MONOCHRON

Design documents

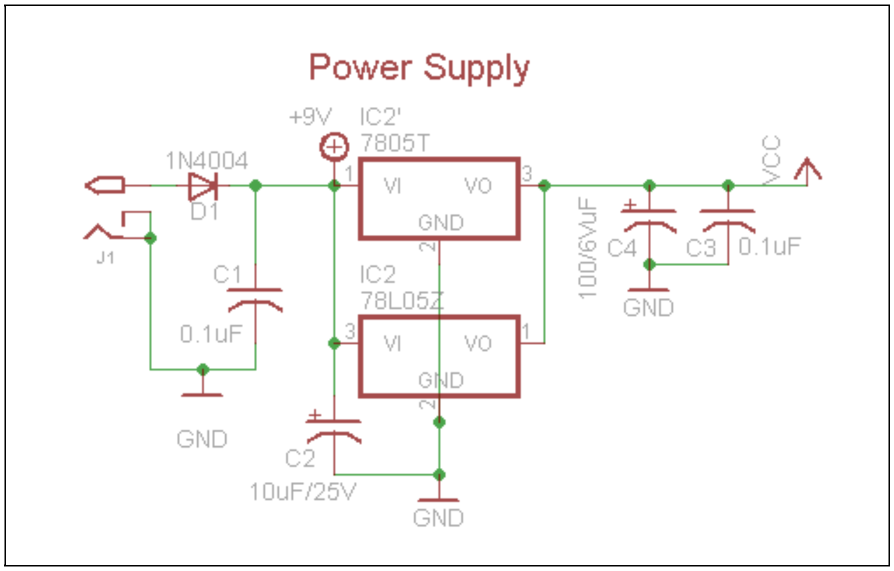
May 17, 2011 20:07

Overview

Here is a rough overview of how the kit is designed to work

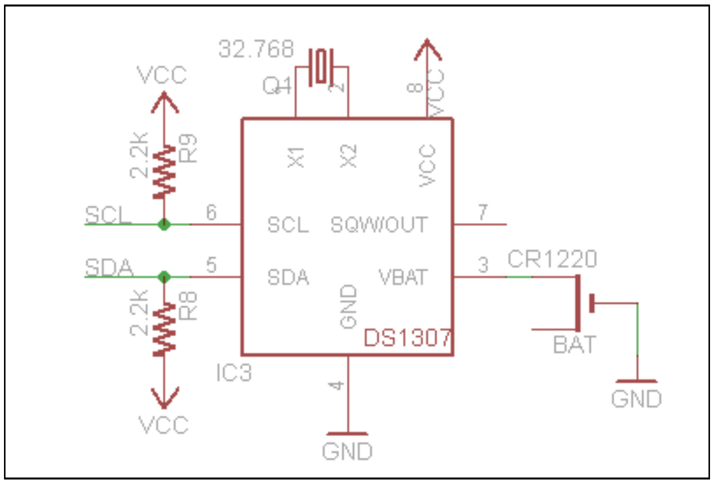
Click here for the full schematic

Power supply



The power supply is very simple. We have a DC power jack, positive tip **J1** which feeds into our polarity protection diode **D1**. After that is two capacitors **C1** and **C2** to smooth out the input voltage. Next are the 7805's. We wanted to have the option of TO-220 (1 amp or more, if you have an EL backlit display or wanted to run something hefty) or TO-92 (100mA). The kit comes with TO-92 as it only requires ~50mA of current, well within the limits of the smaller package. **C3** and **C4** are stabilization and smoothing caps for the output of the regulator

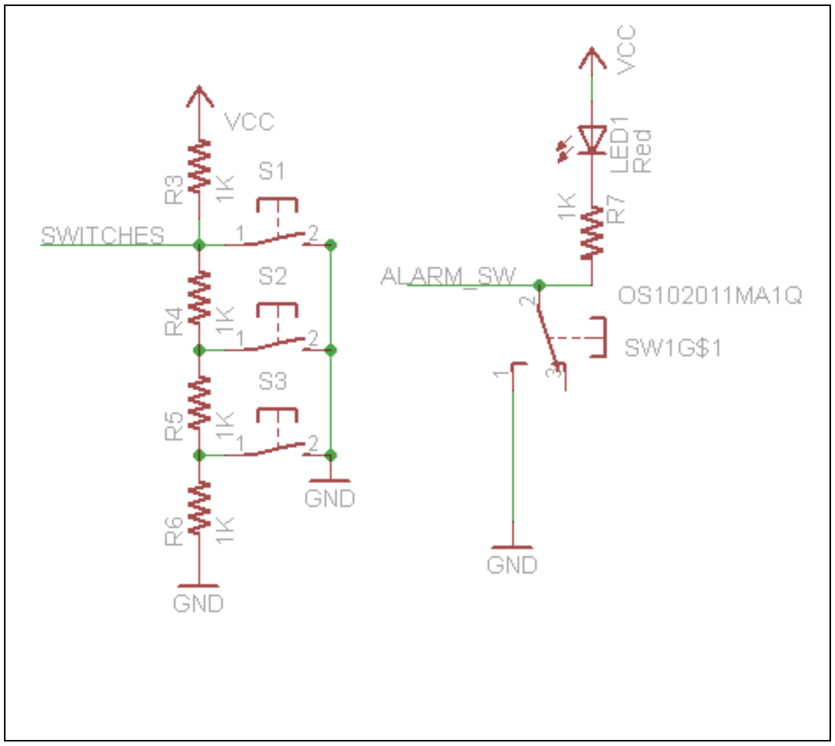
Real Time Clock



Next we'll discuss the RTC (real time clock) chip. We are using the DS1307 which is a 8-DIP clock with battery backup. Normally the clock runs off of the 5v supply. If the supply dies (say because of power loss) the chip switches to the **CR1220** battery, which can run it for *many years*. The RTC requires a 32.768KHz 12.5pF crystal **Q1** to keep time. To communicate with the main chip, i2c is used. I2c is a 2-wire protocol that uses open-drain transistors to communicate, thus we have **R8** and **R9** which are the two pull-up resistors for the open-drain inputs. **SQW** is a square wvae output that we do not use

Note that the RTC will act erratically if the battery is not placed. If you are absolutely sure you dont want battery backup, stick some tinfoil in the battery holder. Otherwise i2c will hang and it will be a real pain to figure out whats going on!

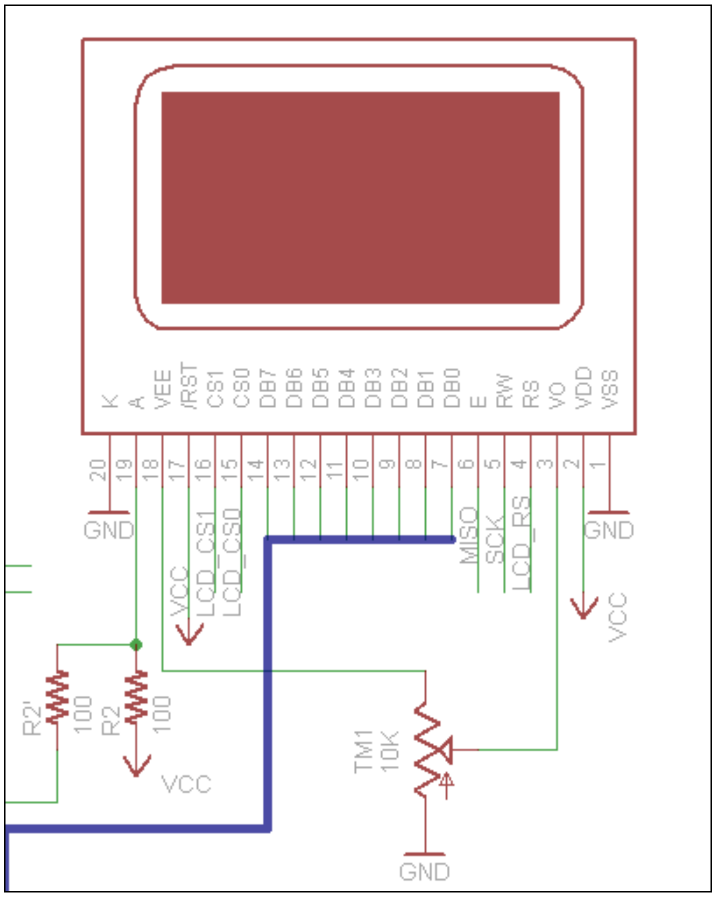
Switches



There are 2 sets of switches, the configuration tactile buttons and the alarm switch. The alarm switch is simply to indicate whether the alarm is on or off. To save a pin, the indicator LED (**LED1**) is connected directly to the switch paddle through a current limiting resistor **R7**.

Since we don't have a lot of spare pins on the microcontroller, we save 2 by having the pushbuttons connected 'totem-pole' style to the analog-digital converter. as each button is pressed, the resistor divider made of **R3-R6** changes and thus we can tell which switch has been pushed. Note that we can't tell if more than one button is pressed - in this application we dont need to so thats OK.

Display



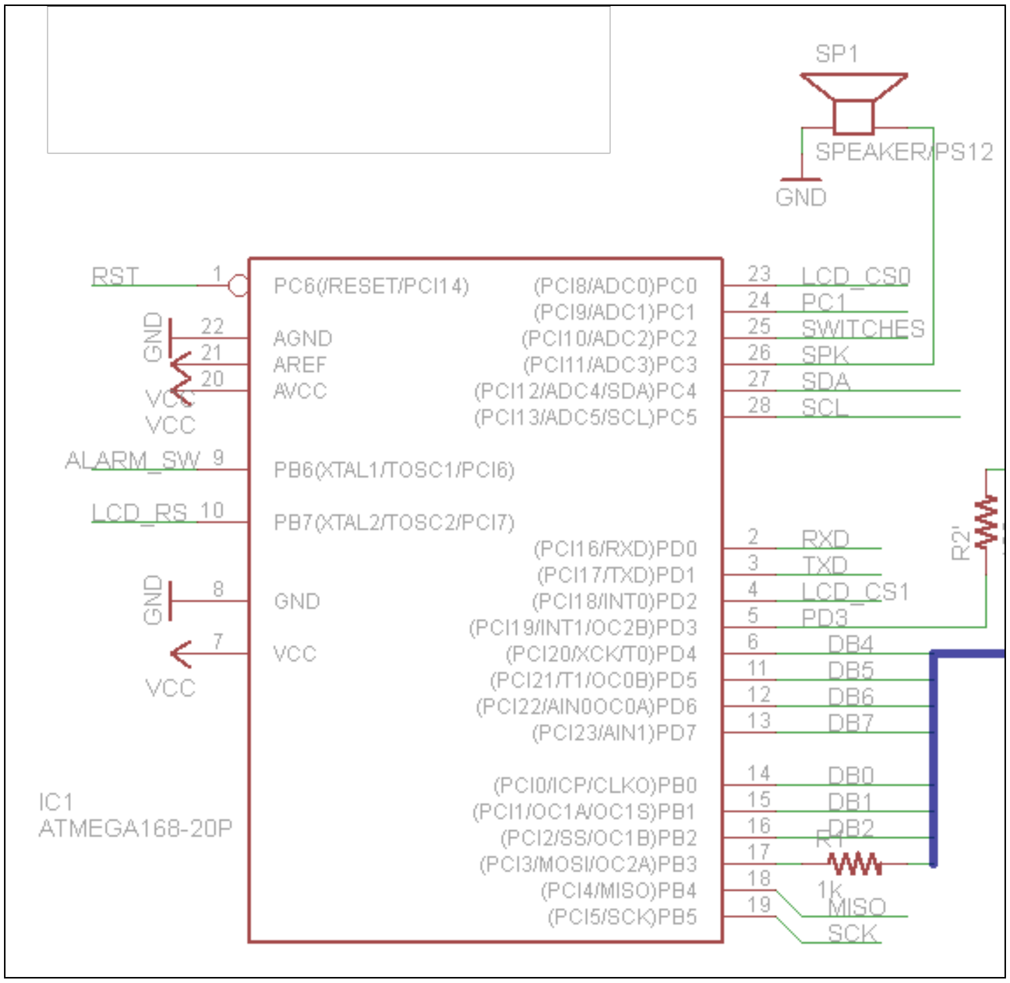
The main display is a monochrome **KS0108** based LCD with 128x64 pixels (two 64x64 pixel controllers) It is a well known and documented display. They are common in electronics shops, and run at 5V. The unfortunate thing about them is they're quite sluggish and so they need software optimization for any sort of animation.

This display has a parallel 8-bit interface and so it needs 8 bits for the port (**DB0-7**), and then 2 chip-select lines (**CS0** and **CS1**) as well as an enable pin **E**, read/write select pin **RW** and command/data pin **RS**.

V0 is the contrast pin, and must be connected to a 10K pot tied between ground and **VEE** which is a negative voltage generated by the LCD.

The backlight of the display is an LED (anode is **A** cathode is **K** and is connected to ground). We use a 100 ohm (5V - 3.4Vf / 20mA = 80 ohms or so) resistor **R2** to se the brightness. If the resistor is placed in slot **R2** we save a pin but cannot adjust the brightness. if **R2'** is used, the backlight is connected to a PWM output of the micro which allows us to control the brightness in the configuration

Microcontroller



Finally we get to the microcontroller. The microcontroller does not have a crystal, we'll be running it using the internal 8MHz oscillator which is good enough, this frees up two pins. **R1** is used because otherwise the LCD output on that pin conflicts with programming (**MISO** and **SCK** go into an input pin on the LCD)

Because of the massive # of pins, the display requires, the only free pins are **RX**, **TX**, and **PC1**. **PD3** is free if you solder **R2** into the hardwired slot (see above). If you'd like to connect something to the i2c bus, there are already pull ups on there.. If you really need an extra pin, **LCD_CS1** is always the invertion of **LCD_CS0** so you can use a transistor and a pullup to create a 'not gate' and the use either **PC0** or **PD2**.

Firmware

The firmware uses Pascal Stang's great AVR libraries. The i2c.c and i2c.h files (which are used to access the RTC) and ks0108 files are from that project. There were a few bugs and sluggish things about the KS0108 library so we made a few modifications to allow inverted drawing, fast rectangle (blitting instead of setting one pixel at a time), etc. The KS0108 library can draw basic shapes and best of all, has a 5x7 font so you can easily print text on the screen

The clock core code is in ratt.c, thats where the stuff that deals with setting and updating the time, snooze, talking to the RTC, beeping the piezo and running the main loop that animates the display.

Button debouncing and interface code is in buttons.c. The ADC runs constantly to look for changes in the resistor divider.

The configure menu system is all in config.c - its basically a state machine, you shouldn't have to modify anything there

The real drawing and any display logic code happens in anim.c. step() is called every TICK milliseconds, and the microcontroller does the work of figuring out where the 'ball' is heading and then whether each 'stick' should attempt to hit it or miss it (depending on whether the time changed)

If you want to design a clock display, you would pretty much just want to edit anim.c and have the init() and step() code change