Technical University of Denmark

Anders Bahnsen
Aleksander Gosk
Kim Rostgaard Christensen

31070

Hands-on mikrocontroller programmering

3 week project report, January 2010

DTU Electrical Engineering
Department of Electrical Engineering

Contents

Chapter 1

Introduction (Anders)

In electrical power production there has to be a balance between production an consumption meaning that production must equal consumption at all times. This is a problem because consumer demand and thereby the load on the electrical grid is unpredictable. Luckily the electric power generators represents a rotating mass which means that energy can be stored as kinetic energy governed by the rotational speed.
In this system a mismatch between production and consumption will influence the frequency of rotation of the system in such a way that an over consumption will lower the frequency and in worst case cause a black out, overproduction would cause the frequency to get higher.
To prevent this, research is done to make equipment intelligent in such a way that they react on the grid frequency by turning off loads or even putting power back in to the system, the last option is an ideal opportunity in a society with a large fleet of electric battery powered vehicles.
In the event of overproduction loads can be turned on in order keep the frequency at the right level, until production can be regulated down. A large fleet of electric battery powered vehicles would also be a way of saving energy for later use.

This course in hands-on micro controller programming concerns the use of a micro controllers applied as a part of an intelligent energy system. The micro controller is used to measure the grid frequency and control loads connected to the power grid. In this way we have a frequency responsive system that can adapt its consumption based on the frequency of the power grid.

Chapter 2

Requirements analysis and specification (Anders, Kim)

The ELSAM agreement states that the grid frequency must be 50Hz at all times . If the frequency is lower, our device should turn off unneeded loads or devices. ELSAM defines two types of loads; Normal operation reserve and Disturbance reserve.
The Normal operation reserve should be completely activated when the frequency drops 49.9Hz. Under these conditions it should turn off equipment, which is able to do so in 2-3 minutes or less. These devices would include equipment not crucial to production (e.g.heater or lighting).
Disturbance reserve mode is used when the grid frequency does not return to normal state. In our case we turn off a battery powered device (a laptop) and try to make sure we don't empty their batteries completely before turning the power back on.

To make the ELSAM demands more accessible and more applicable to our assignment we have put the demands in to a hierarchy see Figure 2.1 .
This makes it easy to see which requirements will be applied to the normal operation reserve and to the disturbance reserve. Many of the demands are not implemented yet, this means that the they should also be viewed as suggestion to further work.

To be able to do more "intelligent" grid offloading, we must be able to communicate with the outside world. As there is no protocol specified, the logical choice would be tcp/ip, as there is is already a well-established global infrastructure using this. The data should be transferred in an easy parsable format, such as XML.

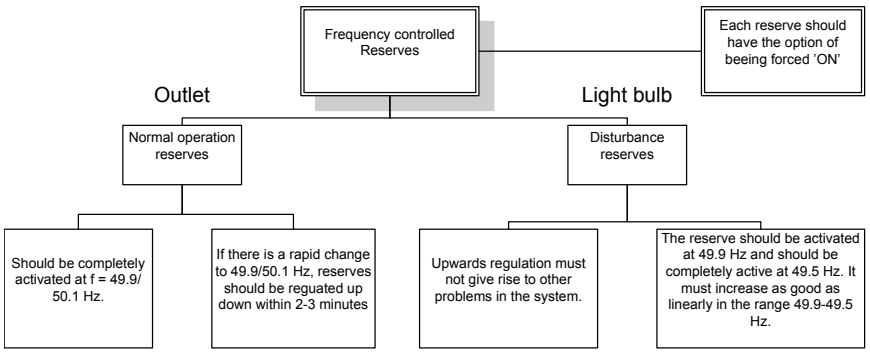


Figure 2.1: Hierarchy of the frequency controlled reserves in this report

Users of the device should also be able to view the status, current configuration and make changes to the configuration as well. The status and configuration should be available both from a physical interface and via a remote interface. As our device has a touchscreen it will serve the purpose as a physical interface, and the remote interface should be done in html.

This leaves us with the following requirements:

- Detect grid frequency, and respond to changes
- Toggle relays based on frequency algorithm
- Implement a TCP/IP stack
- Implement a HTTP server
- Design human interfaces for local access
- Design human interfaces for remote access
- Design machine interfaces for remote access

2.1 Tools used (Kim)

An application development process' success or failure can depend largely on the tools used in the process. Good tools for documenting, debugging and versioning should be considered bare minimums.

2.1.1 Debugging tools and methods (Kim)

When developing the communications interface we use the program "wireshark" [1] to verify http requests and responses.
The board we are using contains a JTAG port, for interactive debugging. This means we are able to stop the processor, read and modify registers or memory. This feature is extremely useful when debugging an application.

2.1.2 Information scraping (Kim)

For doing quick notes on random information concerning code, documentation, requirements or specifications. We will be using a wiki for this purpose

2.1.3 Versioning (Kim)

When developing an application, a versioning system is very useful both for experimenting with new features, because you have the ability to quickly roll-back to a working version. It is also a great way to do backup on your project.

2.1.4 Thoughts on user interface (Kim)

Initally we had some thoughts on what the purpose of the user interface should be, and which functions should be attached to it.

To indicate that the system is running we can use a blinking led to indicate a heartbeat.
On system error we could do a fast blinking led or the buzzer.
The Ethernet status and uptime could also be a nice feature.

The application should be able to display relay status, real time frequency (mean), message label, error label, RMS values and power.

The application should have different screens showing settings, status and full status.
It should also be possible to adjust frequency range and manually override the settings on the relays.

Other features could be to turn the back light of the LCD off to save power.

[1] www.wireshark.org

Chapter 3

Description of program

3.1 Program structure (Kim)

The developed application is divided into three modules as seen in figure 3.1

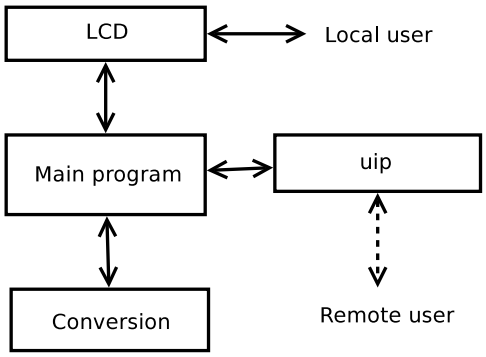


Figure 3.1: The overall program structure

The conversion part is responsible for most of the ADC, GPIO and interrupt handling parts of the program. The LCD is taking care of the physical user interface, both touchscreen and displaying values on the screen. The uip is accepting remote connections and serving data to remote clients.

The program roughly runs the sequence depicted in figure 3.2

The first thing done is the initialization. This obviously only run once. It configures the interrupts and configures the appropriate GPIO pins. When

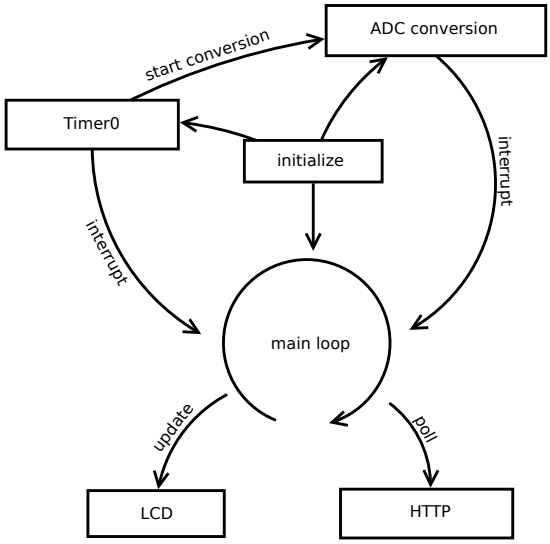


Figure 3.2: The overall program sequence

done, it enters the main loop, where the application remains for remainder of run-time, while not processing interrupts.
The Timer0 runs at a predefined frequency, and takes care of starting the ADC conversion process, reading the touchscreen coordinates, detect zero-crossings, do http timer ticks and controlling relays.

The ADC interrupt handler function detects touches to the screen and stores the data from the measurements, does filtering and accumulates the squared values (integrates), and is used to calculate the corresponding RMS values.

When not doing interrupts the main loop updates the screen, checks if a user has touched the screen and moves the cursor accordingly.
It also does periodic checks on the IP stack, sending and receiving packets at a reasonably stable frequency. This is done via the HTTP server.

3.1.0.1 Debugging the timing

Throughout our development process we had to verify that no single interrupt took so long time to execute that the next interrupt was delayed.

We thought of three possible ways to ensure this.
One method is to start another timer at the start of the interrupt routine, and stop it again at the end. We would now have the time the interrupt

routine took.
Another method would be to clear the timer interrupt flag at the start of the interrupt routine and check the register at end of the routine. If it is high again - then our interrupt routine takes too long.
The third method is to output a logical high signal to the GPIO at the start of the interrupt routine, and then set it low again at the end of the routine. We've used the third method.

3.2 Measurements and control (Aleksander, Anders)

3.2.1 GPIO

We are using a number of GPIO's for various purposes. For example we are using P[1]_18 and P[1]_13 for LED's, P[0]_11 and P[0]_19 for controlling the relays. Other GPIO's are used in the touchscreen handling.

3.2.1.1 ADC

To get the maximum number of samples per second, we use burst mode (illustrated in figure 3.3). As this enables us to free up some cpu time and make the conversion faster. The burst is started by entering the Timer0

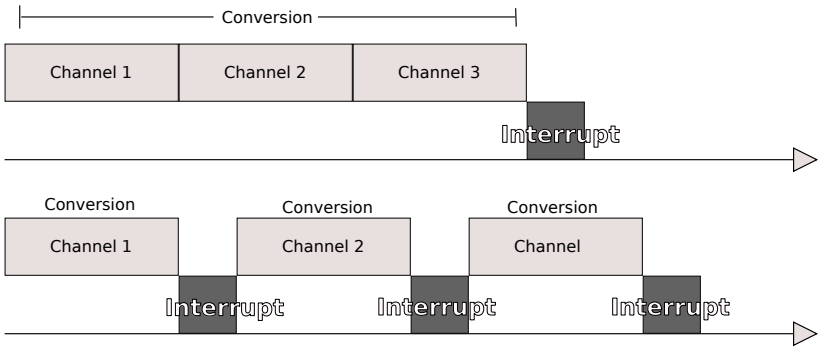


Figure 3.3: Illustration of the principle behind burst and normal mode

interrupt handler, and runs simultaneously with our main loop and Timer0 handler. This is of course due to the fact that the conversion is done in hardware and does not need the processor for calculations.

3.2.2 Measurement algorithms and their implementation

The values measured with the micro-controller (ADC) are:

- Grid voltage
- Grid current
- LCD voltage (checking if, and if so, then where LCD screen has been touched)

3.2.2.1 Voltage and Current measurements

The voltage from the grid is passed through a circuit that lowers it's value to 0...3.3V range, before it can be meassured by the micro-controller. The same thing is done with the current - it is converted to a voltage signal of the same scope. Both signals are scaled afterwards so the output of calculations is in grid values. MATLAB files containing simulated meassurements and calculations on frequency and RMS values are included in the working directory(.zip file) together with the c code.

Voltage scaling First the RMS and offset values of the grid voltage has been measured. The results were:

$$V_{RMS} = 236V$$
$$V_{Offset} \approx 0V$$

then assuming that the value corresponding to 0 V is exactly half of the measurement range of the ADC we set this to 512. After subtracting this value from the raw data we calculate the raw data's RMS value. This value together with the measured RMS voltage can be used to scale the raw data to physical values in volts. The raw data RMS value is 341 which means that the scaling factor for the RMS voltage is

$$V_{scalefactor} = 236V/341 = 0,692082$$

Current scaling A similar approach have been used while calculating the scaling ratio in the case of the grid's current. Here we've assumed that the current converting circuit is very precise and that the output values are exactly 0....3.3V. Thus we can expect that the quantization level corresponding to 0A is equal to 512. Because we can not really measure the actual current

in the system we assume that the system uses the full measuring range, this means that the RMS current signal in raw data values should be

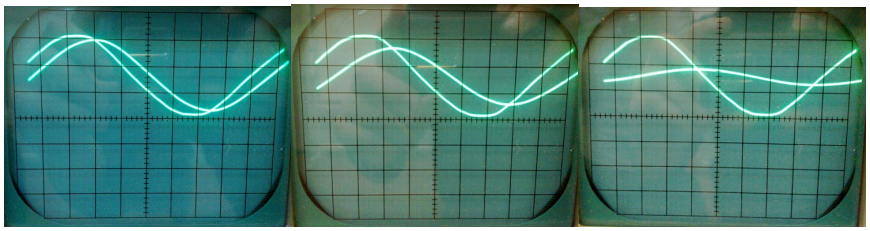
$$Iraw_{RMS} = 512/\sqrt{(2)} = 362$$

The specified maximum RMS current is 1A which gives us a current scaling factor of

$$V_scalefactor = 1/362 = 0,00276243$$

3.2.2.2 Frequency measurement

The frequency measurement is done by calculating the length of the period of a given signal. It is done by calculating the number of samples between the points at which the signal is crossing an arbitrarily defined zero line with an assumed direction (rising edge or falling edge) and comparing it (number of samples) with the sampling frequency.
In order to detect zero-crossing two succeeding samples have to be remembered by the system. If the previous sample was below the zero-level and the current one is above, than the system marks this time as a rising edge zero-crossing and starts to count the samples until the next one occures.
Due to significant disturbances a low pass filter has to be used, in order to smoothen out the data. Without it detecting false zero-crossings would be very common, which in turn would render the method completely useless.
The low-pass filter is used only for this particular purpose, though. All the other calculations, like voltage RMS value, power etc. are performed on raw data. That is due to the fact that the filters output differ significantly from its input (the amplitude is attenuated and the signal is shifted in phase). On the other hand filters, never change frequency, which make them useful in this particular case. This is shown on the Figure 3.4



(a) Cut-off at 100Hz **(b)** Cut-off at 50Hz **(c)** Cut-off at 10Hz

Figure 3.4: Input and output signals of a low-pass filter at a sampling frequency of 10kHz for three different cut-off frequencies.

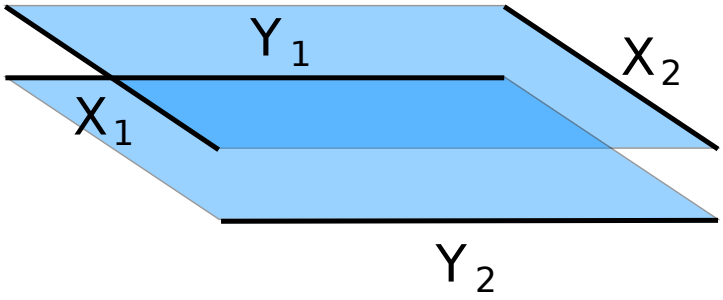


Figure 3.5: Simplified overview of a two layer LCD electrical setup.

3.2.2.3 LCD measurements

The LCD measurements are conducted to check if the touch screen has been touched and if yes, where exactly did it happen. The possition of the touch is determined by measuring the voltage on the LCD's outputs. Measurements of the X and Y coordinates aren't done simultanously. If we want to measure the X coordinate the ADC and the pins have to be set to a certain state. In order to meassure the Y coordinate those set up options have to be different. This is due to the way that the measurements are conducted. The electrical setup of the LCD screen is depicted on Figure 3.5.

In order to measure the X coordinate of the touched place on the LCD screen a potential gradient has to be applied to the bars represented as X1 and X2 on Figure 3.5. The actual X coordinate is determined by measuring the voltage, on the Y layer, wich changes with the distance from one of the X bars, at witch the touch accured. Applying a voltage gradient in an opposite direction would cause the measurement to be giving position references with regards to the other side of LCD (X coordinates would measured from right to left instead of doing it in the left-to-right fashion).
Measuring Y coordinates requires seting up ADC and GPIO pins in reverse manner. Since every LCD output channel is hardwired to a differenht port and we need to change the GPIO configuration each time we want to reset ADC for measuring different coordinate, every coordinate is meassured only every second ADC action. The overview of connections needed for every ADC conversion cyckle is shown in the below table:

Table 3.1: Simplified overview of a two layer LCD electrical setup.

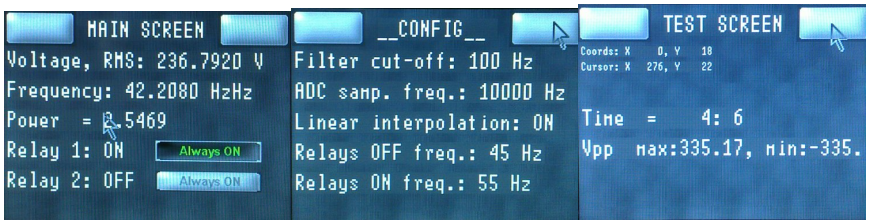
	X normal	X reverse	Y normal	Y reverse
X_1 (pin 24)	set to high	set to low	ADC input	ADC input
X_2 (pin 22)	set to low	set to high	disabled	disabled
Y_1 (pin 23)	ADC input	ADC input	set to high	set to low
Y_2 (pin 21)	disabled	disabled	set to low	set to high

3.3 User interface (Aleksander)

The user interface's layout is presented ont the Figure 3.6.

It is build out of three different screens:

- Main screen
- Config screen
- Test screen



(a) Main screen **(b)** Config screen **(c)** Test screen

Figure 3.6: Screenshots of the graphical user interface parts

There is a panel's name label at the top of each screen. There are also two buttons on it's both sides. They allow the user to switch between screens in a ring fashion (if one button is pressed continuously, then the screens repeat their selves).

3.3.1 Main screen

The main screen is the default panel presented on the LCD. It shows the values of:

- $Voltage_{RMS}$
- Frequency
- Power
- State of relay 1
- State of relay 2

Apart from the above there are two bistable buttons, next to the information on the relay status. If they are on the system forces the relays to be switched on at all times. If not, the relays will be automatically switched off if the grid frequency will drop below a certain level.

3.3.2 Config screen

This panel may depict some of the information on the configuration of the system:

- The cut-off frequency of the low-pass filter
- Sampling frequency
- Usages of the linear interpolation function
- Relay switch off frequency
- Relay switch on frequency

3.3.3 Test screen

This panel may be used during the updating/debugging of the programme for displaying certain variables. Thanks to a special function (*enable_line_of_text()*) only two simple lines of code are required to display a given text line in a specified position on the screen, which makes checking how certain variables change in the real time, fairly easy.

3.4 Communication (Kim)

The communication part of the program is done by reusing the uip network stack. The built-in http server has the ability to both serve files and do some basic server-side scripting.
We plan on refactoring the server, making it able to serve xml and xsl[1] (with the right content-type). Xsl will enable us to generate a single xml file and then transform it for the user to view via a browser. The xml document will be treated as a regular html page, making the transformation transparent for the user.
The really nice thing about xsl is that we only need one source of information for both the user interface (webpage) and for automatically downloading done by programs interested in only the data. The programs will disregard the xml stylesheet and only fetch the data, minimizing overhead.

[1] XML Stylesheet

3.4.1 Implementation

The first step in getting the webserver working as we intended was getting the stand-alone server accepting xml files. This was relatively easy, and so was getting the server to generate content dynamically. This was mostly done by copy-pasting, and the code is included.
One thing that did baffle us though, was the fact that the uIP http server strings was written as arrays of bytes. Very unreadable and pointless, as a string or a char array would been just as good.

Our next step was getting the generic server merged into our main project. The big challenge was that the uip server used a global timer, and not just any timer - the same timer we already used i our main project.
The migration was again done in two steps. First step was moving the httpd timer from timer0 to timer1, and check if that worked. Next step was merging the code of the two timers into one, and get the timing right. The change from timer0 to timer1 resulted in a very slow, but working, webserver.

The problem in merging the two timers was due to the fact that they ran at two different frequencies. One at 20MHz and another at 100Hz. As these are both defined in our config.h as respectively TIMER0_TICK_PER_SEC and HTTPD_TICK_PER_SEC we figured we needed to add a counter and do the following:

$$counter\%\frac{TIMER0\_TICK\_PER\_SEC}{HTTPD\_TICK\_PER\_SEC} = 0 \tag{3.1}$$

The % is the modulus operation. Upon true, the counter should reset and the http tick should increment. This implementation should make the http server insensible to changes in the timer0 frequency.

The webpage itself refreshes every 5 seconds by asking the browser to do a delayed redirect to the same page.

While studying our code more closely, we discovered our slowdown was caused by the LCD doing refresh every free cputime. After getting the refresh down to 5Hz the server performance improved significantly. Later on in the coding process the webserver suffered another slowdown, which we have not been able to fix yet.

3.4.1.1 Historical values

Although not implemented, due to different focus, we have spent some time considering a possible implementation.

At first glance the buffer could be implemented using a linked list - but due to the fact that we do not have at memory manager we would at some point run out of free memory to store these these historical values. So, we could use a fixed size array of int pointers, and a pointer to the last element inserted. Then it should be possible to run through the array "array-size" number of times, starting from "last_inserted" and break if we encounter a null pointer.
The historical values should be displayed on a graph generated as an svg image, using JQuery to reload the values without reloading the page.

3.5 Chapter Summary (Kim)

3.5.1 Communication

The finished result can be viewed by accessing the ip (typically 192.168.0.100) and requesting the data.xml document. When doing this from a browser the transformation works like a charm and gives us a graphical representation of our values. The server is still slow when we enable the LCD and touch-screen, but this should just a matter of optimizing the code that handles these.
The browser refresh is, agreed, a bit quick n' dirty, but also works as intended. The xsl and xml files are with the other files in the uip/http-fs folder.
For communication via TCP/IP on an intelligent power grid, we do not think that xml+http should be the way to go. The overhead would be too large in comparison to a binary protocol.

Chapter 4

Conclusion (Aleksander, Anders)

In the beginning of the project we spent a great deal of time cleaning and refactoring code, making it more modular and easy to navigate. But as the application evolved and grew in size and complexity, it became increasingly difficult to maintain a consistency in the code. Especially when things were not going as planned. New code arose in places where it did not belong.
From this we have learned about the importance of defining a clear program structure and functionality from the start of the project, and enforcing it throughout the development process.
We have also learned that getting aqquainted with the users manual in the first place is a hard task. Still, with the help of the provided example codes it makes the proper configuration of the device much simplier.

The webserver is responding slowly, but is very functional. The xsl transformations works like a charm in modern browsers and should definitely be considered in the case of an official http+xml implementation.

The LCD screen, acting as both in- and output, gives a quick way of getting an overview of the current status of the system. As well as modifying the system parameters real-time.
The values displayed are a bit off, due to programming errors. But it gives you an idea on how a finished system should look and behave.

Measurements done with the use of Analog-Digital Converters need to be

planed quite precisely due to the limitation of the ADC. If the sampling rate is to high compared to ADC's processing speed than we would encounter interferences between ADC and system timer (*Timer 0*) interupts. This would render our measurements completly useless, due to the fact that the sampling period wouldn't be constant. Similiar situation would occure if the interupts handling functions would take too long to execute (would be overfilled with the instructions).
To prevent this errors from happening we've measured the duration of the interupts with the use of ossciloscope. The duration of the interupt was maesured by setting a GPIO pin to 1 at the beginning of an interupt and setting it to 0 at it's end. From these measurements we've concluded that our system would safely perform its tasks with the sampling frequency set to 10kHz.

Without linear interpolation the maximum precision of the frequency measurement would be:

$$SAMPLES\ PER\ PERIOD = \frac{frequency_{sampling}}{frequency_{grid}} = \frac{10000Hz}{50Hz} = 200$$

$$FREQUENCY\ RESOLUTION = \frac{frequency_{grid}}{SAMPLES\ PER\ PERIOD} = 0.25Hz$$

With the interpolation we are able to increase the precison of the frequency calculations by approximately a factor of 10.