The Uzebox Project

Alec Bourque
March 2010

1 Introduction

A couple years ago, I found on the internet a Pong game made out of a single PIC microcontroller and a few resistors! The microcontroller was generating a monochrome video signal, reading the joysticks and outputting basic sound all at once with cycle-counting precision. I found this pretty awesome and wanted to build my own video project using only AVR microcontrollers. I initially settled for some sort of retro computer that would use multiple microcontrollers for the various functions like the main CPU, sound synthesis and the keyboard matrix decoder. However, after adding some external memory (ROM & RAM), the video synchronization generator and all support chips (multiplexers, shift registers, etc), I ended up filling a 2x1 foot proto board...not quite what we could call hobbyist friendly! The approach had another flaw: I had to continually switch the ISP programming port between the 3 AVRs. No, there had to be a better solution.

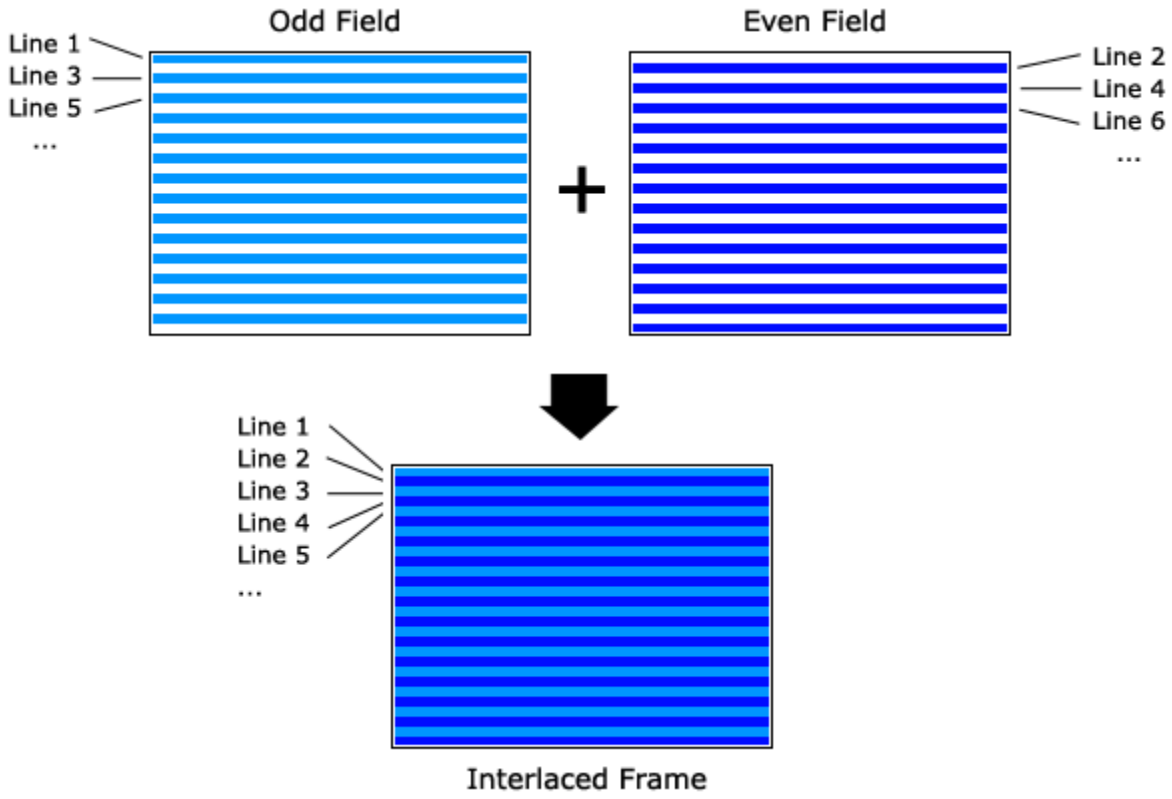
I went back to the drawing board just about the time when the ATMega644 came out. This chip could run at 20Mhz, had 64K of flash and 4K of RAM. Seemed perfect for a simple game console. I roughly calculated it would have enough power for the video generation, 4 voices sound mixing and running the main program. However, since I now wanted it have at least 256 simultaneous colors, color generation became an issue. As you'll see in the following video primer, monochrome video is pretty easy to generate. Color on the other hand is much more complex and not practical on low end MCU. Of all the color generation methods I investigated, I opted for a simpler one: an AD725 color generation chip. Commonly called a RGB-to-NTSC converter, this chip accepts 3 voltage inputs, one for each red, greed and blue component and handles the magic required to produce that complex NTSC color output.

Driving this new design was the uttermost simplicity, something any hobbyist could build up in no time, ideally made out only of DIP chips. So, apart from the AD725, the console is solely based on an Atmega644 and discrete components like resistors and capacitors. I also decided to invest time developing some sort of interrupt driven game “kernel” that would mix music, read joypads status, generate video synchronization and render frames independently of the main game program. Although I knew I couldn't get around insane cycle-counting for the kernel, once done, it would makes the actual game development a piece of cake.

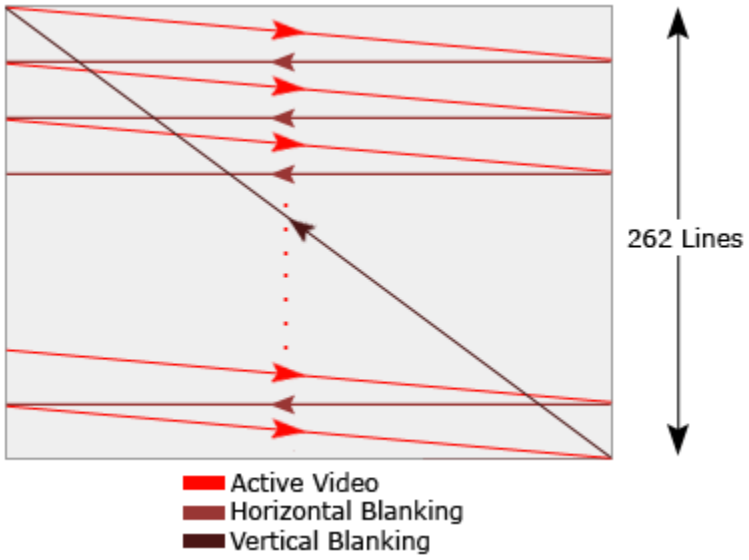
2 Video Signal Primer

When I started the project, I didn't spent a lot of time thinking about the possible video output. A good old classic composite TV output seemed most retro and appropriate than, say, VGA. After all, everybody has a TV. And it's almost always placed in the living room, a much better place (I thought) to show off your games to family and friends! :) The term composite video refers to the fact that both intensity, color and synchronization information are mixed together into a single signal. And one thing I've noticed about it: there's many standards out there. Seemed like each part of the world decided to make their own format, mostly incompatible with each other. Since I live in Canada, the Uzebox produces NTSC composite video, the analog standard mainly used in North America. It worth noting that, if you live in Europe everything is not lost. Although the NTSC standard is not compatible with the PAL or SECAM standards many in Europe told me that most recent TVs sold there supports NTSC.

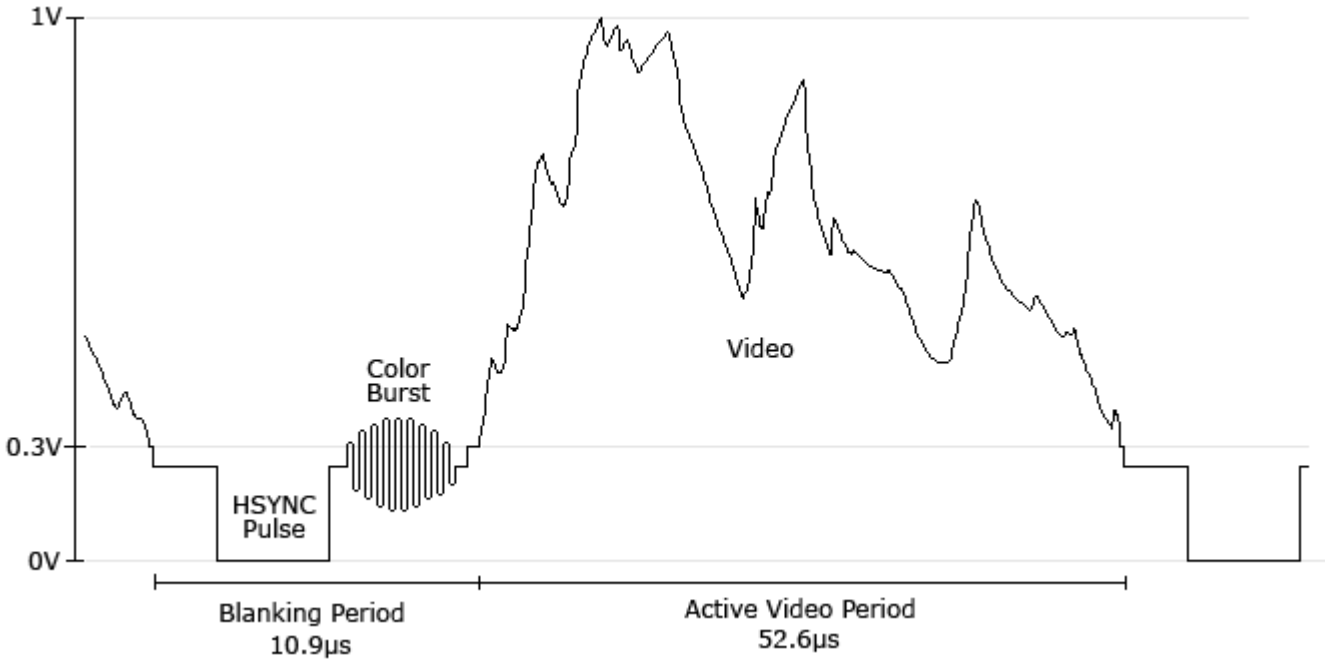
Up to recently, with the advent of LCD and plasma TVs, we were still using the cathode ray tube (CRT for short) as the basis for picture display. The CRT is a vacuum tube containing an electron gun (a source of electrons) and a fluorescent screen, with internal means to accelerate and deflect the electron beam, used to form images in the form of light emitted from the fluorescent screen.
A video signal is made of 30 frames per second (29.97 to be exact). A frame is the actual picture that fills the screen of your TV. Each video frame is in turn made up of two fields; the 'even' and 'odd' fields. Fields in turn are composed of scanlines, 524 scanlines in total (normally 525, more on that later). Fields are a remnant of the past when technology would not allow the screen to redraw enough times per second to avoid flickering. Engineers decided to draw half a picture at a time, first a field with only the odd lines, then they draw only the even lines. Hence, fields are drawn at a rate of 60 times per seconds. This interlacing of the video lines coupled with the phosphor coating persistence (the time that the phosphor continues to glow after being hit by the electron beam), blends to the human eye, creating smooth motion and imperceptible flickering.



A field begins rendering at the top-left corner of the screen and the electron beam crosses the screen from left to right rendering the video line. When it reaches the right side, the beam is turned off and brought back the left side. That's called the horizontal blanking. When the beam's back to the left side, the next line begins rendering and this continues until the 262nd line is completed rendering. The beam is then turned off and brought back to the top-left corner during what is called the vertical blanking. Then the next fields begin the same process. Here's what a field rendering looks like:

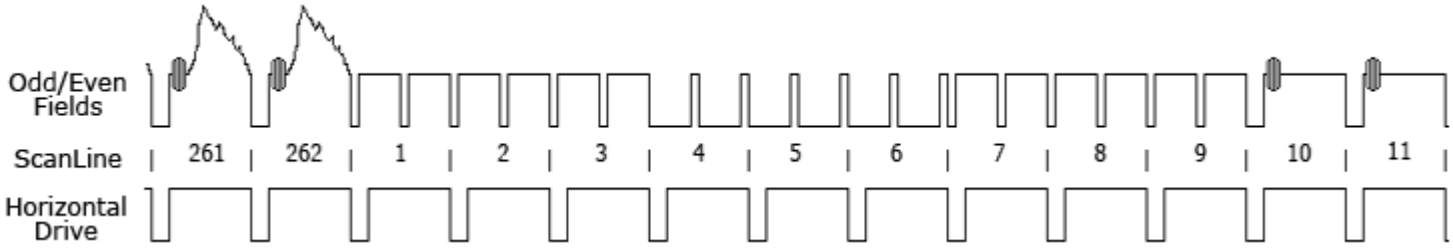


TVs require some signals to indicate them when to perform horizontal and vertical blanking. That's the synchronization signal mentioned earlier. A full composite video signal fits within one volt. The "normal" or reference voltage is at around 0.3v. The synchronization signals, or sync for short, are "negative going" pulse going to zero volts. The active video on the other hand, "sits" on the reference voltage and swings from .3 to 1 volts. The typical signal for a scanline look like this:



From that picture we can notice a few things. A scanline is made of two main sections. The blanking interval and the active video period. The blanking itself is composed of the horizontal sync pulse (called HSYNC) and a color burst. Color generation in NTSC is tricky and the explanation of how it precisely works is way out of the scope of this documentation. With that said, I'll risk a simple explanation. When TV came out it was only monochrome, the active video part of the signal represented the luminance or intensity of the picture. When color came, the NTSC engineers wanted to embed color (or chrominance) in the same signal and have it in a way that would be backward compatible with all the B/W TVs out there. There was a short period of time after the HSYNC pulse and beginning of the active video still free, so they put in there a color burst. This burst, made of 9 cycles at 3.57Mhz, is used to synchronize the TV circuitry with the chrominance signal. The chrominance is a 3.57Mhz signal modulating the luminance signal. The way they attained color was by changing the phase of the chrominance modulation. Although this scheme had its issues (some said NTSC meant Never The Same Color), it was clever enough to work on B/W television without major artifacts.
The last part to be covered is the vertical blanking. The TV circuitry needs way to know when the last line was completed and the beam needed to go back to the top. This comes with a series of special sync pulses called the vertical sync or VSYNC. Theses pulses are output at exactly double the line rate. There are 6 pre-equalization pulses, 6 serration pulses and 6 post-equalization pulses. No color burst are issued during VSYNC. After the 18th pulse (or the 9th line) regular HSYNC pulse and color burst resume.

Here's the vertical sync signal. It happens at a rate of 60Hz and signals the beginning of each field. The horizontal drive is just an internal reference corresponding to the line rate.



To conclude, here's something I wish I had known before and that NO other tutorials ever mentioned. It's worth noting that the official NTSC RS170 standard mandates that each field is made of 262.5 lines (yes, 0.5 line! hence 525 lines in total). This means that the raster beam stops mid-way on the last line and after the vertical retrace is complete, it starts mid-way on the first scanline on the next field. In my very first prototype, this sync stuff was rather confusing to me so I played safe and used a video sync generator chip. To my great deception, the picture was flickering a lot and I could not explain why. I dug out and plugged in by good old NES and SNES and notice their image were not flickering at all! Mesmerized, I turned on my oscilloscope and carefully examined their signals. It turns out that the problem was that 0.5 line! There was no such thing on those consoles, fields would contain exactly 262 lines, no more. Then I wrote the timing in software with those 262 lines per field and...success, no more flickering! According to a real video engineer: "Essentially what you're doing there is by dropping the half line, the TV is *not* scanning in interlaced mode. Basically it's progressive scan." So there you are.

3 Hardware

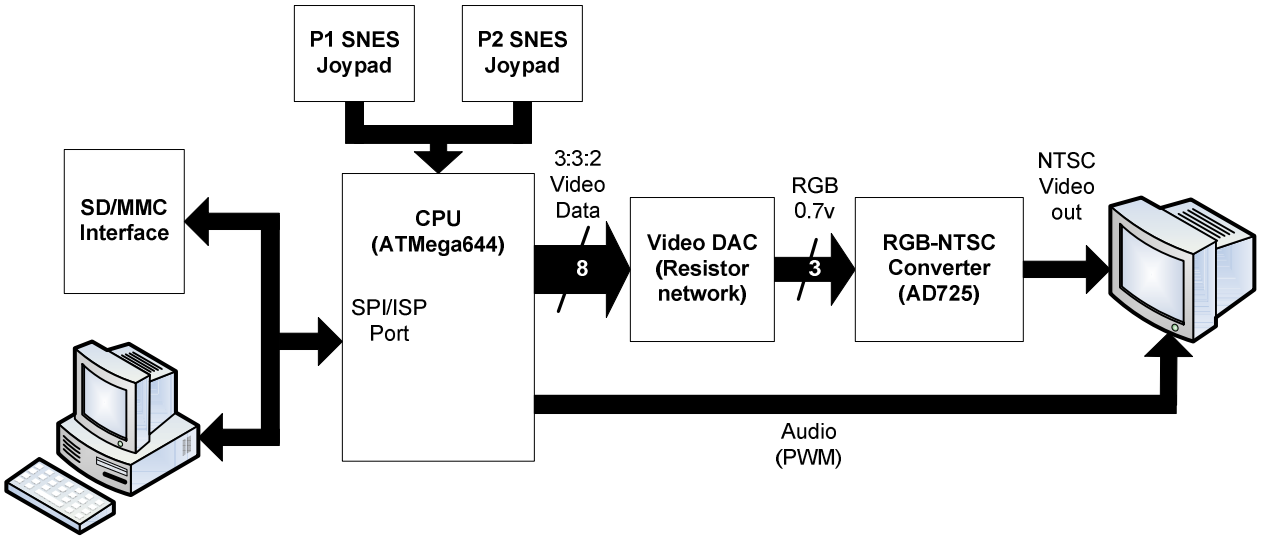


Figure 1:Uzebox block diagram

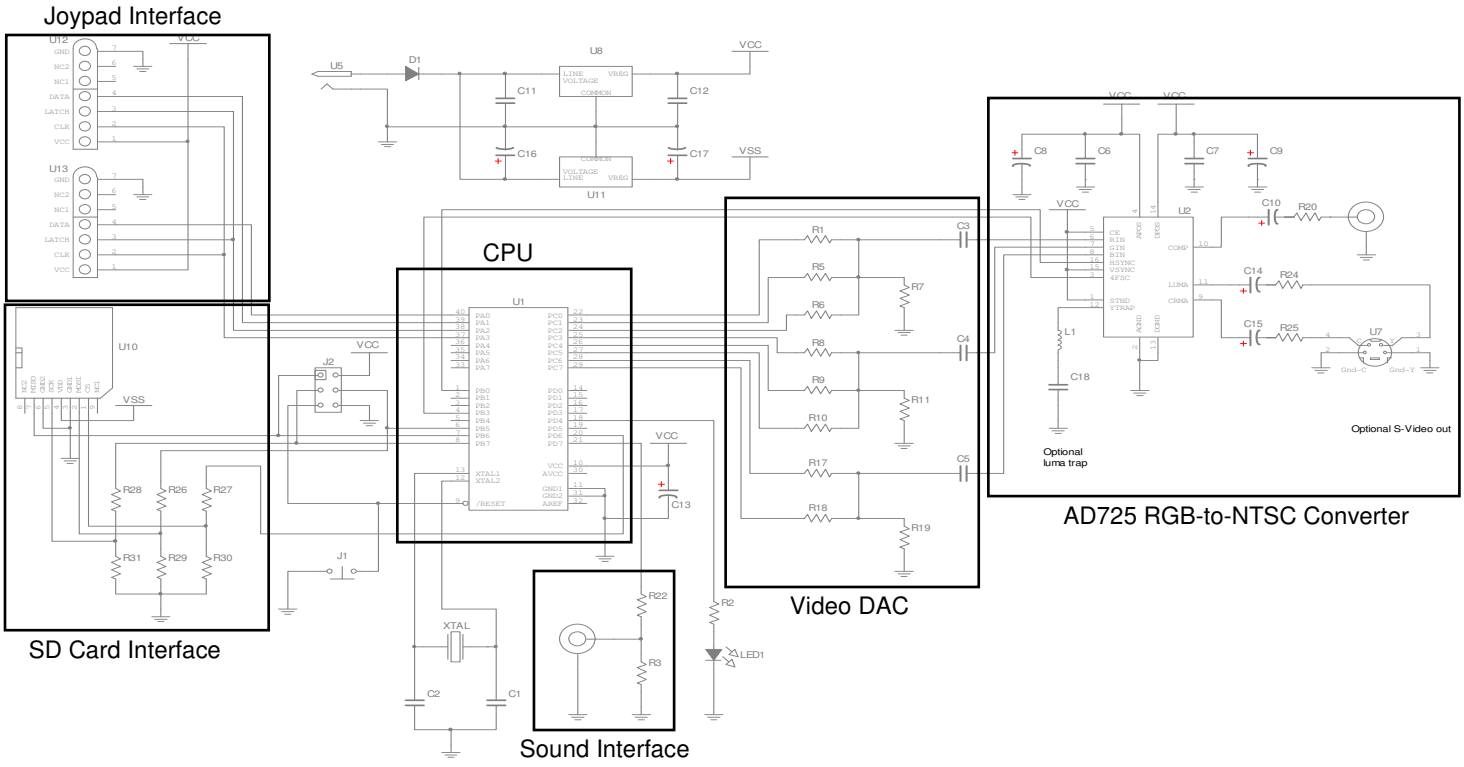


Figure 2: Block diagram-to-Schematic mapping

3.1 The CPU

The hearth of the system is the Atmega644. Its job is quite simple: do everything! Well, of course, everything except the NTSC color generation. As mentioned earlier, this would have been quite time consuming for such a low-end MCU. So what means almost everything?

- Generate the video synchronization signals. The AD725 doesn't do that, only the color modulation
- Render the picture. This include scrolling and determining sprites transparency against the background
- Mix and output the music and sound effects
- Read the joystick buttons states
- Read and handle the UART
- Handle accesses to the SD card
- And last but not least...run your games!

The most important thing coming out of the ATmega is the graphics 'pipeline' (PORTC) onto which pixel data is output. Each pixel needs exactly one byte to represent its color. 3 bits are allocated to the red component, 3 bits to the green component and 2 bits for the blue. Those are fed to a DAC composed of three R-2R resistor ladders. In theory, this gives 256 simultaneous colors. Though in practice there are a number of colors that are very close to each others.

It's worth noting that the MCU is rated at 20Mhz but runs overclocked at 28.6Mhz. This is a direct result of using the AD725 and its requirement for a 14.3Mhz clock (4 times the NTSC color burst frequency, more on that later). To avoid video aliasing, the MCU and AD725 clocks must be synchronized and either run at identical or multiple of each other. Since it seemed a waste to downgrade the Atmega644 to 14.3Mhz, I tried the other way around by overclocking it and, to my amazement, it handled it flawlessly.

3.2 The DAC

A DAC, or digital-to-analog converter, is a circuit that converts a binary value to a proportional output voltage or current. For example, if your DAC has a maximum output of 1V, and you feed it a binary value of 128 (half the maximum value that can hold a byte), you'll get 0.5 volt at the output. There is several ways to implement a DAC. You can take off-the-shelf chips or build one out of resistors. With the resistors approach, there are two main types: R-2R and weighted which the Uzebox uses. Explaining how that works is out of the scope of this document, so I'll encourage readers (or writers of this book ;-) to search the internet.

The Uzebox DAC is in reality composed of 3 smaller ones; one for each color component. If you look at Figure 3, you'll see them, each boxed with the color they represent. You can see the red and green each have 3 wires coming in from the MCU, while the blue component only has 2. Also, if you look closely, you will notice that each block's output goes to an input pin of the AD725, which correspond to the appropriate color. In between, those little capacitors (C3-C5) are used to block DC voltage from entering the AD725.

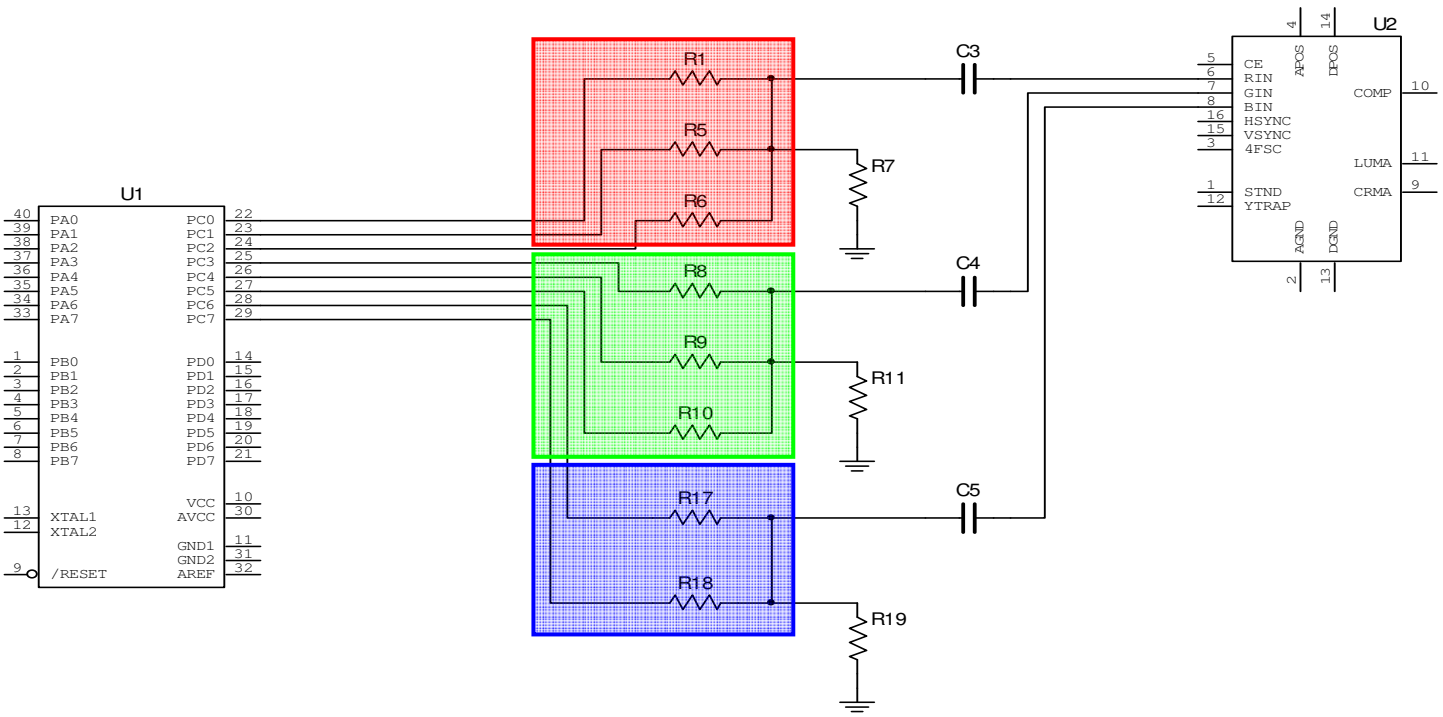


Figure 3: Video DAC

So in the end, the DACs simply converts binary values that represent color components(0-7 for red and green and 0-3 for blue) to voltage ranges (0-0.7V) suitable for the AD725.

3.3 The AD725

As mentioned earlier, NTSC composite color generation is not trivial. And especially not efficient in pure software. The AD725 (and the more recent AD723) take out that problem from the equation. It accepts as input discrete voltages for each RGB color components and, internally, converts them to the appropriate luminance and chrominance information. It also generates the color burst, adds it to the composite sync generated by the MCU, does some filtering and outputs composite video and S-video simultaneously. No hassles and guaranteed stable results.

The AD725 requires two more input to perform its job. One is the composite sync signal generated by the MCU. The other one is a clock signal at 14.31818Mhz. This clock happens to be four times the frequency of the color burst and is absolutely required in order to generate color. That clock frequency also happens to be exactly half that of the MCU. This is no coincidence. The MCU has a timer set to toggle an output pin at half its main clock.

All circuitry immediately surrounding the AD725 is taken directly from the datasheet's reference design so please refer to it for more details.

3.4 Sound

The console's sound "system" is very simple. It consists of a mono signal output on a single pin via pulse width modulation (PWM). A simple resistor divider at the output (PD7) insures the TV receives a 1 volt peak-to-peak signal.

3.5 Joypad interface

The beauty of the SNES interfaces is its simplicity and ease of integration. In each SNES joystick there's a 16-bit shift register. The way it works is simple. When the console wants to read the buttons state, it strobes the latch line on the shift register to load and hold all the button states. The console then starts toggling the controller's clock line and 16 bits are serially shifted into the console's data line. No external parts required, and only four I/O lines are required to support the two joysticks. Can't be much simpler, really. Note that the NES interface is identical except it's shift register is only 8 bits wide.

For more details, have a look at this very good NES/SNES interface document from Parallax:
http://www.parallax.com/dl/docs/prod/prop/Hydra-Ch6All-v1.0.pdf

3.6 SD Card Interface

The SD card interface is also very simple. It requires only few external components. A SD card socket, a few resistors to act as voltage translator (the Atmega644 lines output 5V and the SD must receive 3.3V) and another voltage regulator (since the SD card works at 3.3 Volts).

Everything else that could be remotely complex is part of the SD card itself! To make things even simpler, the SD cards support communication with microcontroller using the SPI protocol.

4 Software

4.1 Kernel

When running it appears the Atmega644 is performing many tasks at once, like video rendering, reading the controller and playing music. All these operations are in fact executed sequentially but fast enough to appear simultaneous. The set of these core, low level functions onto which the API and game depends is referred to as the kernel. It is responsible to:

- Initialize ports, timers and other hardware peripheral upon reset
- Generate the composite video synchronization pulses required by the AD725
- Decode the music score data and process sound effects
- Mix sound samples for the four voices
- Output sound sample from the mix buffer at a regular interval
- Read the controllers buttons and mouse movement
- Read the UART for inbound data and store it into a buffer

The kernel perform its work in a background task called an *interrupt*. As the name implies, this task interrupts the main program by saving it's state, doing some work, restoring state and resuming the main program.

Interrupts allows the main program to stay simpler and run asynchronously to the kernel not having to care about its complex and time sensitive tasks.

With only 64K of flash memory, great attention must be made to keep the kernel as small as possible to leave the main program with enough room to implement interesting games. The use of compile-time switches are used heavily in the kernel. These switches are specified in the Makefile and will conditionally include or remove code, allocate varying amount of bytes for the video memory, select the video mode, disable the Uzebox logo, etc.

4.1.1 Initialization

Upon initialization, the kernel will configure many thing, most importantly a timer which will generate interrupts at 15.7Khz, the NTSC scanline frequency. That is, the kernel will be invoked for every scanline of a video picture (in fact not all of them, more on that later). Figure 4 gives an high level view of the kernel's initialization process.

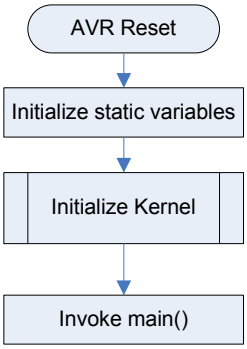


Figure 4: Kernel initialization process

Figure 5 describes the initialization function in more details. Upon it's completion, the main program will start getting interrupted by the kernel.

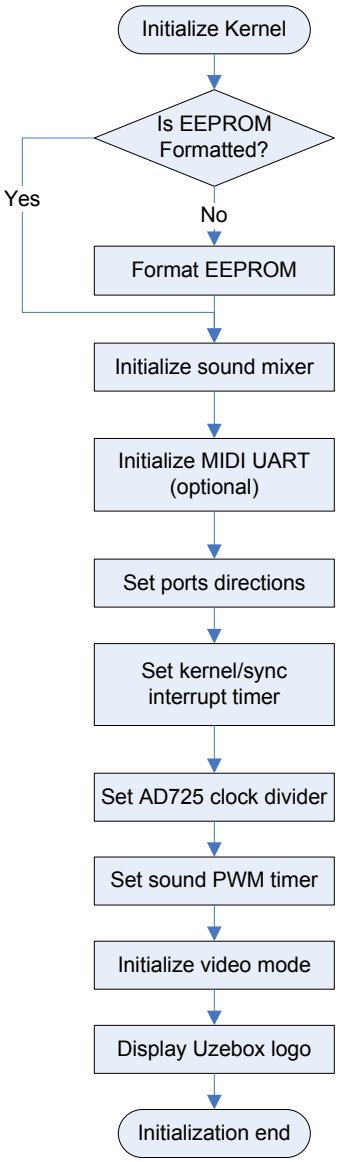


Figure 5: Kernel initialization function

4.1.2 Kernel Interrupt

As mentioned earlier, the kernel interrupt is responsible to pause the main program in order to render video and mix music. The kernel interrupt is completely written in assembler because of speed and timing requirements. Timing is particularly critical here since just a single clock cycle in excess or missing can cause a drift in the resulting video signal. Eventually as this error accumulates, the TV will have problems to sync and the picture will suffer jitter and/or stuttering.

The development of the interrupt was the most time consuming (and sometime frustrating) part of developing the Uzebox software. Using AVR Studio's simulator cycle counter was crucial in timing up this code. Without it I would have been in the dark!

Figure 6 describes the kernel's interrupt process.

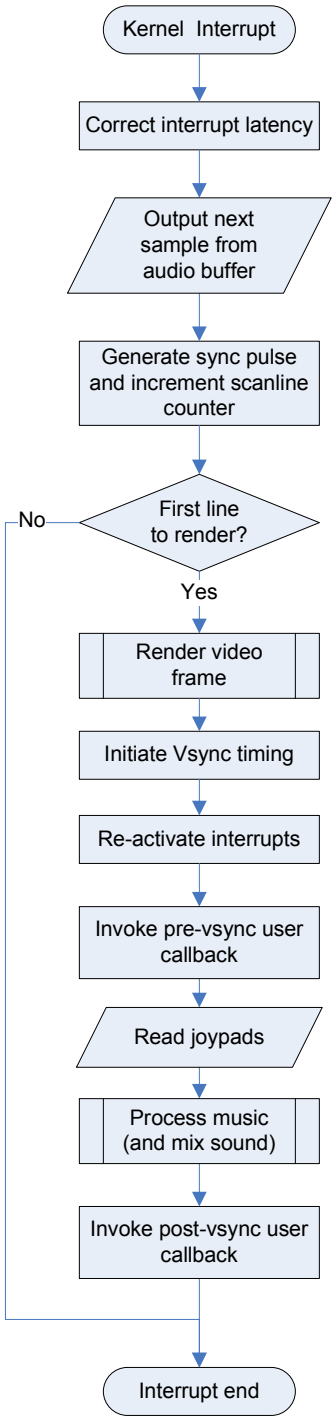


Figure 6: Kernel interrupt process flow

The kernel uses Timer1, the 16 bit counter, to trigger interrupts. The timer starts at zero and counts up to 1820 at which point it will automatically roll over to zero and generate an interrupt. The 1820 value comes from dividing the main crystal frequency (28.63636 Mhz) by the NTSC scanline rate (15.73426 kHz).

4.1.2.1 Latency correction

The AVR does not have a deterministic interrupt response time (cycle count) so we can never know for sure how many cycles have elapsed before executing the interrupt's first instruction. This comes from the fact that AVR instructions takes between 1 to 4 cycles to execute and will always complete before the AVR acknowledges an interrupt. If not corrected, the picture will experience jitter. Fortunately, the way to fix this is pretty simple. We read Timer1 value right at the beginning of the interrupt and subtract from it a fixed number corresponding to the minimum interrupt latency in cycles. From there remains zero or more "extra" latency cycles that will be used in a special loop to correct latency. After this loop execution, will we always be at a known state, say exactly 100 cycles after Timer1 rolled over.

4.1.2.2 Synchronization and sound output

The kernel will then generate synchronization pulses and output sound. Sync pulses are negative going, so they are generated by toggling the sync pin from one to zero, wait a finite amount of cycles, then toggle back to one. In order to minimize wasted cycles "just waiting", some work is actually performed in between toggles. The next sample from the audio ring buffer is read and output to the audio PWM port. Also, if the UART RX is enabled (i.e.: for MIDI), the kernel will move received data into a buffer. Note that both these operations are handed tuned assembler to fit within a sync pulse (i.e.: about 63 clock cycles). After generating the sync pulse, the kernel will check if it has reached the scanline at which the picture begins. If so, it will invoke the frame rendering function and initiate VSYNC. Otherwise, the interrupts completes and returns control to the main program.

4.1.2.3 Frame rendering

Frame rendering consists in drawing a complete video picture for $1/60^{th}$ of a second. Since the Uzebox uses 224 lines of resolution and a TV has 525, the same frame is rendered for both odd and even video fields. Frame rendering uses one of the many available *video modes*. Each video mode implements a different algorithm to use the limited resources in order to produce a picture. They each have trade-offs that must be. For instance, video mode 3 support sprites and full screen scrolling but takes significantly more memory and CPU cycles when compared to mode 1, which does not. So far, eight video modes has been implemented each with is very unique particularities. Due to many factor amongst other complexity and flash memory limitations, the kernel currently support only one video mode per game. This mode is used via the VIDEO_MODE compile-time switch. Although the most widely used and powerful video mode, mode 3 is also the most elaborate an complex one. So we will instead describe the most simple one, mode 1, the same one used by Megatris.

Conventional graphics adapter like VGA used a frame buffer to hold picture information. Each and every discrete point on the screen (pixels) took one byte in the frame buffer to describe its color. The classic 320x200 pixels mode hence required 64K of RAM to hold a complete picture. So how can the Uzebox that doesn't have a frame buffer, less have 64K of RAM, still manages to render a 240x224 picture in 256 colors? With an old trick used by the first generations of console (like the NES) called *tile-based* rendering. Similar to a text-based video mode using a matrix of character indexes instead of discrete pixels, the picture will be made of repeating patterns called tiles.

Mode 1 is precisely a *tile-based* video mode. Although the final rendered picture is 240x224 pixels, internally it is represented by a 40x28 array of 16bits pointers to 6x8 pixels tiles in FLASH. The VRAM (video RAM) will directly contain relative memory pointers to the tile to be rendered. This contrasts with some other video modes

which stores 8bits *indexes* in VRAM and can display no more than 256 tiles simultaneously. So while a 240x224 image would require ~53K of VRAM, this 40x28 array of (16-bit) pointers consumes only ~2.2K.

Figure 7 shows the how the memory is organized in a tile-based VRAM. Tiles pixels are stored linearly in FLASH one row after another. Figure 8 shows a 6x8 pixel tile as used by mode 1. In Figure 9, we can see that the different rows that makes a tile are store consecutively in FLASH memory.

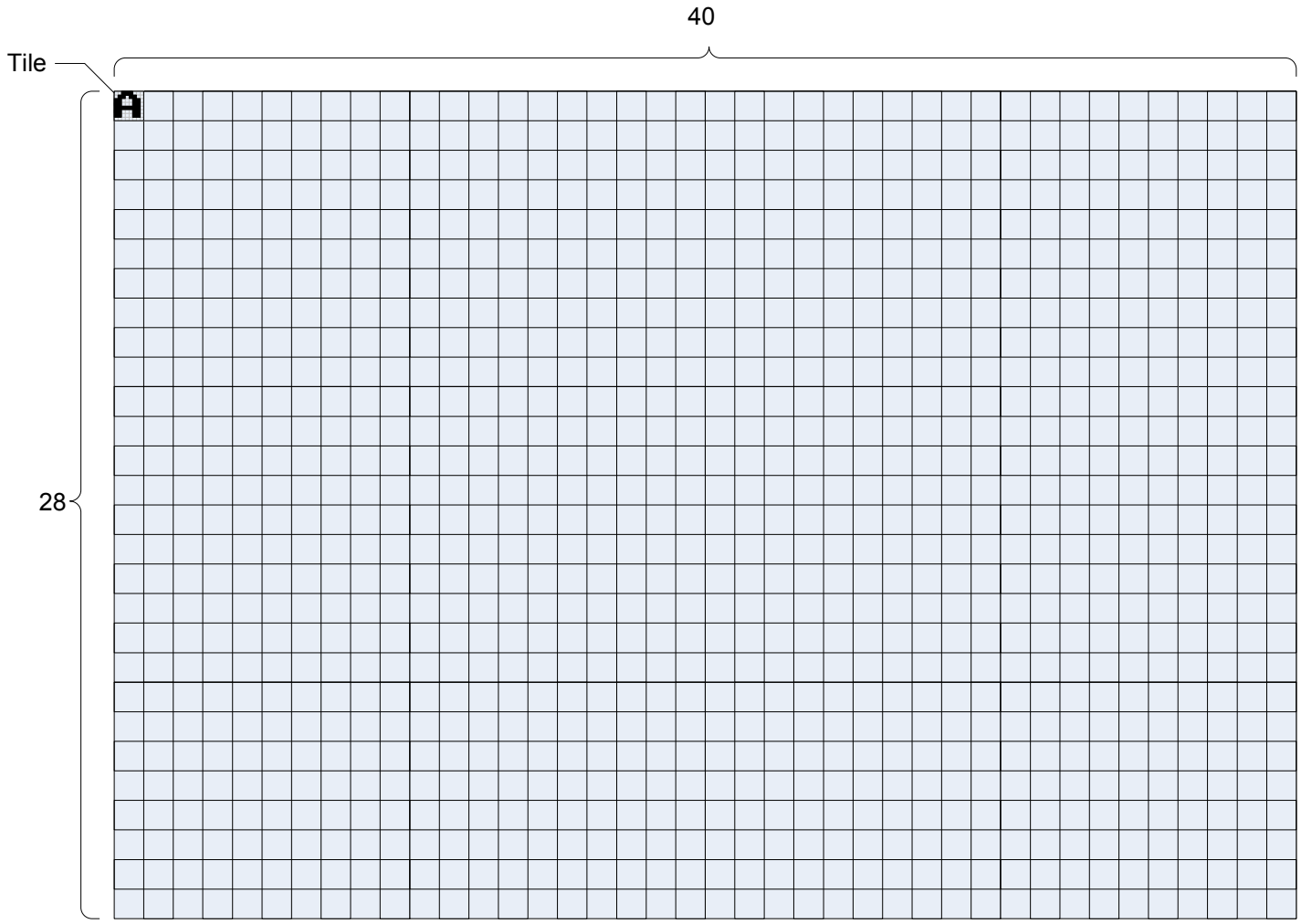


Figure 7:Tile-based VRAM in video mode 1

During rendering, the VRAM is traversed row by row and each tile pointer is used to retrieve it's pixels.

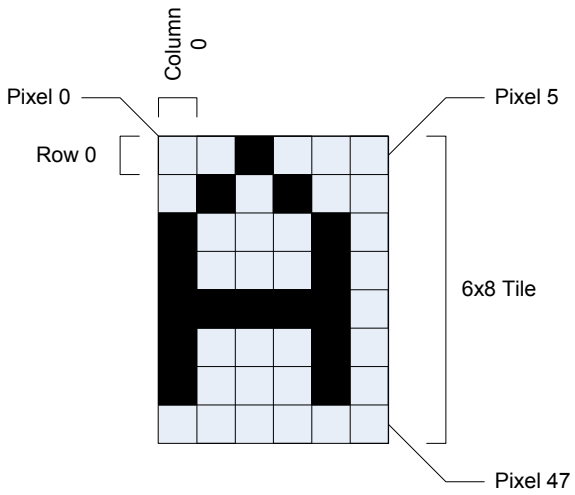


Figure 8: Tile

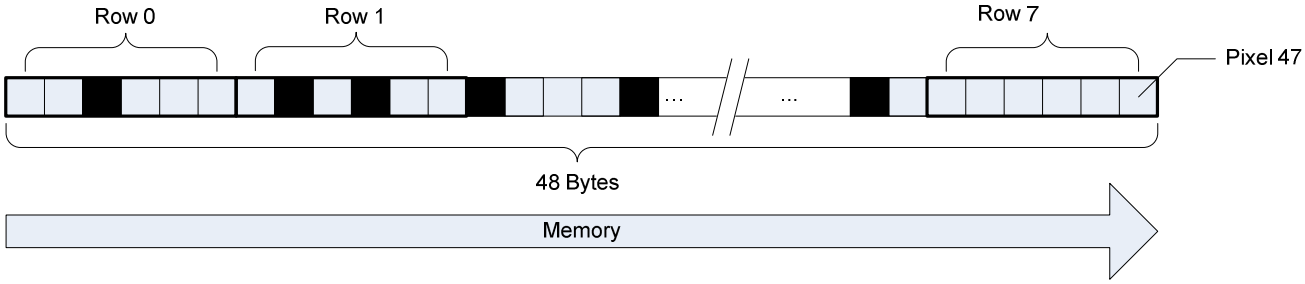


Figure 9: Tile pixels storage

Once the kernel reaches a defined scanline (FIRST_RENDER_LINE), frame rendering will begin. During this phase, all scanlines will be drawn for a full field before control is returned and the interrupt completes. This is because there is not enough time during HSYNC to restore all registers, get back to the main program et re-acknowledge another interrupt. This means that timing will be extremely critical. Each and every line, including the sync code must take exactly 1820 clock cycles.

The frame rendering function is made of a main loop that will maintain variables (VRAM pointer and the tile row offset) and call the scanline rasterizer for each video line. The rasterizer will then load the VRAM for the next tile pointer to be rendered for this scanline and start fetching and outputting pixels for this tile. Intertwined with this, the code will fetch the next tile address and compute it final address. This process is repeated 40 times, the width of the screen in tiles. The rendering process is illustrated in Figure 10.

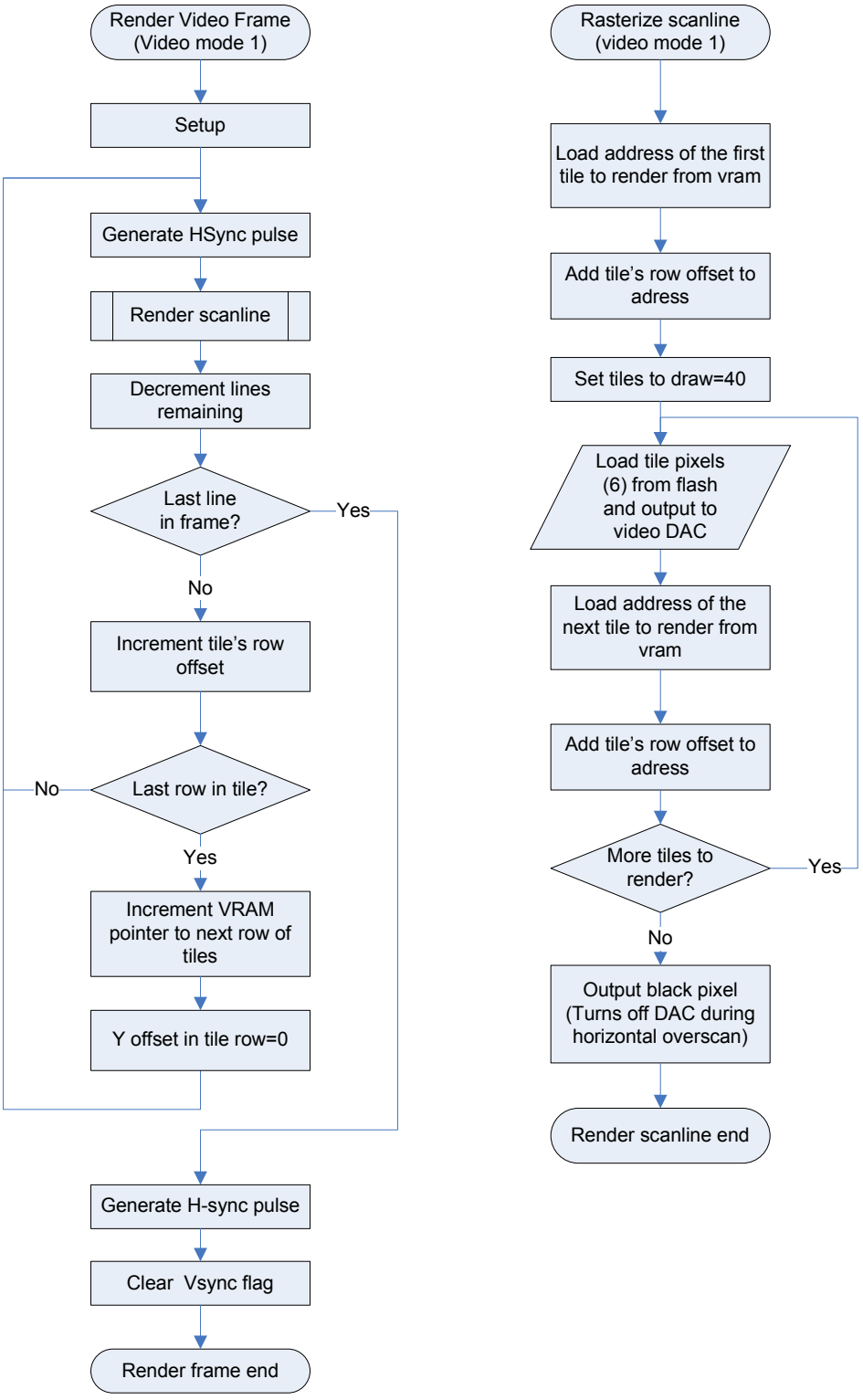


Figure 10: Mode 1 rendering process

The following listing is the scanline rasterizer for mode 1:

```
;***********************************************
; Renders a line within the current tile row.
; Draws 40 tiles wide @ 6 clocks per pixel
;
; r22     = Y offset in tile row (0-7)
; r23     = tile width in bytes
; Y       = VRAM adress to draw from (must not be modified)
;
; cycles  = 1495
;***********************************************
render_tile_line:

        movw XL,YL                      ;copy current VRAM pointer to X

        ;/////////////////////////////////////////
        ;Compute the adress of the first tile to draw
        ;/////////////////////////////////////////
        ld      r20,X+                  ;load absolute tile adress from VRAM (LSB)
        ld      r21,X+                  ;load absolute tile adress from VRAM (MSB)
        mul r22,r23                     ;compute Y offset in current tile row
        movw r24,r0                     ;store result in r24:r25 for use in inner loop
        add r20,r24                     ;add Y offset to tile address
        adc r21,r25             ;add Y offset to tile address
        movw ZL,r20             ;copy to Z, the only register that can read from flash

        ldi r18,SCREEN_TILES_H ;load the number of horizontal tiles to draw

mode1_loop:
        lpm r16,Z+                      ;get pixel 0 from flash
        out VIDEO_PORT,r16      ;and output it to the video DAC

        ld      r20,X+                  ;load next tile adress from VRAM (LSB)

        lpm r16,Z+                      ;get pixel 1 from flash
        out VIDEO_PORT,r16      ;and output it to the video DAC

        ld      r21,X+                  ;load next tile adress from VRAM (MSB)

        lpm r16,Z+                      ;get pixel 2 from flash
        out VIDEO_PORT,r16      ;and output it to the video DAC

        rjmp .                          ;2 cycles delay

        lpm r16,Z+                      ;get pixel 3 from flash
        out VIDEO_PORT,r16      ;and output it to the video DAC

        add r20,r24                     ;add Y offset to tile address
        adc r21,r25             ;add Y offset to tile address

        lpm r16,Z+                      ;get pixel 4 from flash
        out VIDEO_PORT,r16      ;and output it to the video DAC

        lpm r16,Z+                      ;get pixel 5 from flash
        movw ZL,r20                     ;load the next tile's adress in Z
        dec r18                         ;decrement horizontal tiles to draw
```

```
out VIDEO_PORT,r16      ;and output it to the video DAC
brne mode1_loop

rjmp .                          ;2 cycles delay
nop                                     ;1 cycle delay
clr r16                         ;set last pixel to zero (black)
out VIDEO_PORT,r16

ret
```

4.1.2.4 VSYNC

When the picture finishes rendering, VSYNC will be initiated. Since this phase happens before the next frame to be rendered it is the perfect time to "prepare" stuff in advance. Such things will include reading the controllers, mixing music and blitting sprites (for modes that supports it). Before these happens, the kernel will be the user program a change to read or change kernel parameter via a user callback. Another callback will be invoked when all the kernel's VSYNC processing is complete just before returning control to the main program. The video timing will also be updated to generate the equalization and serrations pulse necessary to indicate to the television that a new frame is to begin.

4.1.2.5 Read Controllers

Controllers buttons for both ports and the SNES mouse are read during VSYNC. The function simply strobes the latch line to store the buttons states in the controller's shift register and clock the bits in to be read and packed into two integers. These integers can then be read using the ReadJoypad() API function.

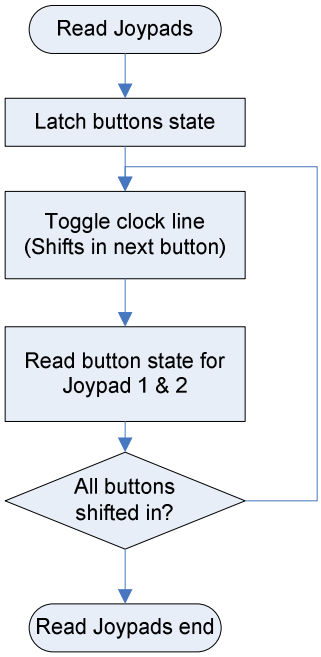


Figure 11: Controller read process

4.1.2.6 Music Processing and Sound Mixing

Another important step that happens during VSYNC is the sound processing. This is separated in two logical steps: the music processing and the actual low level mixing of the individual voices or sound channels. Before getting into theses processes, we will explain the sound engine general concepts.

The Uzebox sound engine does not use any hardware synthesizers, the music is 100% rendered by software. The engine has 4 independent sound channels, three melodic and one noise channel. Each have their own independent waveform, volume and pitch (for the melodic channels).

Music processing refers to the reading of the musical score and the processing of envelopes, tremolo and other high level functions in order to set the parameters of the low level mixer. The music replayer is designed to play MIDI streams. MIDI is a very compact and space efficient format for music. It is made of a continuous stream of events each of which is separated in time using a number of 'ticks' (a delta-time value). Events can be notes, tempo change, modulation change, etc. and can be associated to a specific channel or be global (like tempo events). Instruments are supported in the form of “patches”, and old term referring to the patching of cable in the very first synthesizers to obtain different sounds. Uzebox patches are a stream of byte commands that can alter the timbre, pitch or volume of a voice. Since these commands are read once per frame (60hz) , complex and intricate sounds can be produced. Patches are simple C byte arrays and typically look like this:

```
//FX: "Echo Droplet"
const char patch01[] PROGMEM ={
0,PC_WAVE,2,
0,PC_ENV_SPEED,-12,
5,PC_NOTE_UP,12,
5,PC_NOTE_DOWN,12,
5,PC_NOTE_UP,12,
5,PC_NOTE_DOWN,12,
5,PC_NOTE_CUT,0,
0,PATCH_END
};
```

The first parameter is the “delta-time” or the delay in frame until the next command will play. The second byte is the actual command. The third byte is the command’s parameter. Note that patches are not only used to play music but also used to defined sound effects in a game. The TriggerFX(patch, volume) API call will be used to trigger the play back of that patch.

When the conversion process is complete, it is pretty easy to play songs. Simply initialize the engine using:

```
InitMusicPlayer(myPatches);
```

Then start the song using:

```
StartSong(mySong);
```

Stop/pause the song using:

```
StopSong();
```

And resume where you last stopped with:

```
ResumeSong();
```

Figure 12 describes the music player process.

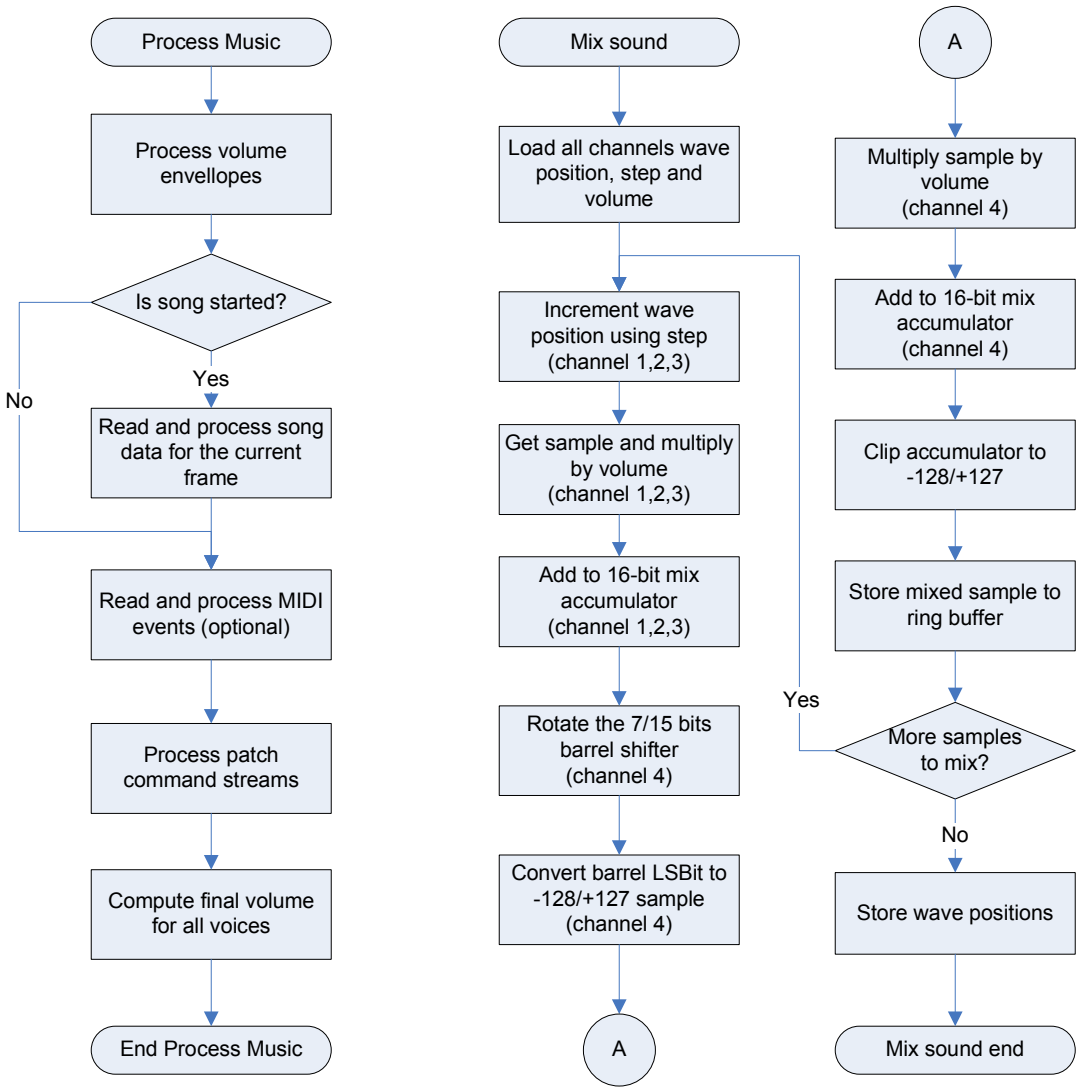


Figure 12: Music and sound mixing processes

Once the music has been processed, notes has been set and volume envelopes processed, control is passed to the mixer that will actually generate the tones that will be heard on the TV.

The mixer stores the computed samples to a circular buffer (also called a ring buffer). It is a byte array in RAM logically segmented in two parts (A and B). Part A plays while part B is mixed. Then at each frame beginning, they are switched and part A is mixed and part B is played. Each part contains exactly as many samples as there is scanlines in a video field (two interlaced fields makes a frame), in this case, 262. So at 15.7Khz (the NTSC line rate) during the HSYNC pulse, a byte is read from the circular buffer and output to the sound port (by mean

of PWM). HSYNC pulses happens non-stop even during blanking intervals. A whole field worth of music is mixed "one-shot" and all four channels are mixed simultaneously without resorting to a temporary 16-bit signed buffer since there's not enough RAM. Naturally, music mixer code is in assembler and, for optimal speed, all registers are used during mixing. Figure 13 illustrates the circular buffer in some arbitrary point in time.

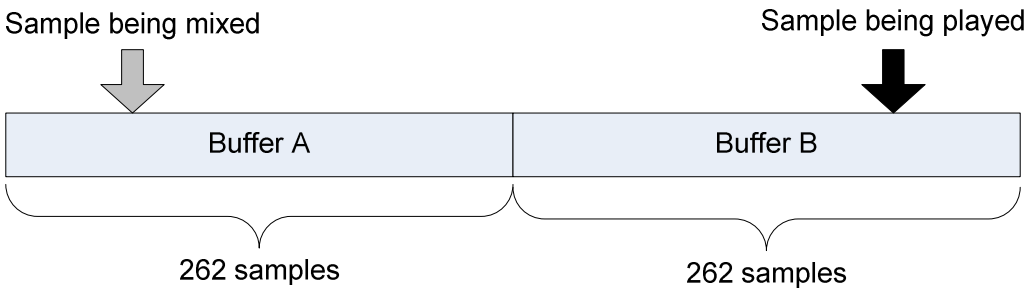


Figure 13: Circular buffer mixing

To obtain different timbre, the engine uses a table made of short, repeating waveforms for the first 3 channels. Each wave is exactly 256 samples long (8 bits signed) and are forced-aligned on an 8 bit boundary in ROM. Because of this, we only need a 8 bit pointer for the waveform's position. Position will wrap automatically, effectively giving "free-running oscillators". Using a any sound tool, it is easy to create waveforms that can vary from a simple square wave to a sine or filtered triangle. The 4th channel is the noise channel and implements a switchable 7/15 bits linear feedback shit register (LFSR). LFSRs are pseudo random. The 7 bits mode is more metallic sounding because all bit states are repeated each 1287 samples. The 15 bits mode sound much more like white noise because they are repeated each 32768 samples.

Pitch is done like most "MOD" players, with a "step table". This table consists of pre-calculated 8:8 fixed point values that represents the input sample's (i.e.: the 256 bytes wave) pointer increment per output sample (the mix buffer). There is one fixed point word per note, for a total of 127 notes. The wave table is composed of 256 bytes waves and each wave models exactly one "sound cycle". I.e. for a triangle wave, it would contain : /\ . Let say the mixing rate is 8Khz and we want to play a C5. We look in the step table for note 48 (C5). It says note frequency is 8Khz and its calculated stepping is hence 1.000. For each output sample, we increment the input pointer by exactly one. Now say we have a C6, an octave higher (so its double the frequency). The stepping of this note will be 2.000. That means that for each output sample, we increment the input by two samples, effectively skipping one of them. You get the idea. Note that for high stepping, a lot of sample are skipped and combined with wrapping it introduces aliasing. That can be somewhat minimized by using slow rising/ending waves. Check Figure 12 for the mixing procedure.

The MIDI music tracks are converted using a custom tool available with the project's sources.

5 Programming the Uzebox

The Uzebox uses the open source GNU GCC for AVR tool chain. On Windows you will need to install the WinAVR package. For development Atmel's AVR Studio can be used or Eclipse for C using the AVR plug-in.

Using Eclipse is better because it support GDB (GNU debugger) and hence you can step-debug directly on the Uzebox emulator.

5.1 Simple Hello World Example

The following listing uses video mode 1 do display a classic hello world, Uzebox style. As we can see, all complexity has been abstracted in the kernel's background task and the development of games are very simple and straightforward. Naturally since the Uzebox functions are 100% software, more savvy users can tweak the kernel or even invent new video modes or sound mixers.

```
#include <avr/io.h>
#include <stdlib.h>
#include <avr/pgmspace.h>
#include <uzebox.h>

#include "data/fonts.pic.inc"

int main(){

        //Set the font and tiles to use.
        //Always invoke before calling ClearVram()
        SetFontTable(fonts);

        //Clear the screen (fills the vram with tile zero)
        ClearVram();

        //Prints a string on the screen at the specified (x,y) location.
        //Note that PSTR() is a macro that tells
        //the compiler to store the string in flash.
        Print(8,12,PSTR("HELLO WORLD FROM THE UZEBOX!"));

        //Embed program must never return from main() so wait forever.
        while(1);

}
```

6 Important resources

The main project page: http://uzebox.org

A wealth of information has been written since the project went live. Tutorials, documentation, troubleshooting and more can be found on the project WIKI at http://uzebox.org/wiki.

The forum is a good place to ask question to a cheerful community always eager to help:
http://uzebox.org/forums

The Uzebox news feed: http://uzebox.org/news

EOF