Content:

A digital DC powersupply -- part 2: the software

Abstract:

This is the second part in the series about the digital powersupply. You might want to read the first part first.

There will be a third part where we add i2c communication to control the powersupply via command from the pc and maybe a fourth part where more fancy things are added. I am thinking of not only producing DC voltage but also DC + pulses and spikes. This way you can test circuits to make sure that they are resistant to noise and variations in power.

The hardware is available from shop.tuxgraphics.org.

Introduction

Using a clever microcontroller based design we can build a power supply which has more features and is a lot cheaper than traditional powersupplies. This is possible because functions which are traditinally implemented in hardware are moved into software.

In this article we will do two things:

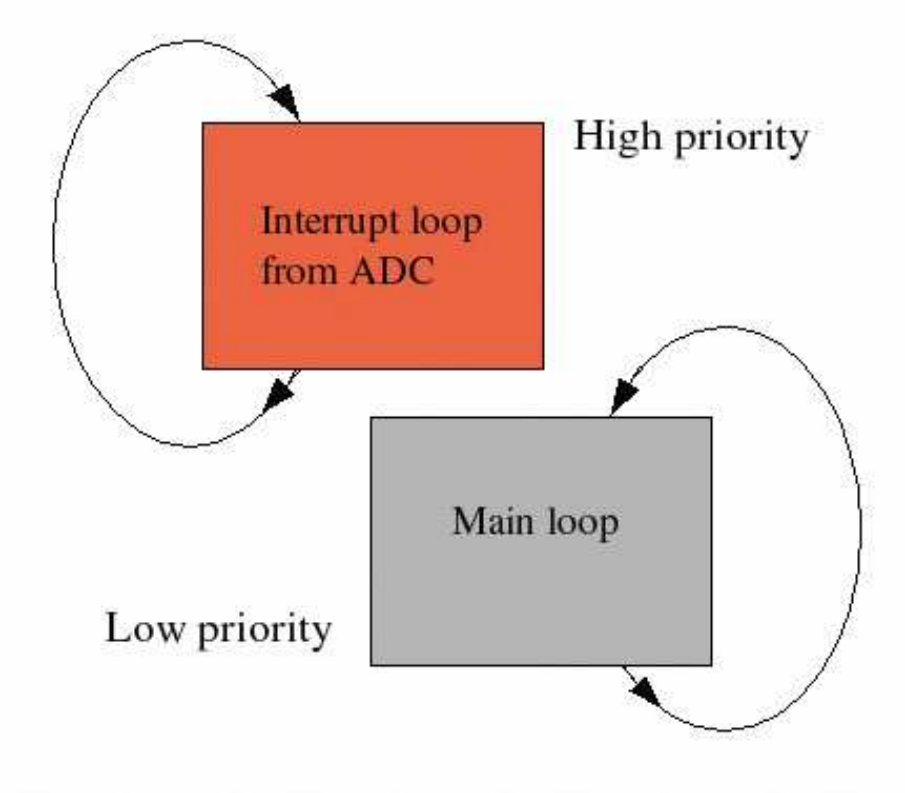
- I will explain how the different parts of the software work.
- Code to store settings permanently will be added.

A word of warning

This article will give you insigths as to how the software works and you can use the knowledge to do modifications. However be aware that the short circuit protection is also only software. If you make a mistake somewhere then this protection may not work. If you then cause a short circuit on the output your hardware will go off in a could of smoke. To avoid this you should use a powerful resistor (e.g bulb from a car front light) which will draw enough current to trigger the protection (e.g 6A) but not enough to destroy the hardware. This way you can test a short circuit without any danger to loose the hardware.

The structure of the software

When you look at the main program (file ddcp.c) you will see that there are only a few lines of initialisation code executed at power on and then the software enters an endless loop.
There are 2 endless loops in this software for the powersupply. One the main loop ("while(1){ ...}" in file ddcp.c) and the other one is the periodic interrupt from the Analog the Digital Converter (function "SIGNAL(SIG_ADC){...}" in file analog.c). During initialiation interrupt is configured to execute every 100µ Sec. All functions and code that is executed runs in the context of one of those tasks (task the name for a process or thread of execution in a real time OS).



The interrupt task can stop the execution of the main loop at any time. It will then execute without beeing interrupted and then execution continues again in the main loop at the place where it was interrupted. This has two consequences:

1. The code in the interrupt must not be too long as it must finish before the next interrupt comes. What counts here are the amount of instructions in machine code. A mathematical formula, which can be written as just one line of C-code may result in hundreds of lines of machine code.
2. Variables that you share between interrupt code and code in the main task may suddenly change in

the middle of execution. This is also valid when you hand more than one byte of data from the interrupt to the main task. To copy two bytes will require more than one instruction and then in can happen that the first byte is copied before the interrupt while the second byte is copied after the interrupt. What to do? In most cases it is not a problem because the measurement results from the ADC will not differ too much between two interrupts. In cases where you can not affort this type of occasional fault (it may happen only once every hour) you have to use a flag which you can check to see if your code was interrupted during the copying.

All this means that complex things like updating of the display, checking of push buttons, conversion of ampere and volt values to internal units etc ... must be done in the main task. In the interrupt we execute only things that are time critcal: Current and voltage control, overload protection and setting of the DAC. To avoid complex mathematics all calculations in the interrupt are done in ADC units. That is the same units that the ADC produces (integer values from 0...1023).

Here is the exact logical flow of operations that we do in the main task:

```
1) Copy the latest ADC results from the interrupt task
2) Convert them into display values (ampere and volt)
3) Convert the wanted ampere and volt values (what the user has set)
   to internal equivalent ADC values
4) Copy the wanted equivalent ADC values to varialbles such that
   the interrupt taks can use them.
5) Clear the LCD display
6) Convert the numbers which we want to display on the LCD into
   strings.
7) Write voltage values to the display.
8) Check if the interrupt taks regulates currently voltage or current
   (current limitation active)
9) If voltage is the limiting factor then write an arrow behind
   voltage on the display
10) Write ampere values to the display
11) Check if the interrupt taks regulates currently voltage or current
    (current limitation active)
12) If current is the limiting factor then write an arrow behind
    current on the display
13) Check if a button was pressed. If not what 100ms and check again.
    If a button was pressed then wait 200ms. This is to have a good
    response of the buttons and not too fast scrolling if they are
    permanently pressed.
14) Go to step 1).
```

The interrupt task is much simpler:

```
1) Copy the results from the ADC to variables
2) Toggle the ADC measurement channel between current and voltage
3) Check if exessive current is measured. If so set the DAC immediatly
   to a low value (It does not have to be zero since the voltage
   amplifier circuit works only from 0.6V on wards (0.6 volt input
   produce still 0 volt output)).
4) Check if voltage or current needs to regulated
5) Check if the DAC (digital to analog converter) needs updating
   according to the decission from 4).
```

This is the basic idea of the software. I will also explain what you find in which files an then you should be able to understand the code (given that you are familiar with C).

Which file contains what

```
ddcp.c -- this file contains the main program. All initialisation is
  done from here.here.
  The main loop is also implemented here.


analog.c -- the analog to digital converter and everything that
  runs in the context of the interrupt task can be found here.

dac.c -- the digital to analog converter. Initalized from ddcp.c but
         used exclusively from analog.c

kbd.c -- the keyboard driver

lcd.c -- the LCD driver. This is a special version which will not need
         the rw pin of the display. It uses instead an internal timer
         which should be long enough for the display to finish its task.
```

New functionallity: store settings

The new functionallity we add in this article is not much since I spent already a major part of this article to explain how the software works and I don’t what to make the article too long.

Still the function we add now is essential: Store the setting such that the voltage and current must not be set again after the next power on. We store those values in the eeprom of the microcontroller. All eeproms (including usb-sticks) have limit as to how often a eeprom storage cell can be written. For the Atmega 8 this is 100000 times. After that the eeprom is waren out and may not keep the values any longer. A trick to get longer life time is to write over several cells but lets first calculate what this means for us. 100000 wite cycles corresponds to storing 10 times a new setting per day for 25 years. This is more than enough. We can therefore just use the simplest solution and store into one eeprom address.

So how do you store/read something to/from the eeprom? There are two instructions eeprom_read_word and eeprom_write_word to read or write 16bit integers into the eeprom. eeprom addresses start from zero and counted on bytes.

One compication is that the eeprom is erased when we upload new software. So we to be able to know if we have read some garbage from the eeprom (because the software was previously flashed) or if we have valid ampre and voltage values in the eeprom. We do this by writing a magic number into the eeprom. In other words we store everytime 3 things: ampere limit, voltage limit, magic number. If we read after power on the eeprom then we check first for the magic number. If it is our number then the values for ampere and volt are used. The magic number can be anything which is not likely to be there by default (e.g 19). To see the exact code look at the function

store_permanent() in ddcp.c (download at the end of this article).

The software for this article is digitaldcpower-0.3.X where X is the revision which I plan to step if there are updates needed. (software for the previous article was (digitaldcpower-0.2.X).

Have fun! ... The next article will add I2C communication to the powersupply from the PC. So you can not only press a button on the powersupply to change something but you can do it via command.

I am looking for people who can port the i2c host programs to different operating system. Let me know if you can help here. You need some knowledge about control of the rs232 interface and a compiler. The actual change affects only one line of code.

The whole circuit with all parts and a printed circuit board is available from shop.tuxgraphics.org (see below).

References/Download

- Download page for this article (updates and corrections will also be available from here).
- How to program the atmega8 with gcc: November2004 article 352
- Tuxgraphics electronics section, a collection of all articles in this series.
- Tuxgraphics online shop, microcontroller section, You can order all parts (transistors, passive components, LCD display, PCB, microcontroller, ...) from here.

<--, tuxgraphics Home | Go to the index of this section

© Guido Socher, tuxgraphics.org

2005-07-17, generated by tuxgrparser version 2.52