Amplifier protection with Arduino

Figure 1: The amplifier

The photo shows my power amplifier with a classic design built into a nice Italian cabinet.

Introduction

I have two ICEpower Class D amplifier modules (type 500A) which I want to drive my bass speakers. These modules are able to deliver 500 Watt of continuous power pr. channel with peaks exceeding 1.600 Watt. The build in protection features are limited and after I blew up one of the amplifier modules, I decided that I needed better protection of the amplifier as well as the speaker.

I decided to go for an Arduino project with soft start and numerous protection features that can shut down the amplifier within milli- or microseconds depending on the fault.

Not the first Arduino project

I started my Arduino journey with smaller projects that introduced me to the Arduino environment. The internet is full of documentation and community support that is highly recommended and which ensures an easy start.

When I started this project, I expected that it would be rather simple taking a month or so. Over time, it grew in size and features and I ran into interesting challenges and opportunities caused by the balanced output. I will try to extract general ideas and recommendations from my project and I will ignore many programming details.

Major components

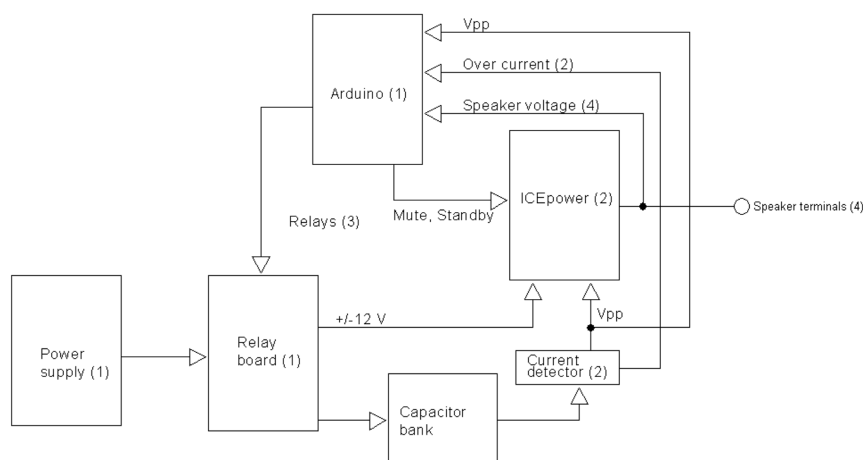

Figure 2: Major components

The figure shows the major components and interconnects with the number of components in parenthesis. This article is only about the protection and not about the amplifier in general and all connections that are not part of the protection scheme are ignored.

© Steen Frank Jensen 2015

The ICEpower is a balanced amplifier with four speaker terminals to feed two speakers. Power (+/- 12V, 80V) is supplied via a relay board with three relays that handle power up and soft start. The current detectors test for over current on the Vpp supply line to the amplifier and deliver a digital signal if the limit is exceeded.

Arduino controls power distribution via the three relays and it controls the ICEpower modules via two control lines, Mute and Standby. Arduino receives input from two digital signals that monitor the current limit and it measures Vpp (80 v) and the voltages on the four speaker terminals.

The capacitor bank for Vpp is positioned after the relay board to support soft start.

My Arduino is a basic version (Arduino Uno, R3) with 6 analog inputs and 14 digital inputs/outputs.

Power state

Power state is a number of steps used during power up. Each step contains one or more activities (like turn on a relay) and it takes a predefined delay to allow a relay to draw or a voltage to stabilize. Power state has steps that support soft turn on and it has steps that verify that voltages are as expected.

In case of a serious fault, I perform power down, which simply goes directly to PowerOff without any delay.

#	State	Delay [mS]	Comment
0	PowerOff	500	All relays off, Mute = high, Standby = high
1	VccOn	500	Turn on rel1 (+/-12 V)
2	VppSoft	1000	Turn on rel2 (80 V via soft start)
3	VppOn	500	Turn on rel3 (80 V)
4	TestVpp	1	Vpp within range (65V - 80V)
5	Enable	350	Enable amplifier via Standby signal (note 1)
6	AutoZero	1	Perform autoZero on speaker terminals
7	DeMute	500	Turn of Mute (active low)
8	ICEOn	NA	Normal operation

Figure 3: Power states and activities

Each state is identified with a number and a name. Most steps are controlled by the delay. Two delays are very short in reality I perform a test and if OK I terminate the state immediately. The comments show the major task for each state. Tests are described in the next section.

Note 1: ICEpower is controlled via two active low control lines called Mute and Standby. I use the words mute, de-mute and enable and disable to describe the action of the two control lines.

Amplifier faults and symptoms

An amplifier can have a number of faults with different symptoms. A shorted output transistor will draw the output low (~0V) or high (Vpp) and with a balanced output it should be possible to detect this fault, because the output is no longer balanced. A fault in the front end including a fault in a preamplifier will properly pull the two output terminals to the speaker in opposite direction in which case it is impossible to tell if it is a fault or a music signal.

The protection scheme

The protection scheme is designed to protect the amplifier as well as the speakers. Playing too loud for too long may damage the speakers and my ears, an accidental short of the output terminals will damage the amplifier or the power supply (I know from experience) and a transistor that fails can destroy the speaker in no time.

© Steen Frank Jensen 2015

Initially I focused on protection during normal operation, later I found that some failures may even be catastrophic during power up therefor I decided to implement tests in all power states. I found that the best tests depended on the actual power state and I even found that balanced output gave some very interesting opportunities.

Tests are based on 2 digital inputs that signal that the current limits are exceeded and 5 analog inputs that measure Vpp and the four 4 speaker terminals every mS.

The amplifier has balanced outputs and the signal on two output terminals driving a single speaker can be expressed as:

```
VO+(t) = DC(t) + music(t)
VO-(t) = DC(t) - music(t)
```

The DC signal on each terminal will be 0 V during power up. When the power amplifier is enabled, the DC signal will shift from 0 V to Vpp/2. The music signal will be 0 V until the amplifier de-mutes. After de-mute the music signals on the terminals will be of opposite phase.

If we calculate the sum and the difference of the two signals, we get:

```
Sum(t) = 2*DC(t)
Difference(t) = 2*music(t)
```

In sum, we see that the music signals cancel and we get a pure DC signal that changes with the power state. In difference, we see that the DC signals cancel and we get the music signal that the speaker sees. Access to sum and difference is possible because of the balanced output and it is an important part of the protection scheme.

Power state		**Tests**					
		Current limit (2)	**Vpp(1)**	**Terminals (4)**	**Sum (2)**	**Difference (2)**	**Power (2)**
0	**PowerOff**	all states		0V			
1	**VccOn**			0V			
2	**VppSoft**			0V			
3	**VppOn**			0V			
4	**TestVpp**		Range test	0V			
5	**Enable**		Range test			0V	
6	**AutoZero**		Range test	Vpp/2+auto zero			
7	**DeMute**		Range test	clip	Vpp	DC offset	power limit
8	**ICEOn**		Range test	clip	Vpp	DC offset	power limit

Figure 4: Tests depending on power state

The table gives an overview of the tests. The numbers in parenthesis illustrate that I have 2 current detectors, 1 Vpp signal, 4 output terminals and 2 speakers for which I calculate sum, difference and power. In the program, this is reflected by arrays and program loops of size 2 and 4. The table also shows expected theoretic voltages in reality I accept deviations of 1-2 V.

Current limit

Over current is the most time critical fault because a short circuit can destroy the amplifier in a few mS. In order to optimize the response time I have implemented an over current detector that generate a digital high when the current limit is exceeded.

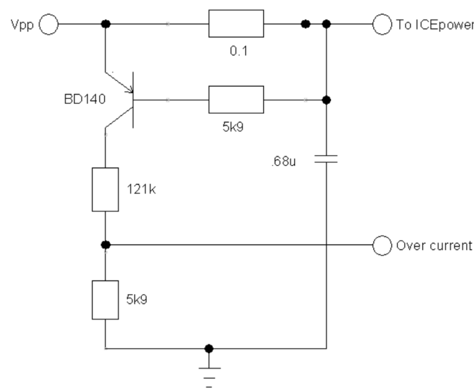

Figure 5: Over current detector

I test for over current in all power states with a response time less than 1 mS. When I detect over current I stop the amplifier via the Standby control signal which blocks the amplifier in just one uS. As a secondary protection, I turn off the relays that send power to the amplifier but this is much slower.

Vpp

The capacitor bank is charged during power up. After Vpp has stabilized, I test that Vpp is within normal operating range (65-80 V). Vpp is unregulated and changes with varying mains voltages and because Vpp is an important reference voltage, I measure Vpp together with the four speaker terminals.

Four terminals

The voltages on the four terminals change with the power state. Until ICEpower is enabled, all terminals should be at 0 V. If an output transistor has failed, we might see the terminal voltage increase as the capacitors are charged and we can detect this fault very early during power up by testing for 0 V. During enable the terminal voltages will swing from 0 V to Vpp/2 which is not useful for individual test on the four terminals, instead I test the difference which should be 0 V during enable. After enable, all terminals should be at Vpp/2 and auto zero collects offset calibration data that are used to optimize subsequent voltage measurements.

Clipping can be caused by harmless peaks in the music. I test that a single clipping has a maximum duration of 10 mS. Longer duration could be a sign of a serious fault. The limit of 10 mS corresponds to a heavily clipped 20 Hz signal. Higher music frequencies will cause shorter clippings that are limited to a maximum of 10% of the time for 1 second and 3% for 1 minute. All clipping tests are slow because duration is part of the test.

Sum

As previously mentioned the sum of the two speaker terminals will cancel the music signal and deliver Vpp. This is tested during the last two power states. Deviations from Vpp is a strong sign of a serious fault and will cause immediate shut down.

Difference

Taking the difference between two terminals deliver the speaker signal. During enable, the DC signal on the two terminals will shift from 0 V to Vpp within 250 mS and if all is normal then the speaker will be silent and the two signals will follow closely and the difference will measure close to 0 V on all samples.

The last test is the traditional DC offset measured over the speaker and seen on many amplifiers. It is designed to detect DC down to 1 V without being triggered by a full power 20 Hz music signal. The difference signal is send to a digital 1 Hz second order low pass filter that attenuates the music signal. The filter is implemented in software as an “Infinite Impulse response” filter (IIR).

Power

My bass speakers are rated at 150 W continuous power (IEC 17.1). I have defined limits of 200 W for 1 second, 50 W for 10 seconds and 20 W for 1 minute. These limits are conservative and may be changed later.

Power and energy: The power is measured in Watt and is calculated every mS. In this calculation (p=v*v/r) I treat the speaker as an ideal 4 Ohm resistor which is a simplification of the real impedance. Each power sample (p) is added to variables that sum power over time, which is the same as energy. Energy is often measured as kWh, in this application I measure energy in Watt*mS. The energy is collected in three variables called energyPrSec, energyPr10Sec and energyPrMin. To be scientifically correct I must admit that the power limits are implemented as energy limits. A maximum power of 20 Watt in one minute is the same as a maximum energy of 20 Watt*60,000 mS = 1,200,000 Watt*mS. Every mS I test for maximum energy and every minute I clear the energy counter to start a new measurement period.

Fault handling

Faults are considered critical or less critical. Critical faults result in power down less critical faults result in mute for 30 seconds. Exceeding power limits are considered less critical most other faults are critical.

In order to protect the amplifier and the speaker it is vital to respond as quickly as possible. The response time consists of two parts: The time it takes to detect a fault and the time, it takes to close down. The detection time depends on the type of fault. Over current is a digital signal that is handled with a response time less than 1 mS, often a few uS. All other fault detection is based on analog inputs that are sampled every mS and in some cases, it takes several mS to detect a fault because duration is part of the detection. As an example, I consider a clipping that lasts more than 10 mS as a fault.

Close down time: Arduino responds to faults by sending signals to the two ICEpower control lines (Standby and Mute) and by turning of the power supply relays. The Standby signal is able to stop all activities in the amplifier module in just one uS. This quick response is essential for handling critical faults. As a secondary and much slower protection, I turn off the power supply relays. Less critical faults are handled via the Mute signal that turns the music on and off within 250 mS.

LEDs

A blue LED visualizes the power state with increasing light intensity and faults are visualized with blinks on the red LED.

Precision measurements

I use a number of tricks In order to obtain noise free and precise analog measurements.

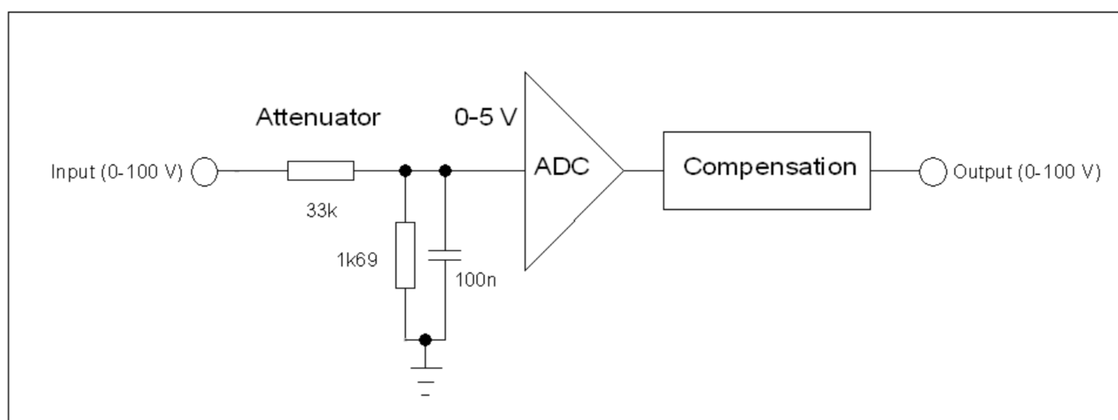

Figure 6Analog measurement

I have designed the input to handle voltages from 0-100 V. This is attenuated 1:20 and the resulting 0-5 V is send to the build in analog to digital converter (ADC). It has a resolution of 1024 steps, which means that I can measure 0-100 V with a resolution of 0.1 V. On paper, these data are more than adequate but some issues had to be solved.

Resistor tolerance. With 1% resistors, each attenuator (two resistors) will have 2% tolerance in gain and when you measure DC over the speaker, 4 resistors are involved giving a worst case error of 4% or 4 V. This must be improved. Trim pots would be the classic solution but I implemented **gain calibration** as an array of constants which I "trimmed" to a precision of 0.5% of full scale (FS).

Noise. High frequency noise from the class D amplifier was the next issue. A first order low pass filter with an fc of 1,000 Hz removes this. This is the capacitor in the attenuator.

High CMRR. I need high precision measurement of the voltage across the speaker. In a classical electronic circuit, I would implement a differential amplifier with high CMRR. To reach the same result in the digital world I needed to optimize the precision of the voltage measurements. I have already mentioned what I call gain calibration to within 0.5% FS. In order to improve even further, I have implemented an auto zero state that is executed when the speaker terminals have reached their normal operating voltage of Vpp/2 and before de-mute. The result is a small offset voltage that improves precision of the difference signal during normal operation.

All of the tricks come into action in the program line where I read the analog input from the speaker terminal and convert it to a floating point voltage:

```
currentV[i] = analogRead(i)*vFactor*GainCal[i]+offsetCal[i];
```

[i] is an index that loops through the output terminals. currentV[i] is the voltage of speaker terminal [i] after calibration. analogRead() returns the ADC result as an integer 0-1023 this is multiplied with vFactor giving the uncalibrated voltage. The uncalibrated voltage is multiplied with GainCal[i] the fixed calibrations that I have entered and finally offsetCal[i] from auto zero is added.

With these tricks, I have obtained very precise and stable measurements. When I turn down the music signal I read 0.00 V over the speakers.

About programming

So far, I have focused on electronic issues. In the rest of this article, I will focus on the programming activities and the experience I have gained.

Real time clock

I have implemented a real time clock that maintains the current time (day, hour, minute, second and millisecond) since the program started running. This is used as a timestamp for all events and is an important tool to verify correct program execution.

```
d:0 h:0 m:0 s:2 ms:501 setpowerState
  newPS=5 Delay=350

d:0 h:0 m:0 s:2 ms:851 setpowerState
  newPS=6 Delay=1

d:0 h:0 m:0 s:2 ms:851 testLines50%
  currentHTV4[4] = (40.22, 40.05, 40.12, 40.05)

d:0 h:0 m:0 s:2 ms:851 autoZero
  offsetCal[4] = (0.45, 0.62, 0.55, 0.62)

d:0 h:0 m:0 s:2 ms:852 setpowerState
  newPS=7 Delay=500

d:0 h:0 m:0 s:3 ms:352 setpowerState
  newPS=8 Delay=0

d:0 h:0 m:0 s:28 ms:103 setError
  eGroup=4 eCode=3 lineIndex=1
  ALARM - Max 20 W 1 min. E=1200.03 WattS, Timer=25.25sec, AvgP=47.52W
    => Mute 30 seconds
  mute=1

d:0 h:0 m:0 s:58 ms:0 setError
  eGroup=0, eCode=0, lineIndex=0
    => Error cleared
  mute=0
```

Figure 7: The use of timestamps

This example illustrates timing of some major events. We see the timing of the last power states (newPS=5, 6, 7, 8) during power up. At 2.851 second we see the test of the four speaker terminals for Vpp/2 and within same time slot, we also see auto zero including the offset calibration data.

At 2.852 second we reach power state 7 (de-mute) and I start power calculation. From this time, my signal generator sends a sinus of approximately 50 Watt and at 28.103 second this triggers the limit of max 20 Watt for one minute. The power measurement lasted 25,251 milliseconds (28,103-2,852) and the power was 47.52 Watt (and not 50 Watt) giving a total energy of 47.52 Watt*25,251 mS ~ 1,200,000 Watt*milliseconds (20 Watt*60,000 mS). This calculation illustrates how precise time stamps are used to verify the functionality.

30 seconds later, I clear the fault and de-mute.

DSP engine

*“**Digital signal processing** (**DSP**) is the numerical manipulation of signals, usually with the intention to measure, filter, produce or compress continuous analog signals. It is characterized by the use of digital signals to represent these signals as discrete time, discrete frequency, or other discrete domain signals in the form of a sequence of numbers or symbols to permit the digital processing of these signals.”*

Definition from Wikipedia

In line with this definition, I have implemented a DSP engine that consists of a set of timers (1mS, 10 mS, 100mS, 1 S, 10 S, 1 Min) that makes it easy to perform signal processing at discrete time intervals.

All analog signals are sampled every millisecond. Most calculations and limit tests are performed every millisecond and some of the slower timers are used to reset measurements at the end of a measurement period like maximum power for one minute.

This is supplemented by an idle loop that contains activities that can be performed when the CPU is not busy with other tasks. Because the idle loop is executed very frequently, I use it for high priority activities like testing for over current.

With a DSP engine, you can take advantage of digital filters. Infinite Impulse Response filters (IIR) are simple and efficient filters that I have used to implement second order low pass filtering to detect DC signals across the speakers.

Simulation of electronics

With a DSP engine you may even simulate an electronic circuit. In this way I have implemented a de-bounce circuit consisting of a first order low pass filter followed by a Schmitt-trigger.

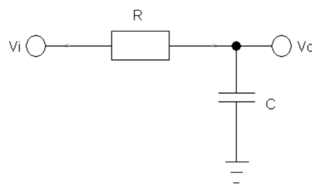

Figure 8: Low pass filter

I calculate the output voltage Vo with just one program statement:

```
vo = vo + (vi-vo)*k;
```

This statement is executed every mS. It calculates the new Vo as a sum of the previous Vo plus an increment that is proportional to the voltage across the resistor (Vi-Vo). k is a filter coefficient that can be calculated as `k = 2*Pi*Fc/Sr` where Fc is the cut off frequency and Sr is the sample rate (1000). This solution is simple and efficient. If you implement a first order low pass filter as an IIR-filter, you will get the same result.

The output from the filter (Vo) is followed by a Schmitt trigger that calculate the digital output (trigger) in this way:

```
if (vo > 0.9) trigger = true;       // 0.9 V high trigger level
if (vo < 0.1 ) trigger = false;     // 0.1 V low trigger level
```

I find that simulation of electronic circuits can be an attractive supplement to more programmatic solution.

DSP and reliability

It is said that all software contains two kinds of errors: Those errors that we know and those errors that we do not know, but I have found this Arduino program to be extremely reliable and free of software errors. Why is that?

In traditional programming, most programming statements are only executed under certain circumstances. These circumstances may be very rare and it may take very long time before they occur and when they finally occur we are not prepared to test if they execute as expected.

In DSP programming, most program statements are executed very often, like every millisecond. If I make a mistake in a formula that is executed every millisecond I can see the effect immediately.

I think that the DSP-programming style shares some attractive properties with traditional electronic circuits:

When I test an electronic circuit, I use an oscilloscope and a probe to see how signals vary with time. I do the same when I watch how digital values change over time. When I test an electronic circuit, my probe needs access to all internal pins in order to improve test. This is also true with software. When I design an electronic circuit, I prefer simple circuits where the output comes from a single source that may feed many inputs. In traditional software, you may see variables that are set from many different places, which complicate matters. When I design an electronic circuit, I avoid multiplexed buses because they are inherently complex. Most traditional software reminds me of multiplexed buses.

The conclusion is that DSP as a programming style is a good way to obtain a reliable solution; all you need is easy probing. In software, easy probing can be translated into test routines that are optimized to watch time varying signals and I have developed numerous test- and print routines for this purpose.

Extensive tests

Software for protection requires extreme focus on testing. Under normal operating conditions there will be no response from the software, it will just be watching what is going on. When something goes really wrong, it must react in milliseconds to prevent catastrophic errors, at the same time it must not be over sensitive and close down unless absolutely necessary.

Mosts tests will stress the amplifier beyond safe operating conditions and if the software does not react as expected your amplifier may be damaged.

As a consequence you need an extreme focus on tests and your software must be very reliable before you start to overload your amplifier.

To meet these requirements I combine a systematic workflow with unit tests and integration tests.

I work with iterations that takes 2-5 days and consist of 20-30 changes. Each iteration starts with short descriptions of the changes. After all changes are implemented, I make intensive tests of all changes and I update my documentation.

In one of the first iterations I created the main program structure (every1mS, every1S etc.) and the real time clock. Power up and power state took several iterations and handling of faults was a major task.

Most unit tests are performed with stubs that simulate real input. Instead of scanning the four terminals, I write small pieces of code that simulate real input. This can be a DC signal, a triangle or a sinus with a well-defined value and start time. The test will show how my controller reacts. If I generate a DC signal how long will it take for DC Offset to react? If I generate a high power sinus how long will it take before the power limit is detected?

As most tests collect data in arrays (example: energyPrSec[2]) it is important to follow how data changes over time so I have developed printing functions for the arrays. All prints start with the real time clock followed by the array name and its contents. Most of this code is temporary for debug but major events like changes to power state and detection of faults are printed as part of the final solution.

Integration test

Integration tests are tests where every thing is put together in the final solution. Because I tested the program as much as possible before integration test I have only had minor problems where the most important was that 500 kHz switching resulted in noisy measurements.

Running out of resources

During development, I ran out of resources on the Arduino.

Memory

My Arduino has 2.048 bytes of memory for variables, stack etc. The compiler reports how much memory is used for variables and when it reached 1.800 bytes I had problems. The symptoms were missing characters during print. Function calls and parameters use dynamic memory and in my case, I used the last bytes to print long strings. I removed some features and variables and made shorter strings for print.

CPU power

As I added more and more tests, I ran out of CPU-power. I found that most of the time was spent on AD-conversion (ADC) which is a very slow process with standard Arduino function *analogRead()*. After some study on the net, I implemented what is called interrupt driven ADC. In praxis, this means that I am able to use the CPU for my calculation at the same time as an AD conversion takes place. This solved all problems with CPU power.

Analog inputs

I use 5 out of 6 analog inputs. If I had more analog inputs on the Arduino board, I would have measured speaker current, which would give a more precise power measurement. But even with more analog inputs I would still keep the implemented over current detector because it gives a faster response than sampling analog input.

The last programming comments

Keep it simple

I try to follow the general recommendations in Arduino style guide and other guides, see resources. I try to keep programs well documented and as simple as possible and I avoid advanced constructions including pointers. I also follow a strict standard for naming of variables and constants. If you search the net for CamelCase you will find a number of variations, in my variant I start constants with uppercase and variables and functions with lowercase.

enum, case and switch

enum, case and switch are useful programming features that you may not discover in the beginning. They improve on flexibility and readability and are highly recommended. Search the net for more information.

What would I change?

During development, I have been through many iterations with continuous improvements and now I have a robust and reliable solution that I can happily live with.

If I should start all over I would choose a more powerful Arduino and if I had unlimited time it could be fun to find ways to measure or calculate the voice coil temperature but other projects are waiting.

© Steen Frank Jensen 2015

Resources

1. ICEpower 500A data sheet.
   http://www.icepower.bang-olufsen.com/files/solutions/icepower500adata.pdf

2. Arduino homepage.
   https://www.arduino.cc/

3. Arduino style guide.
   https://www.arduino.cc/en/Reference/StyleGuide

4. Style guide for variable names.
   https://da.wikipedia.org/wiki/CamelCase

5. Digital signal processing
   https://en.wikipedia.org/wiki/Digital_signal_processing

6. Nice cabinets from Italy, see Galaxy.
   http://www.modushop.biz/site/index.php?route=common/home

7. My homepage
   steenfrankjensen.com

© Steen Frank Jensen 2015