly for
* all delays as it quite complex and I'm too lazy to test it

Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
delay MACRO
  IF (\1) < 1
    ERROR 'Negative delay'
ENDIF
:delbase
  IF (delbase & $E00) = (delay9 & $E00)
    IF (\1)=6
      IF (\1)/3=1
        nop
      ENDIF
      IF (\1)/3=2
        nop
      ENDIF
      IF (\1)/3 > 0
        REPT (\1)/3
        tnop
      END
      ENDIF
      ENDIF
      IF (\1)>5 AND (\1)<10
        call delay6 - (\1)-6
      ENDIF
      IF (\1)>9 AND (\1)<1027
        mov w,((\1)-6)>>2
        call delay_short_0 - (((\1)-6)>>3)
      ENDIF
      IF (\1)>1026
        IF (((\1)-12)//1017)<10
          mov w,(((\1)-12)//1017)+1017)>>2
          call delay_short_0 - (((((\1)-12)//1017)+1017)>>3)
          mov w,(((\1)-12)//1017)-1
        ELSE
          mov w,(((\1)-12)//1017)>>2
          call delay_short_0 - (((((\1)-12)//1017)>>3)
          mov w,(((\1)-12)//1017)
        ENDIF
        call delay_long
      ENDIF
      ELSE
        IF (\1)<7
          IF (\1)/3=1
            nop
          ENDIF
          IF (\1)/3=2
            nop
          ENDIF
          IF (\1)/3 > 0
            REPT (\1)/3
            tnop
          END
          ENDIF
          ENDIF
          IF (\1)>6 AND (\1)<11
            page delay6
            call delay6 - (\1)-7
          ENDIF
          IF (\1)>10 AND (\1)<1028
            mov w,((\1)-7)>>2
            page delay_short_0
            call delay_short_0 - (((\1)-7)>>3)
          ENDIF
          IF (\1)>1027
            IF (((\1)-14)//1017)<10
              mov w,(((\1)-14)//1017)+1017)>>2
              page delay_short_0
              call delay_short_0 - (((((\1)-14)//1017)+1017)>>3)
              mov w,(((\1)-14)//1017)-1
            ELSE
              mov w,(((\1)-14)//1017)>>2
              page delay_short_0
              call delay_short_0 - (((((\1)-14)//1017)>>3)
              mov w,(((\1)-14)//1017)
            ENDIF
            page delay_long
            call delay_long
          ENDIF
          ENDIF
          ENDIF
ENDM

;****************************** delay functions ******************************
/* Different delay functions to be able to create long delays */
/* using as few bytes of program memory as possible */
/* These functions are required by the delay macro */
/* delays with exact clock count uses no registers */
/* short delays use temp0 */
/* long delays use temp0 and temp1 */
;****************************** delay functions ******************************

; delay9   nop   ;1 entrypoint of delay9 that delays 9 clocks
; delay8   nop   ;1 entrypoint of delay8 that delays 8 clocks
; delay7   retp   ;1 entrypoint of delay7 that delays 7 clocks
; delay6   retp   ;3 entrypoint of delay6 that delays 6 clocks
; delay_short_3   nop   ;1 entrypoint of delay_short_3 that delays 4w + 8
; delay_short_2   nop   ;1 entrypoint of delay_short_2 that delays 4w + 7
; delay_short_1   nop   ;1 entrypoint of delay_short_1 that delays 4w + 6
; delay_short_0   mov   temp0,w   ;1 entrypoint of delay_short_0 that delays 4w + 5
delay_short_m    
  dec sz temp0  ;(2) decrease counter, mainloop of delay short  
  jmp delay_short_m  ;3 keep looping until counter is zero  
  retp  ;3 return back to caller  

delay_longmov 
  mov w, #251  ;1 set long time counter from w  
  mov w, delay_short.m  ;3 move time to delay in short delay  
  call delay_short.m  ;1012 time to delay is 251*4+8=1012  
  dec sz temp  ;(2) decrease long time counter  
  jmp delay_longmov  ;3 keep looping until counter is zero  
  retp  ;3 return back to caller  

STRUCTOR  jumpstar  
  jum p start  

STRUCTOR  memtovide o  
  mov temp0, w  ;1 set pixel counter  
  mov mov, temp0, w  ;1 set pixel counter  
  mov mov, video, w  ;1 send to video output  
  mov mov, w, ind  ;1 get upper level byte from mem  
  mov mov, w, ind  ;1 get upper level byte from mem  
  inc mov, ind  ;1 point at next pixel  
  cl rb mov, frr, 5  ;1 set lower bank  
  nop mov, w, ind  ;1  
  mov mov, video, w  ;1 send to video output  
  dec sz temp  ;(2) decrease pixel counter  
  jmp memtovide o  ;3 keep looping until all pixels are done  
  retp  ;3 get outta here  

STRUCTOR  setgraphics  
  mov mov, temp3, w  ;1 set pixel counter  
  mov temp3, #PALETTE_PAGE  ;1 set page  
  mov v, temp0  ;1 get image pointer  
  iread mov, #0, #10, rom  ;4 read pixel data from rom  
  mov mov, w, m  ;1 get slowmem nibble  
  mov mov, w, m  ;1 set pixel counter, assuming all palettes within the same page  
  mov mov, #PALETTE_PAGE  ;1 select page  
  iread mov, #0, #10, rom  ;4 read palette  
  mov mov, ind, w  ;1 select second level bank  
  mov mov, #F95C  ;1 mask out two upper bits  
  mov mov, ind, w  ;1 store second level two upper bits  
  mov mov, ind, w  ;1 move upper bits into correct position (1/2)  
  mov mov, ind, w  ;1 move upper bits into correct position (2/2)  
  mov mov, w, m  ;1 set second level lower nibble  
  mov mov, w, m  ;1 or third level lower nibble  
  cl rb mov, frr, 5  ;1 get back to first level bank  
  cl rb mov, ind, w  ;1 point at next pixel memory position  
  inc mov, temp0  ;1 point at next nibble  
  mov mov, #1, #0  ;(2) decrease pixel counter  
  inc mov, temp1  ;1 if page overflow, go to next page  
  dec sz temp  ;(2) decrease pixel counter  
  jmp memtovide o  ;3 keep looping until all pixels are done  
  retp  ;3 get outta here  

IMITER  simplescolorfa  
  mov temp2, w  ;1  
  mov temp0, #63  ;2  
  mov temp1, #black  ;2  
  skip mov  

IMITER  simplescolor  
  mov temp2, w  ;1 set color cycle counter  
  mov mov, temp0  ;2 set first level  
  delay 4  ;4 delay to get 12 cycle loop  
  mov mov, temp1  ;2 set second level  
  dec sz temp  ;(2) decrease color cycle counter  
  jmp simplescolor  ;3 do all cycles  
  retp  ;3 get outta here  

IMITER  strout  
  mov mov, temp0  ;1 used as character temp storage  

More info available at: http://www.rickard.gunee.com/projects
; * temp2 used as character counter
; * temp1|temp3 = pointer to string
; * temp4 = line (0..7) + FONT_BASE
; * temp5 = length
; * clocks: 8 * 12 * w = 44 + 1
;******************************************************************************

strout_cl vout black ;2 pixel three to seven
delay 2 ;2
rr temp0 ;1
snc ;1(2)
mov w,#53 ;1
mov video,w ;1
decsz temp2 ;1(2)
jmp strout_l ;3

strout mov m,temp1 ;2 set character page
vout black ;2 pixel one starts here
mov w,temp3 ;1 get pointer to characters
lread ;4 read one character
add w,temp4 ;1 update according to line and fontbase
mov m,##(gamdata + FONT) >> 8) ;1 set font-page
lread ;4 read character pixels from font
mov temp0,w ;1 store character pixels in temp0
mov w,#black ;1
rr temp0 ;1
snc ;1(2)
mov w,#53 ;1
inc temp3 ;1 point at next character
mov video,w ;1
snz ;1(2)
inc temp1 ;1
delay 2
vout black ;2 pixel three starts here
mov temp2,#5 ;2
rr temp0 ;1
snc ;1(2)
mov w,#53 ;1
mov video,w ;1
decsz temp5 ;1(2)
jmp strout_cl ;3
vout black
ret ;3

;********************************************************************************
;                 charout  ❏                 
; * output character from font in rom
; * temp0 used as character temp storage
; * temp1 = intensity
; * temp3|temp2 = pointer to char
; * clocks: 112 + 1
;********************************************************************************

charout mov m,temp3 ;2 set high part of pointer (i.e. page)
mov w,temp2 ;1 set low part of pointer
lread ;4 read character data from rom
mov temp0,w ;1 store character data in temp0
mov temp2,#8 ;2 character is 8 pixels wide
charout_l vout black ;2 start with black level
delay 2 ;2 delay to keep phase
rr temp0 ;1 rotate character data
snc ;1(2) if last was zero, keep black
mov w,temp1 ;2 get amplitude of character
mov video,w ;1 output to video DA
decsz temp2 ;1(2) decrease pixel counter
jmp charout_l ;3 go all pixels
vout black ;2 set black level
retp ;3 get out a here

;********************************************************************************
;               mul_12  ❏               
; * multiply contents of w with constant 12, result in w
; * clocks: 13
;********************************************************************************
mul_12 mov temp0,w ;1 temp2 = w
add temp2,w ;1 temp2 = w+2
add temp2,w ;1 temp2 = w+3
clc
r1 temp2 ;1 temp2 = w+3+4 - w+12
mov w,temp2 ;1 result in w (one could optimize one word and clock here, but
I'm too lazy to recalculate the timing of the main program)
ret ;3 return back

;********************************************************************************
;               makepaddle  ❏               
; * Creates paddle graphics one a line, based on y position. 
; * This is used for both left and right paddle. 
; * fsp = pointer to graphics buffer.
; * temp0 = temporary storage, contents destroyed
; * temp1 = temporary storage, contents destroyed
; * temp2 = y-pos of paddle
; * temp7 = line number
; * clocks: 514 + 1 clocks
;********************************************************************************

makepaddle clr temp1 ;1 clear output register
mov temp0,temp0 ;2 mov temp0 = y
sub temp0,temp7 ;2 temp0 = y - checkline
sc ;1(2) check if result is negative, y < checkline
setb temp1.0 ;1 no, y > checkline, set first bit
mov temp0,temp2 ;2 temp0 = y
add temp0,#PAD_ENDSIZE ;2 temp0 = y + endsize
sub temp0,temp7 ;2 temp0 = y + endsize - checkline
c ;1(2) check if result is negative, yendsize < checkline
setb temp1.1 ;1 no, yendsize > checkline, set second bit
mov temp0,temp2 ;2
add temp0,#(PAD_MIDSIZE + PAD_ENDSIZE) ;2
sub temp0,temp7 ;2
c ;1(2)
cleb temp1.0 ;1 mov temp0,temp2 ;2
add temp0,#(PAD_ENDSIZE + PAD_MIDSIZE) ;2
sub temp0,temp7 ;2
c ;1(2)
cleb temp1.1 ;1 mov w,temp1 ;1
add pc,w ;3 select what to do according to result
jmp paddle_black ;3 0 - no paddle
jmp paddle_bottom ;3 3 - top part of paddle
jmp paddle_top ;3 1 - bottom part of paddle

paddle_middle
delay 19
mov temp0,##(PAD_ENDSIZE + BALL + 40) ;2 set low rom pointer to paddle graphics
mov temp1,##(PAD_ENDSIZE + BALL + 40) ;2 set high rom pointer to paddle graphics
mov temp2,##(PALETTE_BOW + gamedata2) ;2 set palette
mov w,##;1 padd = 8 pixels wide
jmp setgraphics ;253

paddle_bottom
mov temp3,#PAD_ENDSIZE ;2 set bottom reference
mov temp0,##(PAD_ENDSIZE + BALL + 48) ;2 set low rom pointer to paddle graphics
mov temp1,##(PAD_ENDSIZE + BALL + 48) ;2 set high rom pointer to paddle graphics
mov fsr,#BALL_BUFFER ;2 points to ball buffer
mov w,##;1 padd = 8 pixels wide
jmp setgraphics ;253

paddle_top
mov temp3,#PAD_MIDSIZE ;2 set top reference
mov temp0,##(PAD_MIDSIZE + BALL) ;2 set low rom pointer to paddle graphics
mov temp1,##(PAD_MIDSIZE + BALL) ;2 set high rom pointer to paddle graphics
mov clocks,##;2 calls
jmp paddle_top ;3

paddle_bottom_
add temp2,temp3 ;2 temp2 = y + size
sub temp2,temp7 ;2 temp2 = y + size - linenumber
cleb r1 temp2 ;1
r1 temp2 ;1
mov w,##;1 w = (y + size - linenumber) * 8
add temp0,w ;1 select line in graphics
snc ;1(2) check for overflow
inc temp1 ;1 if overflow, change to next page
mov temp2,##(PALETTE_BOW + gamedata2) ;2 set palette
mov w,##;1 padd = 8 pixels wide
jmp setgraphics ;253

paddle_black
delay 16
mov temp0,##(PAD_ENDSIZE + EMPTYFX) ;2 set low rom pointer to paddle graphics
mov temp1,##(PAD_ENDSIZE + EMPTYFX) ;2 set high rom pointer to paddle graphics
mov temp2,##(PALETTE_BOW + gamedata2) ;2 set palette
mov w,##;1 padd = 8 pixels wide
jmp setgraphics ;253

;****************************** makeball ******************************
; * Creates ball graphics on a line
; * Requires maped as it reuses routines for empty line to
; * save rom memory (as both paddle and ball are 12 pixels)
; * ftemp = pointer to graphics buffer
; * temp0 = temporary storage, contents destroyed
; * temp = temporary storage, contents destroyed
; * temp = temporary storage, contents destroyed
; * temp7 = line number
; * padd, clocks
;****************************** makeball ******************************

makeball bank #00
mov temp2,##(PAD_ENDSIZE + BALL) ;2 set low rom pointer to ball graphics
mov temp1,##(PAD_ENDSIZE + BALL) ;2 set high rom pointer to ball graphics
mov temp2,bally_h ;2
sub temp2,temp6 ;2
snc ;1(2)
jmp noball_j1 ;3
mov temp2,bally_h ;2
sub temp2,#BALL_LINES ;2
sc ;1(2)
jmp noball_j2 ;3
mov w,temp2 ;1
call mul12 ;13
add temp0,w ;1 select line in graphics
snc ;1(2) check for overflow
inc temp1 ;1 if overflow, change to next page
mov temp2,##(PALETTE_BOW + gamedata2) ;2 set palette
mov fsr,#BALL_BUFFER ;2 points to ball buffer
mov w,##;1 padd is 12 pixels wide
jmp setgraphics ;377

noball_j1
delay 8

noball_j2
delay 11
mov fsr,#BALL_BUFFER ;2 points to ball buffer
mov temp0,##(PAD_ENDSIZE + EMPTYFX) ;2 set low rom pointer to paddle graphics
mov temp1,##(PAD_ENDSIZE + EMPTYFX) ;2 set high rom pointer to paddle graphics
mov temp2,##(PALETTE_BOW + gamedata2) ;2 set palette
mov w,##;1 padd = 8 pixels wide
jmp setgraphics ;377

Generating color composite video signals in software, written by Rickard Guneé
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.guenee.com/projects

```
call strout  ; STR_LEN*8*12+44
inc temp4  ;
delay TIME_IMAGE - ((STR_LEN-1)*8*12+44) - INITTEXT4_BASE + INITTEXT4_PHASE - (2*2+2+1+4+(12*4)-7*67*2+1) - ((STR_LEN-1)*8*12+44)
decsz temp7  ;(2)
jmp iscr_l  ;
delay 2
pcall hsync
delay TIME_IMAGE - 17 - 2
mov w,#49
pcall emptylines
delay 12-5
itert STR2,STR2_LEN,INITTEXT5_BASE,INITTEXT5_PHASE
delay TIME_IMAGE - 17 - 2
mov w,#POST_LINES + ((MID_LINES - INITLINES)/2) + (MID_LINES - INITLINES)//2 - 1 ;
pcall emptylines
delay 12-1
pcall hsync

bank 000  ;
ab joylup,old  ;(2)
jmp initnoupwarpr  ;
ab joylup  ;
dec gamekind  ;
mov w,gamekind  ;
xor w,#FF  ;
mov w,#3  ;
snz ;(2)
mov gamekind,w  ;

initnoupwarpr
ab joyldown,old  ;(2)
jmp initn downwar pr  ;
ab joyldown  ;(2)
in gamekind  ;
mov w,gamekind  ;
xor w,#4  ;
snz ;(2)
clr gamekind  ;

initn dow nwar pr
delay TIME_IMAGE - 10 - 8 - 6 - 9 - 4  
ab joylbutton,old  ;(2)
skip  ;
snb joylbutton ;(2)
jmp initi screen ;(3)
clr joylbutton,old  ;
pjmp premain

initnoupwarpr
jmp initnoupwarpr

initn dow nwar pr
jmp initn dow nwar pr

org $200

;******************************************************************************
;* nose sync
;******************************************************************************

vsync
mov w,#4  ;1  odd, make 5 pulses instead
mov temp2,w  ;5  clocks until sync, make those pulses,
call short_sync  ;
clr video  ;1  set video level to sync
delay (TIME_27US5 - 1)  ;
vout black  ;2  set video level to black
call vsound  ;65
delay (TIME_4US5 - 6 - 65)  ;
short_sync ;
jmp long_sync_l  ;(3)
mov w,#5  ;1  odd, make 4 pulses instead of 5
short_sync
mov temp2,w  ;1
clr video  ;1  set video level to sync
call vsound  ;65
delay (TIME_20US4 - 65 - 1)  ;20us long sync pulse
vout black  ;2  set video level to black
delay (TIME_29US6 - 6)  ;
short_sync ;
jmp short_sync_l  ;(3)
reset 5  ;3
pjmp vrealsound  ;62

;******************************************************************************
;* nose sync
;******************************************************************************

hsync
delay TIME_PRESYNC-3-1  ;85
clr video  ;1
call sound  ;61
delay TIME_SYNC-2-61  ;248
vout neutral  ;2
delay TIME_PREBURST-2  ;44
mov temp0,#12  ;2
vout 6  ;2
delay 4  ;2
vout 21  ;2
decsz temp0  ;(2)
jmp hsync1  ;3
delay 2  ;2
vout neutral  ;2
delay time_postburst - 2-3;114
reset  ;3
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunee.com/projects
doserve2 clr bally_speed_h
mov bally_speed_l,#80
mov bally_speed_l,#2
mov bally_speed_h,#FF
bank $20

.; WAVEFORM_FREQ, WAVEFORM_DIP, WAVEFORM_LEN ;
bank $00

.; state.1
j mp

server

.; delay 3-3+12+14+68+68+23+23-3-37
j mp

; main loop

 MAIN

premain test gamekind ;premain sets the game kind when arriving from init
set mixedbits.7
set mixedbits.6

main
call vsync ; vertical sync, frame starts here
call haync ;line 1
bank $00
;j
jb joyup,noup1warpr ;j2(4) joy1 up pressed ?
csa y1,#PLAYFIELD_LINES-PAD_SIZE-1 ;j3(4) yes, can paddle 1 move up ?
icc y1 ;j1 yes, increase y-pos of paddle 1
nop ;j1 delay to keep phase
cab y1,#1
dec y1 ;j1 yes, decrease y-pos of paddle 1
nop ;j1 delay to keep phase
cab y1,#1
dec y1 ;j2 yes, can paddle 2 move down ?
nop ;j1 delay to keep phase

noup1warpr

;jb joydown,nodown1warpr ;2(4) joy2 down pressed ?
cab y1,#1
dec y1 ;j1 yes, decrease y-pos of paddle 2
nop ;j1 delay to keep phase

nodown2warpr

; mover w, state
and w,#FE
jmp warpmove
;jb state.2,warpgover
nop
;jb state.2,warpgover
add bally,bally_speed
;jb bally_speed.7,negxyxwarpr
mov w,bally_h
and w,#00
xor w,#00
snz ;j1(2) if upper 1/4, then clear
clr bally_h ;j1

negxyxwarpr

add bally,bally_speed
;jb bally_speed.7,negxyxwarpr
mov w,bally_h
and w,#00
xor w,#00
snz ;j2(2) if upper 1/8, then clear
cjne bally_h,#1,leftwarpr
mov w,bally_h
and w,#00
xor w,#00
snz
clr bally_h

leftwarpr

;jb joyup,nospush1warpr
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 yes, change x-direction of ball

nospush1warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush2warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush2warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush3warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush3warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush4warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush4warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush5warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush5warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush6warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush6warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush7warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush7warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush8warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush8warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush9warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush9warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush10warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush10warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush11warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush11warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush12warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush12warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush13warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush13warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush14warpr

;jb joyup,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction

nospush14warpr

;jb joydown,nowarp
mov bally_speed,80
add bally_speed,80
add bally_speed,80
bally_speed.1 ;2 add speed in x-direction

nop ;j4 subtract more speed in y-direction
mov   balh_h,#BALLAREA_WIDTH-1 ; stop ball from being out of bounds
bank   520 ;
dosound1 WAVEPAD_FREQ,WAVEPAD_DIF,WAVEPAD_LEN ;
bank   510 ;
jb  joybutton, nosmash2warp ; joy right pressed?
sbl 18 bally_speed, #80 ; add more speed in y-direction
jb  joyup, noscrewwp2warp ; joy up pressed?
addl 18 bally_speed, #560 ; add more speed in y-direction
nop ;
delay
noscrewwp2warp
jb  joydown, noscrewdown2warp ; joy down pressed?
sbl 18 bally_speed, #560 ; subtract more speed in y-direction
nop ;
delay
noscrewdown2warp
jmp  nomiss1 ;
miss
sb  state.0 ; was it the right player that served
inc  pl ; yes, increase player two's points
cne  pl,#49 ; player 2 is going to serve
screen
setb  state.2
bank  520 ;
dosound2 WAVEMISS_FREQ,WAVEMISS_DIF,WAVEMISS_LEN ;
bank  510 ;
delay 44-24
nomiss1
rightwarpprcaw
bally_h,#1,bottomwarpp  ; check if y less than one
negl  bally_speed ; yes, change y-direction of ball
mov   bally_h,#1 ; stop ball from being out of bounds
bank  520 ;
dosound2 WAVENULL_FREQ,WAVENULL_DIF,WAVENULL_LEN ;
bank  510 ;
bottomwarpp
clw  bally_h,PLAYFIELD_LINES-1-BALL_LINES,topwarpp  ; check if y larger than
playfield height
negl  bally_speed ; yes, change y-direction of ball
mov   bally_h,PLAYFIELD_LINES-1-BALL_LINES
bank  520 ;
dosound2 WAVENULL_FREQ,WAVENULL_DIF,WAVENULL_LEN ;
bank  510 ;
topwarpp
warppower
warppower
bank  500 ;
mov  w,rc ;
and  w,#811100000 ;
mov  oldj1,w ;
mov  w,ra ;
and  w,#00001111 ;
or  oldj1,w ;
mov  oldj2,rb ;
delay TIME_IMAGE - 17 - 1 - 28 - 6 - 12 - 68 - 68 - 28 - 23 - 23 - 9
mov  w,#PRE_LINES-1 ;
call emptylines
delay 11
pcall haync
delay 12 - TEXTLINE_PHASE
mov  w,((TIME_IMAGE-14) / 12)-1
pcall simplescolorfb
delay TEXTLINE_PHASE + ((TIME_IMAGE-14) // 12)
pcall haync
delay TIME_IMAGE-5
mov  temp7,#7
mov  temp4,((gamedata + FONT) & $ff)
pcall haync
delay LEFTSCORE_BASE - LEFTSCORE_PHASE
mov  temp3,((gamedata + NUMBERS) >> 8) ; let temp3 point at correct page
mov  w,#63 ; high intensity
sb  state.0 ; left players serve?
mov  w,#40 ; no, lower intensity
cic  temp1.w ;
mov  w,cp1 ;
mov  temp3.w ;
temp2 = points2
rl  temp2 ;
temp2 = points4
add  temp2,((gamedata + NUMBERS) & $ff) + 7 ; add low part of point to numbers
sub  temp2,temp7 ; select line in font
pcall charout

delay TEXTBASE-TTEXT_PHASE-(LEFTSCORE_BASE-LEFTSCORE_PHASE+128)
mov  temp1,((gamedata + STRG) >> 8) ;
mov  temp3,((STRG + gamedata) & $ff) ;
mov  temp5,STRG_LEN
pcall strout
inc  temp4

delay RIGHTSCORE_PHASE+128
mov  temp3,((gamedata + NUMBERS) >> 8) ; let temp3 point at correct page
mov  w,#63 ; high intensity
sb  state.0 ; right players serve?
mov  w,#40 ; no, lower intensity
cic  temp1.w ;
mov  w,cp2 ;
mov  temp3.w ;
temp2 = points2
rl  temp2 ;
temp2 = points4
rl  temp2 ;
temp2 = points8

Generating color composite video signals in software, written by Rickard Guné
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunéé
More info available at: http://www.rickard.gunee.com/projects
Generating color composite video signals in software, written by Rickard Gunée
More info available at: http://www.rickard.gunée.com/projects
org $600

gamedata

dw $000,$000,$000,$000,$010,$030,$030,$010,$000,$000,$000,$000,$000,$000,$07w,$000,$000,$300 ; $000..300f

dw $500,$600,$700,$700,$500,$518,$500,$500,$518,$500,$500,$518,$500,$500,$518,$500 ; $10..30f

dw $83c,$446,$203,$003,$303,$166,$350,$500,$40f,$500,$306,$284,$386,$257,$100 ; $20..30f

dw $063,$267,$036,$57b,$376,$376,$365,$093,$563,$767,$263,$326,$326,$33$w,$050 ; $030..303f

dw $635,$766,$665,$766,$766,$406,$242,$303,$606,$336,$636,$353,$636,$646,$920 ; $040..404f

dw $d3c,$a46,$99c,$518,$235,$366,$330,$440,$663,$763,$803,$500,$500,$518,$670 ; $050..505f

dw $d3c,$336,$235,$41c,$518,$663,$763,$763,$406,$500,$518,$670,$326,$326,$326,$326 ; $060..606f

dw $550,$600,$57w,$763,$763,$58w,$560,$528,$230,$33w,$333,$333,$76w,$57w ; $070..707f

dw $700,$800,$79w,$556,$713,$203,$353,$500,$342c,$556,$556,$971,$706,$570,$470 ; $080..809f

dw $631,$350,$341c,$518,$518,$530,$500,$500,$507,$506,$516,$236,$333,$536,$637 ; $090..909f

dw $645,$250,$005,$346,$203,$518,$528,$526,$516,$256,$006,$006,$006,$016,$267 ; $0a0..a09f

dw $700,$250,$03w,$333,$333,$34w,$550,$518,$707,$506,$536,$516,$666,$516,$267 ; $0b0..b09f

dw $318,$350,$341c,$518,$518,$530,$500,$500,$507,$506,$516,$236,$333,$536,$637 ; $0c0..c0af

dw $440,$250,$005,$346,$203,$518,$528,$526,$516,$256,$006,$006,$006,$016,$267 ; $0d0..d0af

dw $550,$600,$57w,$763,$763,$58w,$560,$528,$230,$33w,$333,$333,$76w,$57w ; $0e0..e07f

dw $700,$800,$79w,$556,$713,$203,$353,$500,$342c,$556,$556,$971,$706,$570,$470 ; $0f0..f07f

dw $550,$600,$57w,$763,$763,$58w,$560,$528,$230,$33w,$333,$333,$76w,$57w ; $100..107f

; * second vSync cycle as vSync is called at twice the speed *
; * as sync, so vrealSound is dependent of realtime *
; * clocks: 58 *

; **********************
; vrealSound **********
; **********************

vrealSounds

temp2.0  : $1(2)

imp  \quad realSound + 54

delay $56

ret  $3

; **********************
; vrealSound **********
; **********************

realSound bank $020

mov  n, #($SINTABLE + gamedata) >> 8

add  $16, wavelpos, wavelSpeed

and  wavelpos_h, #31

mov  soundTemp0, wavelpos_h

add  soundTemp0, #($SINTABLE + gamedata) & $FF

position

mov  w, soundTemp0

iread $4

mov  w, m

mov  soundTemp0, w

mov  c, #10

sbb  c, #10

or  soundTemp0

mov  soundTemp0, c

add  c, soundTemp0

mov  m, #($SINTABLE + gamedata) >> 8

and  $16, wavelpos, wavelSpeed

and  wavelpos_h, #31

mov  soundTemp1, wavelpos_h

add  soundTemp1, #($SINTABLE + gamedata) & $FF

position

mov  w, soundTemp1

iread $4

mov  w, m

mov  soundTemp1, w

sbb  soundTemp1, #3

mov  soundTemp1, w

add  w, soundTemp0

mov  audio, w

bank $00

ret  $3

Generating color composite video signals in software, written by Rickard Gunée

More info available at: http://www.rickard.gunée.com/projects
Appendix E: Game System PCB layout

The mirrored PCB-layout in scale 1:1, also including the component layout: