AU-Herning : EPRO 4
Renewable Energy System

Project - Energy Hub

Counselor: Morten Opprud Jakobsen & Klaus Kolle

BY E10 - Team3:

Theis Christensen (10691)

Dennis Madsen (90248)

Paulo Fontes (10484)

Table of Content

Chapter 1

Realisation Phase

The realisation phase, is where all the analysis, design, implementation, verification and deployments is made. This phase is divided into several timeboxes, where small parts of the development is done in steps. A timebox is a 2 week time period where parts of the product is made ready for deployments for the end product.

The work that shall be done to complete the project is stated below:

System setup / Design:
CO/Module-design. Deadline in week 10
Relay server. Deadline tuesday week 9
Daemon. Deadline week 11
Almost complete system. Deadline week 13
Complete system. Deadline week 15

Hardware:
ARM7 / FPGA connector
- Schematic
- Print layout
- Mounting

PLC module + 5v supply (Dennis responsible with Kristian). Deadline week 9
- Schematic
- Print layout
- Mounting
- Functional test
- EMC test

Power switch:
- Block diagram with state machine
- Schematic
- Simulation
- Print layout
- Mounting
- Testing

FPGA extension board
- Schematic (Temp + humidity circuit + current sensor circuit)
- Print layout
- Mounting
- Functional test

Converters
- 230VAC - 30VDC inverter
- 30VDC - 230VAC converter
- Possible Variable load.

Mechanic:

Box for the hub

Software:
Web application - Database and server side programming
ADC
Daemons
Drivers

1 Time box 1

1.1 Time box planning

Overview of the what work is put into which time boxes.

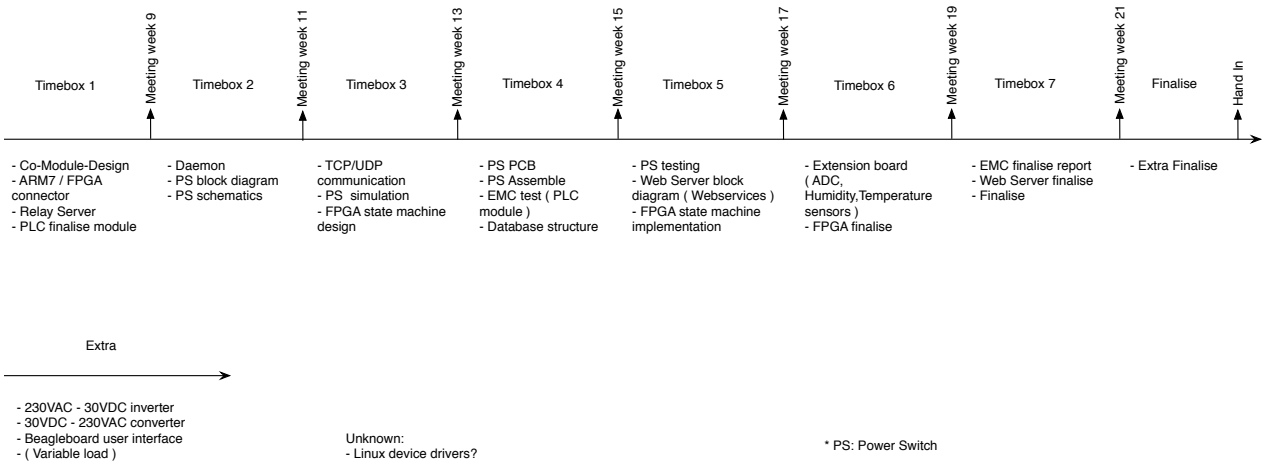


Figure 1.1: Outline of the time-boxes

1.1.1 Work to be done in this time box

- CO-Module-design
- Print connector (connector board between the FPGA board and the ARM board used).
- Relay server
- PLC module (Power line module for communication between the modules)

1.2 Interface analysis (ARM to FPGA) - Dennis

Interfacing the FPGA board (in our case a Spartan6 board by Digilent and the Arm board from EA) can be done in different ways. The fastest way if the number of connections are relatively low, is usually with a bunch of wires between the two boards. But even though such a set-up can be much more flexible, it can be very hard to error searching if something is not working (because of a broken wire or a short circuit) and is also very sensitive to noise. Instead of the wires a PCB solution has been chosen, which will be designed to fit between the two boards pin headers. By placing the boards next to each other an approximation of the board size is found. Size of the board is set to 16 x 5 cm. The connections available on the FPGA boards extension pins is 28, so a 32 bit connection is not a possibility, instead a 16 bit data connection is chosen. The pins routed between the ARM and the FPGA is:

- 16 x Data
- 7 x Address
- 1 x Chip select
- 1 x Write enable
- 1 x Output enable (read enable)
- 1 x interrupt to the ARM processor
- 1 x Reset from the ARM board

The reset pin is used for synchronous reset of the two boards. The interrupt signal makes it possible for the FPGA to interrupt the ARM board. The chip select is used, as two other devices is also connected to the external memory.

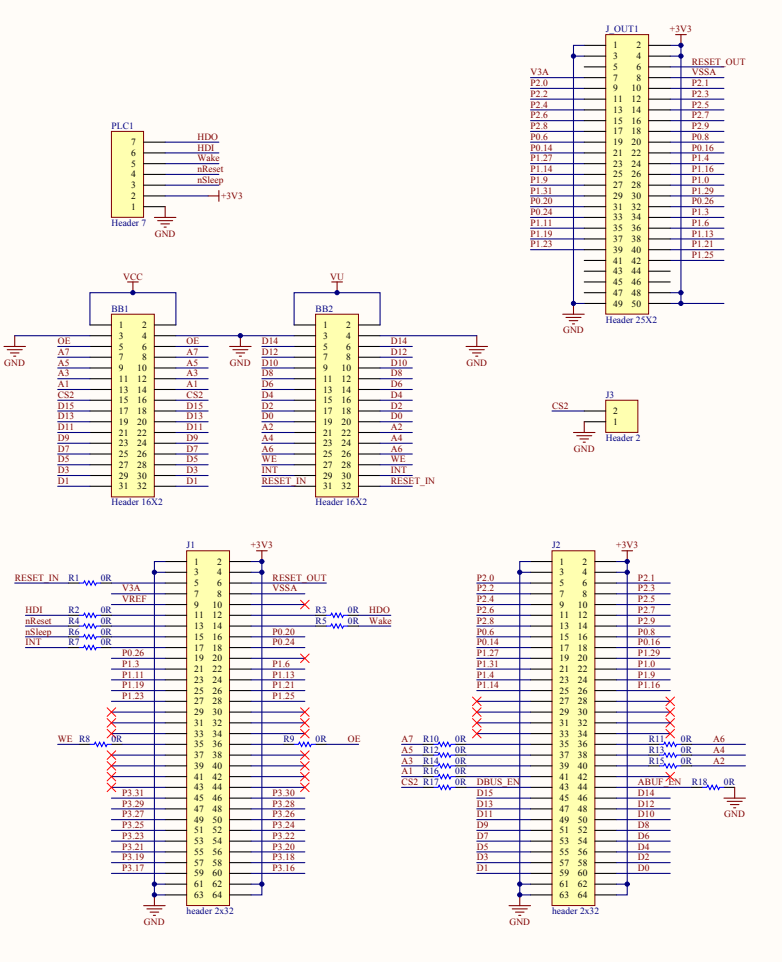


Figure 1.2: Schematic of the ARM to FPGA connector.

A connector for the power line module is also mounted, to easily interface the module with the ARM processor. An pin header with all the unused pins is also added, to make it possible to quickly connect external devices to the processor.

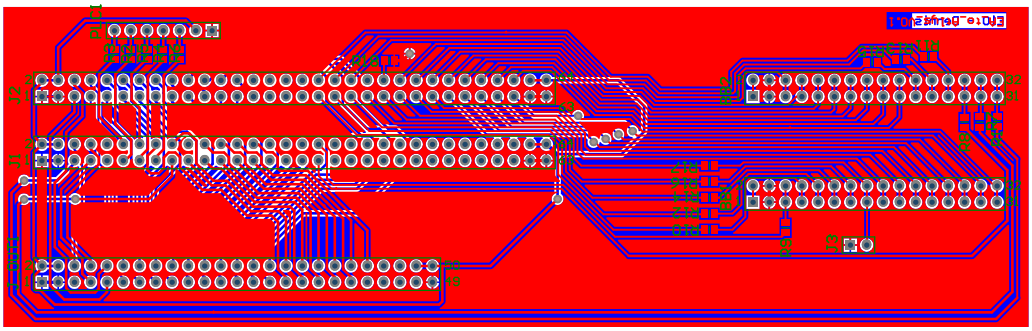


Figure 1.3: PCB of the ARM to FPGA connector (top view)

1.3 Concurrent design - Theis

Concurrent design in the design of embedded system is a method to decided where to place different parts of the system, from the specification and performance requirements. The difference between traditional design and co-design is, in traditional design, two groups of experts design independently hardware and software, to implement when all is finish. In co-design one group is designing the whole system and implement the part where they get the best performance and power use.
Concurrent design has four steps that is:

1. Modelling
2. Partitioning
3. Concurrent synthesis
4. Concurrent simulation

In this time box, only the two first steps are described according to the project time plan.

1.3.1 Modelling

In the modelling phase, the customers needs is taking into consideration, and the platform for the system is choose. In this project the platform is specified from the project requirements, to include an ARM7 CPU with uClinux as processor in the system, and a Spartan 6 or 3 FPGA for hardware parts in the system.

1.3.2 Partitioning

In "technical platform" from the launch phase, the hardware and software specification was made, and the table below shows which parts of the project that can be implemented both in the ARM7 CPU as software and in the FPGA as hardware.

- PLC module
- Power switch
- A/D Converter
- Ethernet
- Web server
- Interface
- Indicator

PLC module is used to communicate with the other modules, this part is best implemented in the ARM7 CPU as software, because it is a data-communication protocols.
Power switch is used to control the switch and is a rather complex algorithm, it have to be dynamic when controlling the switch, because the power consumption and production is never the same, this part is best implemented in software for the ARM7 CPU.
A/D Converter is a very suitable part to implement in the FPGA as hardware, the frequency is high, the memory requirement is small, the task is static. And in this project there is more than one A/D converter, so it

is good to run this part in parallel.
Ethernet is made in software, and supported by the linux kernel. The embedded arm board have a dedicated IC (PHY) to handle communication between ethernet and the CPU
Interface/Indicator is the buttons and LEDs where the user can interact with the system, this is put into hardware, because it is IOs that is best handle in the FPGA for parallel reading the inputs and setting outputs.

1.4 Relay-server - Paulo

Relay server is programmed in C++, this application will run on the development platform so remote access to the EA-LPC2478 board becomes possible.

- Requirements
- Setting the scene
- Server data flow
- Further Implementation
- Documentation (Generated with Doxygen)

1.4.1 Requirements

The requirements given to the Relay Server are:

- FR 1: Able to accept multiple telnet connections
- FR 1.1 Shall accept both TCP and UDP connections
- FR 1.2 Shall be able to connect to the daemon on the target
- FR 13 Shall relay traffic unmodified between the telnet client and the daemon
- NF 1: Shall be programmed in C++
- BR 1: Handling more than one connection
- BR 1.1: If more than one active let the others wait - give a message
- BR 1.2 When a blocking connection is terminated the next waiting connection shall be connected to the daemon

1.4.2 Setting the scene

The C program was created as close as possible to what the daemon is supposed to achieve. This will give advantages in the daemon programming (Time Box 2) since the communication with the TCP protocol is done.

The "daemon" is a TCP echo server, it sends the same message trough the same socket back to the relay server.

The relay server only forwards messages from the client (by TCP or UDP) to the "daemon" without changing any data.

1.4.3 Server data Flow

The final server (daemon) is an echo server, it only reply the last received message, this kind of server is used for debug, to be sure the communication is being established.

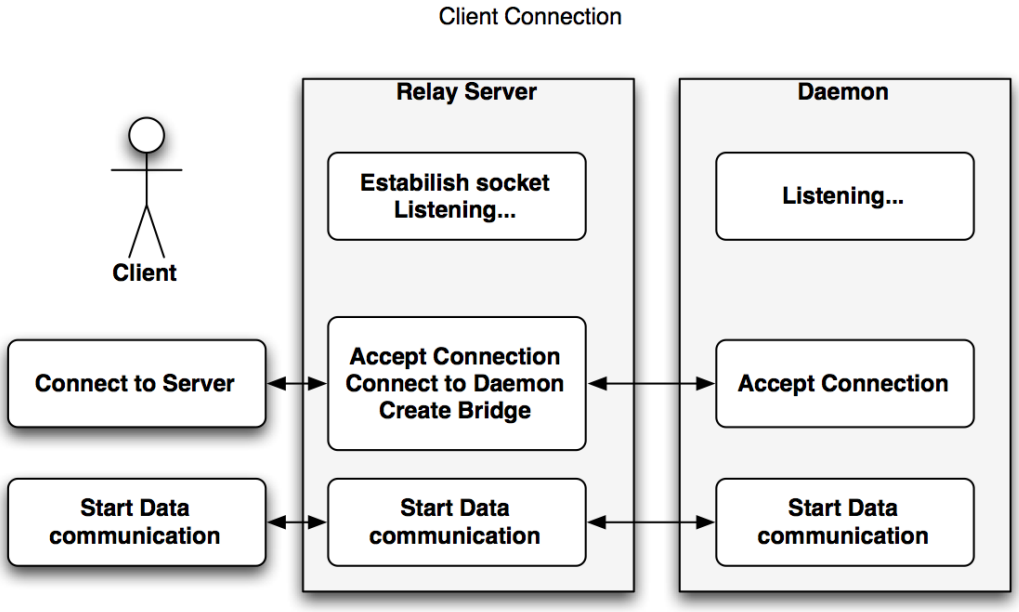


Figure 1.4: TCP Data flow between Client, Server, Daemon - Client connection

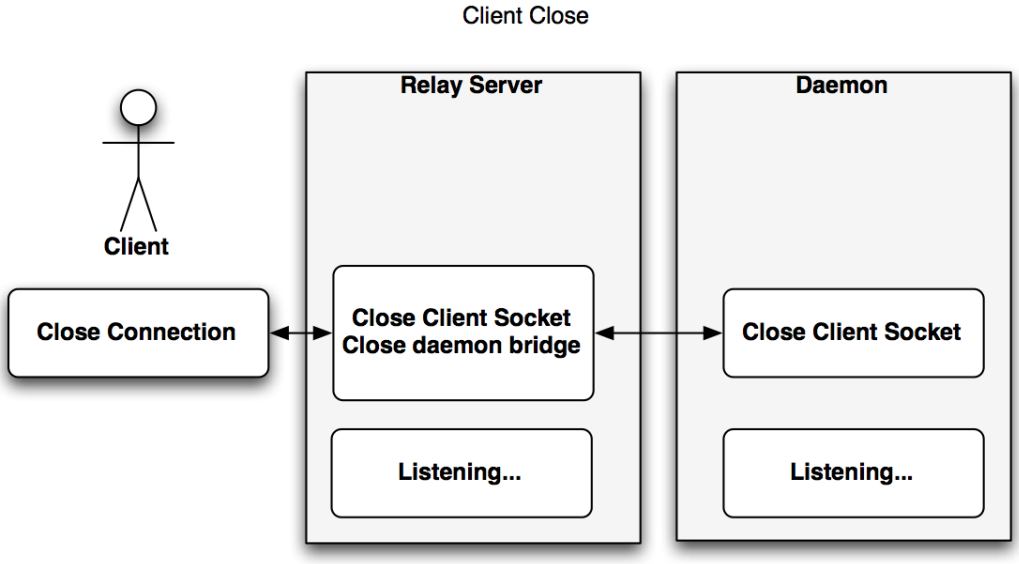


Figure 1.5: TCP Data flow between Client, Server, Daemon - Client disconnect

1.4.4 Further Implementation

- UDP connection not yet implemented in Time Box 1, this will be implemented in Time Box 2.

- Implementation of interrupts for read an write on the sockets will improve performance, this will be implemented in Time Box 2 with the "daemon".

1.4.5 Documentation (Generated with Doxygen)

Documentation can be found in the external document *RelayServer_Documentation*

1.5 Power Line module - Dennis

The documentation of the Power Line module can be found in a separate report called: *EPRO 3 & 4 PLC - Hardware Interface* as the work is done in corporation with another team. So far the first prototype has been mounted and tested, which has lead to some improvements of the design. The second prototype is ready for mounting and testing which will also be described in the design document.

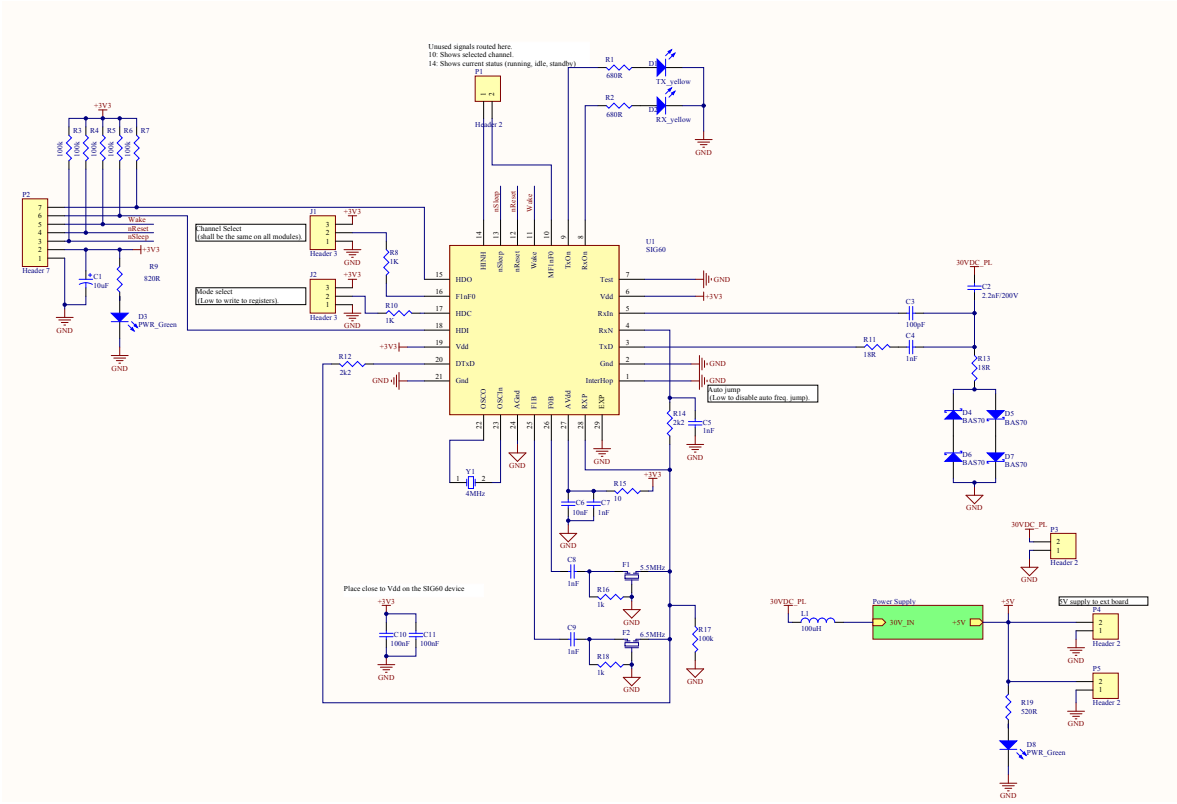


Figure 1.6: Power line circuit version 0.2

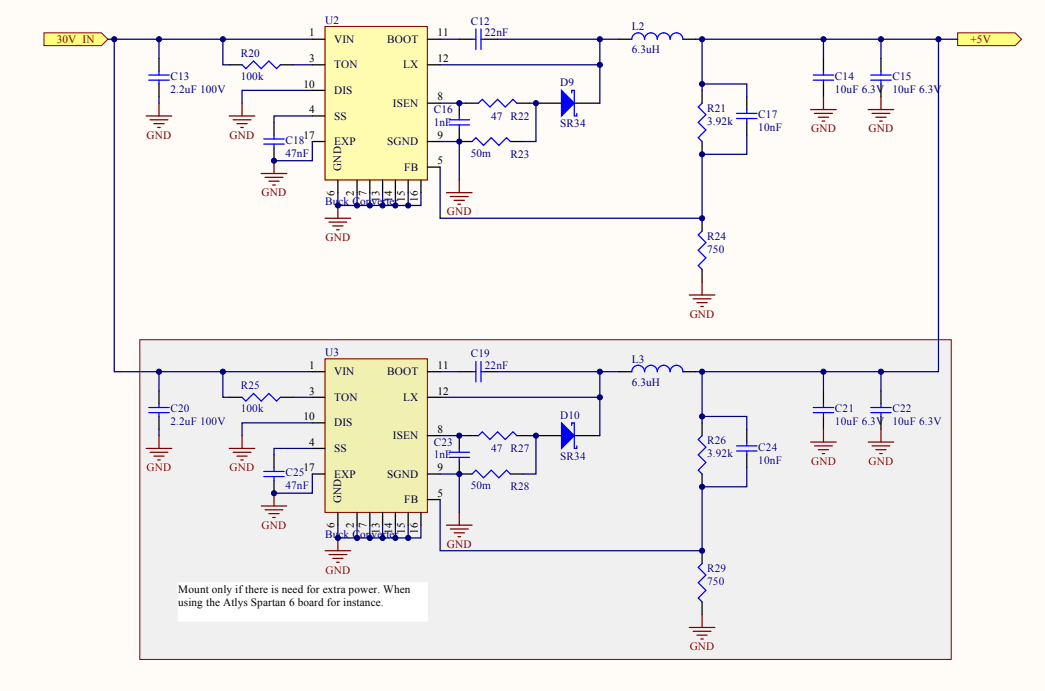


Figure 1.7: Power supply. 2 x 5 volt 3 ampere version 0.2.

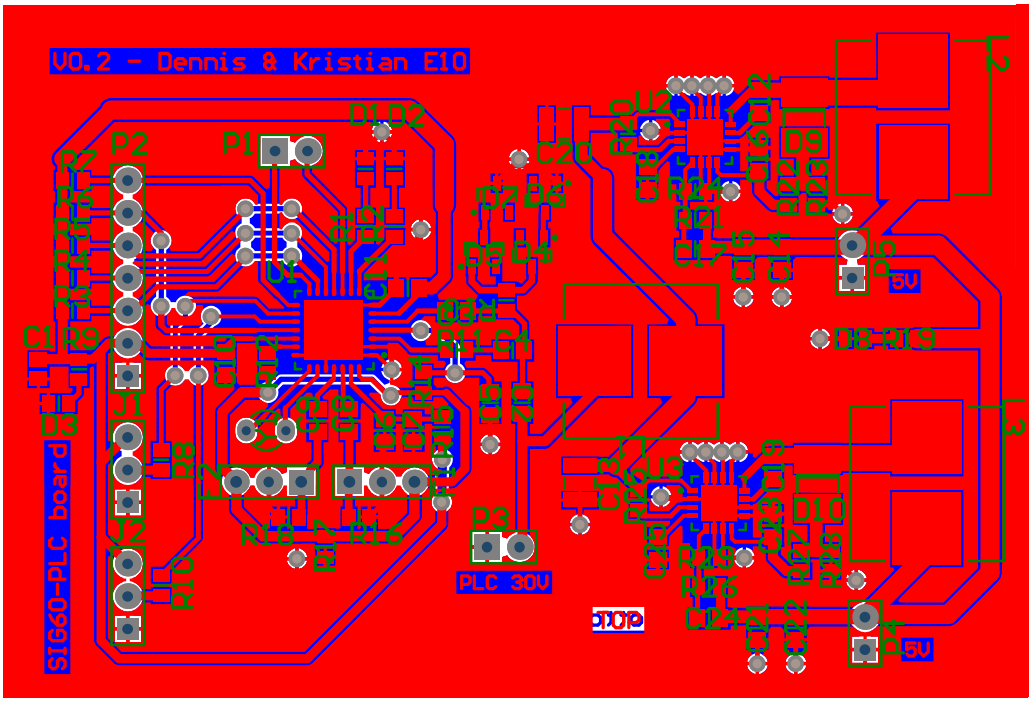


Figure 1.8: PCB layout of the power line circuit and the 5 volt power supply version 0.2.

2 Time box 2

2.1 Time box planning

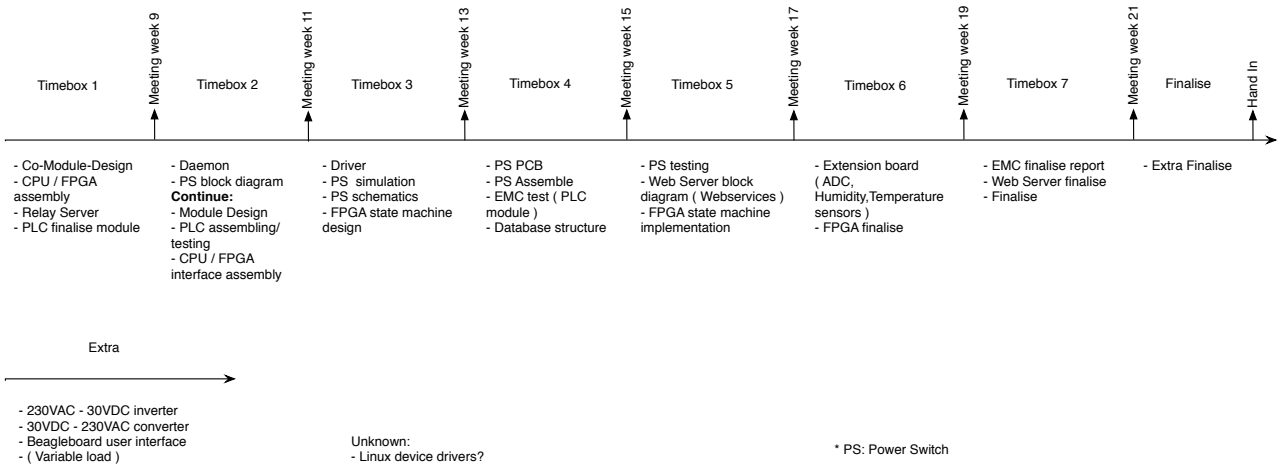


Figure 2.1: Updated time-box

2.1.1 Work to be done in this time box

- ARM7 to Spartan6 interface
  - Mount
  - Test
- PLC module
  - Mount x6
  - Test
- Daemon
  - Overview
  - Background process on uClinux
  - The rc file (startup file)
    - Configure network
    - Start daemon as background process
  - Further implementations
  - Documentation
- Power switch
  - Block diagram
- Module design

Description:

ARM to Spartan6 interface The interface between the ARM7 CPU and the Spartan6 board, has to be mounted and tested.

PLC module The powerline module has to be mounted 6 times, in order to provide one for all the groups. The boards should also be tested and debugged.

Daemon Create a program that runs as background process, there is no direct control between user and the application.

Power switch Block diagram of the power switch, which shall activate the different modules connected to it (wind-turbine, photo-voltaic cells etc.) and control the current flow.

Module design is an overview of the part connection in the system

2.2 ARM to Spartan6 interface - Dennis

The first version of the PCB was hard to solder because of the small pad sizes. A second version of the interface has been made, with increased pad size and increased clearance between the tracks. Also the Chip select connector pin has been more centralized on the PCB to decrease cable length. The hole in the one site of the PCB is needed to avoid removing a connector on the arm-board, the tracks beside this connector have been rerouted to give more space for this connector.

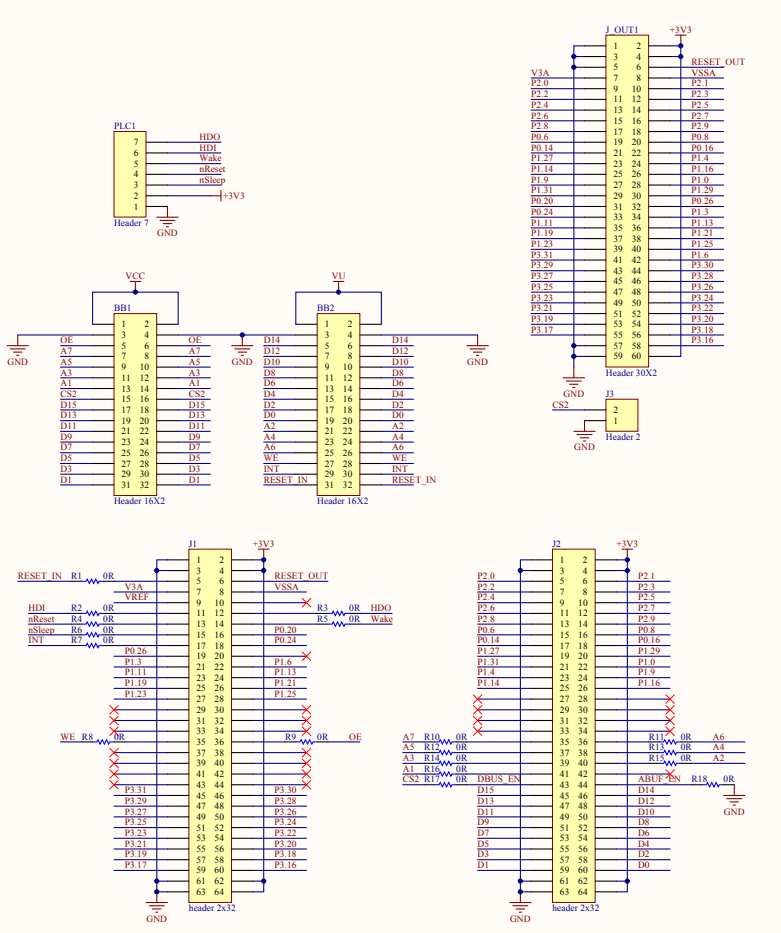


Figure 2.2: Schematic of the ARM to Spartan6 connector, v0.2.

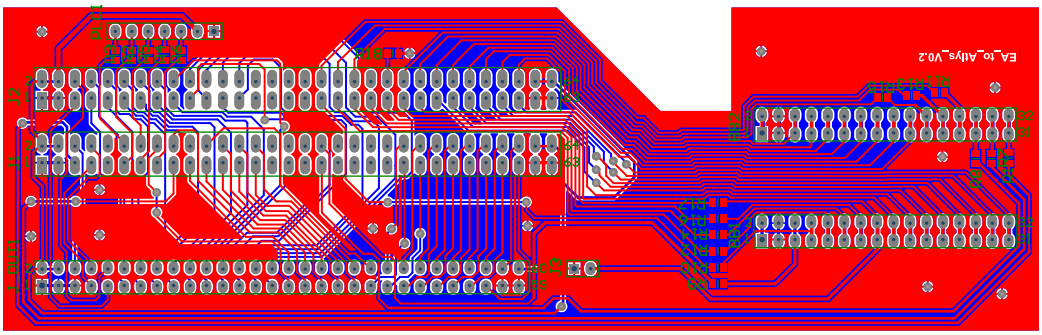


Figure 2.3: PCB of the ARM to Spartan6 connector, v0.2 (top view)

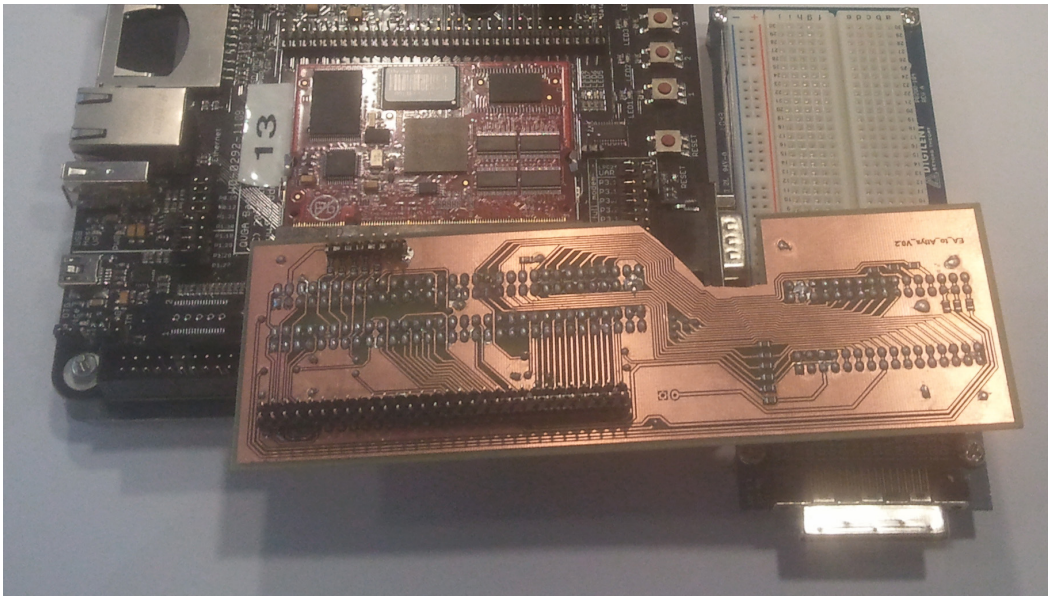


Figure 2.4: Picture of the mounted interface PCB. The connector in the bottom right of the picture is the connection to the Diligent Spartan6 board.

2.3 PLC module - Dennis

The second version of the Power line module was decreased in size, which lead to problems when making the PCB, as a lot of tracks short-circuited. A third version with larger clearance between the tracks and the ground plane has been made. The PCB has been mounted and tested, this is describe in the *EPRO 3 & 4 PLC - Hardware Interface* report. As a short resume, the communication between two boards through power line works without seeing any invalid characters (test period of 15 min.). The 5 volt power supply has been tester to withstand at minimum 2.4 Amp. where a voltage drop of approximately 0.1Volt was observed. A single error has been found and corrected before ordering and mounting the 6 PLC boards needed (one for each module in the system/group).

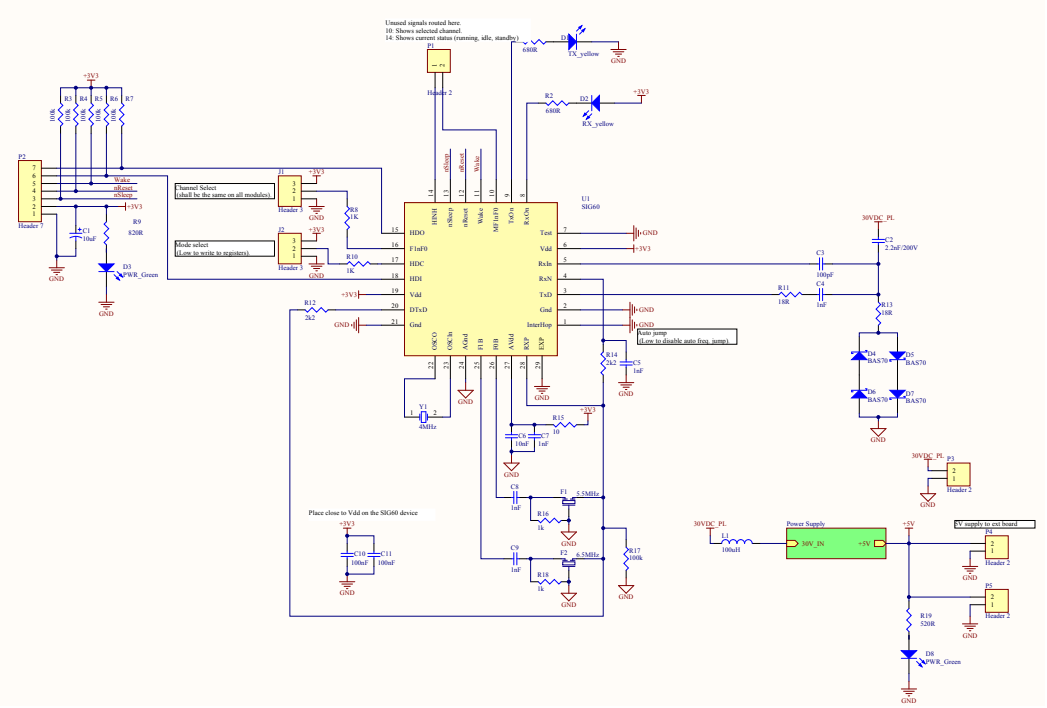


Figure 2.5: Power line circuit version 0.4

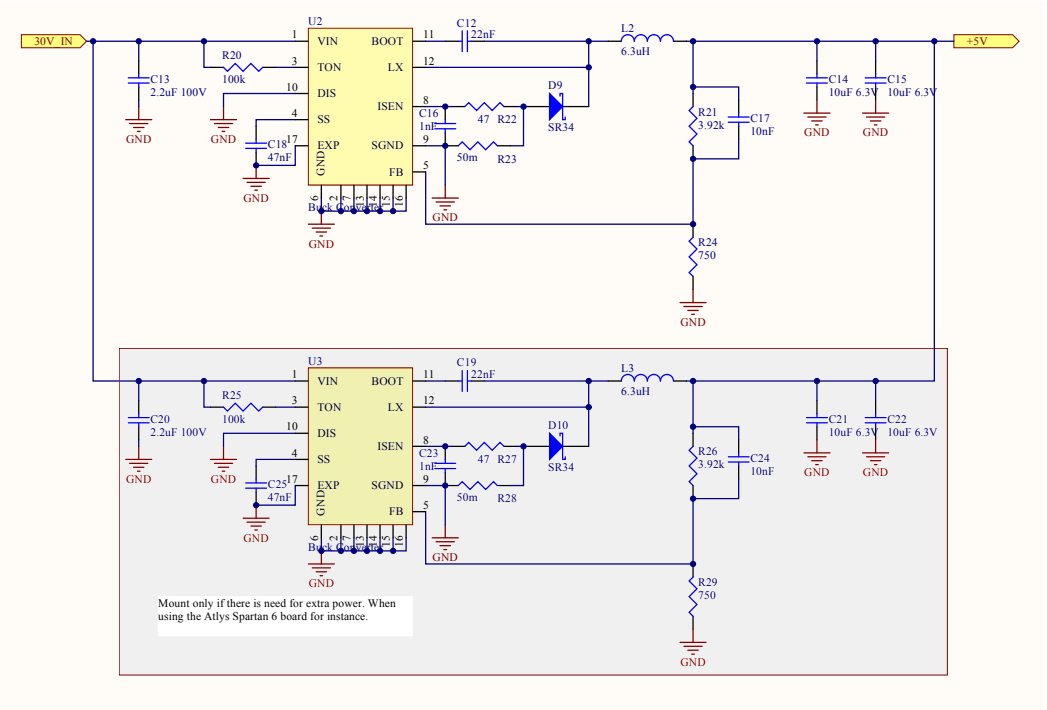


Figure 2.6: Power supply. 2 x 5 volt 3 ampere version 0.4.

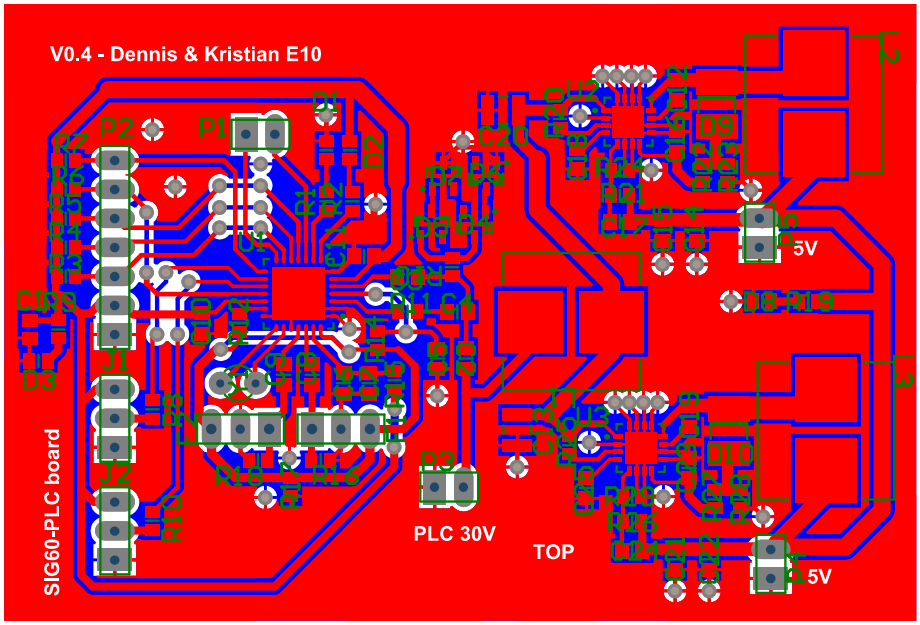


Figure 2.7: PCB layout of the power line circuit and the 5 volt power supply version 0.4.

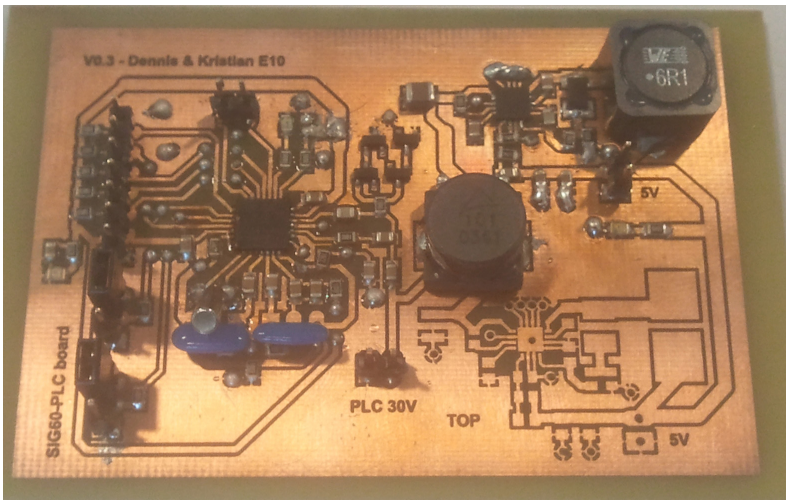


Figure 2.8: Picture of the mounted PLC board (note, only one of the supplies is mounted)

2.4 Daemon

2.4.1 Section Contents

- Overview
- Background process on uClinux
- The rc file (startup file)

  - Configure network
  - Start daemon as background process

- Further implementations
- Documentation

2.4.2 Overview

Daemon is a program that runs as background process, there is no direct control between user and the application.

The background application is going to establish a socket (way of communication through network), that will allow TCP connections to port 5555 of the LPC2478 Development board running the uClinux. This distribution can be found at [1].

In further development of this application, the user is able to get status from the device by connecting to it using a TCP client (Putty, coolTerm, HyperTerminal, etc).

In this version the user is able to light all the four LEDs on the development board by typing the command: led <number of the led to light on><on/off>

2.4.3 Background process on uClinux

A background process is an application that is disassociated from the initial terminal.

To make a application run as a process in UNIX environment:

- Create a separate process.
- Detach the process from the parent.
- Reset file mask.
- Change the current directory.
- Close the standard files (STDIN, STDOUT, STDERR).

uClinux distribution runs on embedded system and since most of them don't have a MMU (memory management unit), the fork() function is not allowed in this implementation.

The fork() function when called causes the creation of a new process (child process). The child process will be a exact same copy of the parent process being this impossible without a MMU. To implement the same functionality the vfork() function is used, this function creates a new process but the memory address is shared by the parent and child.

2.4.4 The rc file (startup file)

The rc is a file that contains the startup instructions for the operation system in use (uClinux) this file can be found at the specific vendor folder. This file contains commands that will run automatically each time the system starts.

[1] `http://www.uClinux.org`

Configure Network When the system starts the network have to be configured so the device can be part of our network. The configuration can be done manually each time with the commands:

ifconfig eth0 x.x.x.x netmask x.x.x.x up

route add default gw x.x.x.x

being x.x.x.x ip adresss.

Defining this commands in the rc file will make the network to be always accessible after a reboot is needed, for example is a update is implemented to the software.

Start daemon as background process The daemon is defined in this script by the command: daemon & , this way the application run as background when the operating system starts, allowing the communication between the device and the user.

2.4.5 Further implementations

- A background application will be implemented to update data on a Mysql server and retrieve commands from the web server, this will be the pipe between all the modules in the system.

2.4.6 Documentation

This daemon (background application), is a TCP echo server, this means it will answer back the same message that was received. This was explain in Time box 1 along with the development of the relay server.

In this TCP echo server, a command handler was implemented, this allow the user to turn on and off the LEDs on the development board: **led <number of the led to light on><on/off>**

The daemon is calling the application already existent in the development board made by Embedded Artists as example.

2.5 Power switch - Dennis

The power switch is the part of the hub that traffics the energy from and to its connected modules, such as: wind-turbine, photovoltaics, hydrogen-cell, battery-storage, load modules and similar. Inspiration to the creation of the power switch has been found at: Powerhub Systems [2] (routing system) and TORC [3] (implementation and control of the hub).

2.5.1 Block diagram

The switch is divided in 10 similar blocks (one for each module). Figure 2.9 shows two modules connected to the power line. The content inside the dotted line block is an abstract description of the power switch. For simplicity two diodes has been used, to control the routing direction.

[2] http://www.pwrhub.com/
[3] http://www.torcrobotics.com/products/powerhub

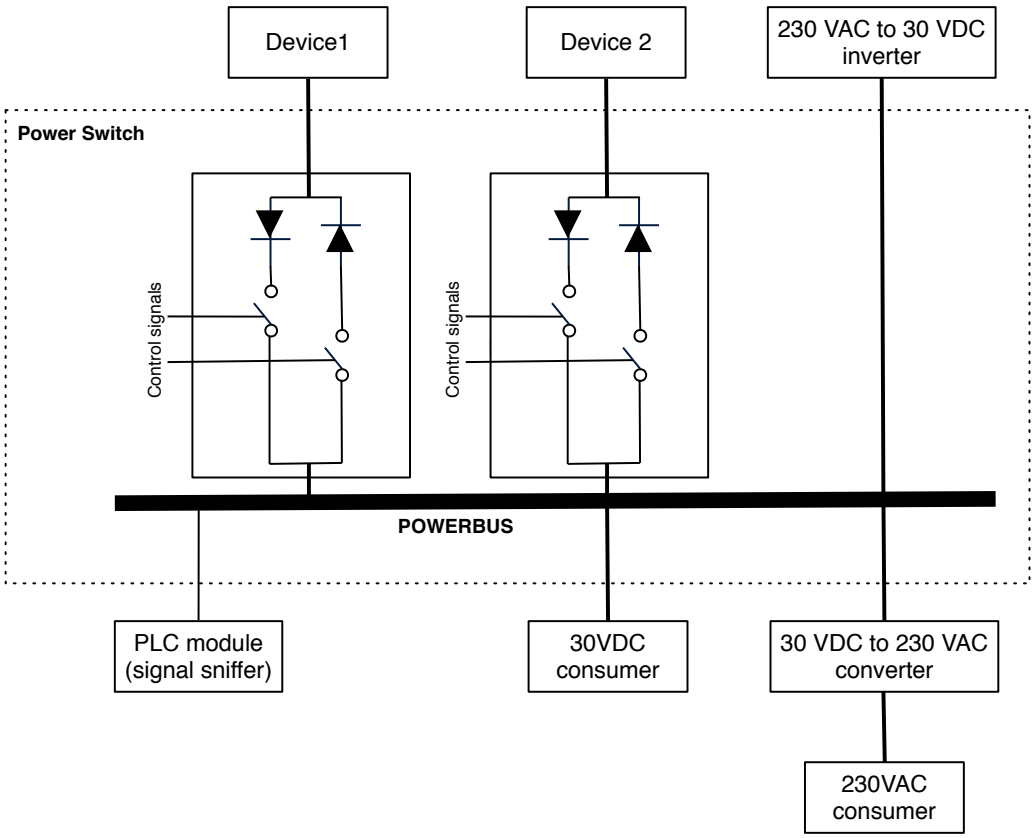


Figure 2.9: Block diagram of the Power Switch v0.1

The inverter is used to supply the system at startup time. The system is also supplied through the inverter if the energy-producing- and energy-storing modules cannot deliver enough energy.

Notice that noise from switching devices on and off shall be minimized to not bother the power line module and in worst case manipulate with the data-packages.

Two signals from the processor will be needed to control each of the switch blocks, however a control section might be added to the switch to decrease the number of pins needed to control all 10 blocks + consumers.

Current measuring for the consuming devices and the inverter will be implemented in the switch, to keep track of how green the system is and if the producers are producing anything.

2.5.2 Diagram and components

Some alternative components to diodes have been found, in order to control the current direction and have as little power loss as possible in the switch.
Component suggestions:

- Control device for MOSFETS, LTC4357[4] from Linear Technology.

[4] `http://www.linear.com/product/LTC4357`

- Demo board for LTC4357[5].
- Power Mosfets IRFP4468PbF[6] with low on resistance and maximum drain current of approximately 200Amps. *Rds*
- If needed, an dual power rectifiers has been found: MBR60H100CT[7], with max. voltage on 100V and max. current on 60 Amps.

An initial diagram drawing have been made with two LTC4357 devices. The diagram shows the possible implementation of a single switch block.

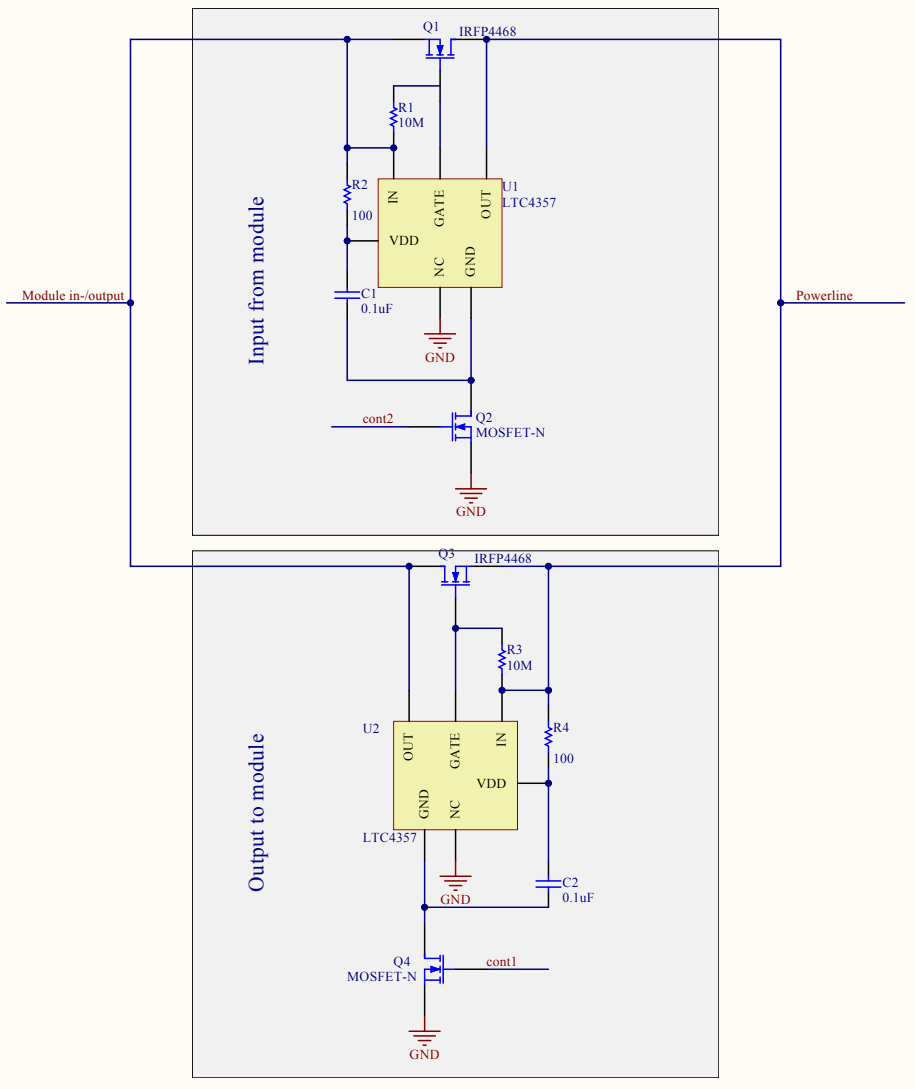


Figure 2.10: Power switch, diagram of a single switch block.

[5] http://cds.linear.com/docs/Demo%20Board%20Manual/dc1203A.pdf
[6] http://pdf1.alldatasheet.com/datasheet-pdf/view/234175/IRF/IRFP4468PBF.html
[7] http://pdf1.alldatasheet.com/datasheet-pdf/view/172142/ONSEMI/MBR60H100CT.html

The system makes use of two LTC4357 to control the two MOSFETS used as diode substitutions. So far no protection circuit have not been added to the diagram, or multiplexing for the control signals, but only the basic concept of the switching mechanism.

2.6 Module design

The module design is to give an overview of the part connection in the system. On the module design is shown which parts shall be made in software and which parts shall be made in hardware. The connection between the parts is also shown, plus the connection to external parts.

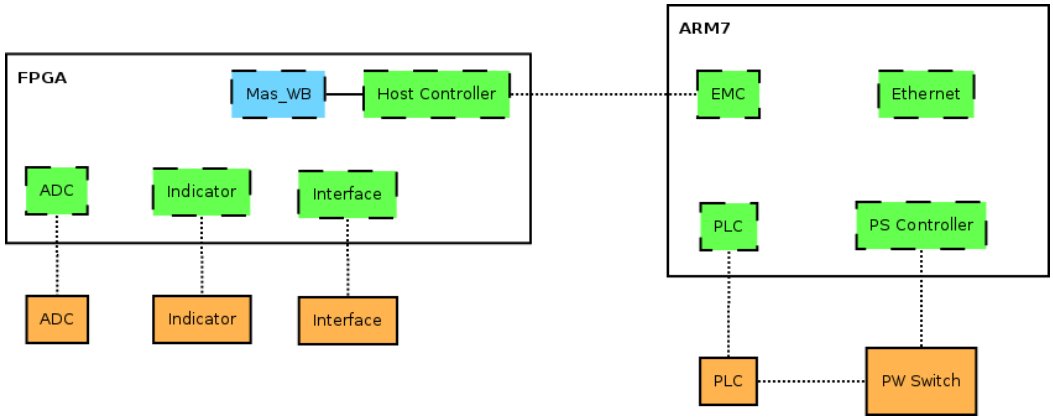


Figure 2.11: Module diagram

Power line module
In the table below the pin names on the PLC module and their corresponding pin on the ARM7 is listed.

PLC-pin	ARM-pin	Other
HDO	p0.11	RXD2
HDI	p0.10	TXD2
Wake	p0.13	LED-B
nReset	p0.12	Port-PWR-B
nSleep	p0.18	MOSI

Table 2.1: Power line module connection

Interface
The pins for the interface on the Spartan6 is listed in the table below

Push buttons	**LOC**	**BANK**	**Pin name**
0	T15	3	IO_L1N_M0_CMPMISO_2
1	N4	3	IO_L1P
2	P4	3	IO_L2P
3	P3	3	IO_L2N
4	F6	3	IO_L55P_M3A13
5	F5	3	IO_L55N_M3A14
Switchs			
0	A10	0	IO_L37N_GCLK12
1	D14	0	IO_L65P_SCP3
2	C14	0	IO_L65N_SCP2
3	P15	1	IO_L74P_AWAKE_1
4	P12	2	IO_L13N_D10
5	R5	2	IO_L48P_D7
6	T5	2	IO_L48N_RDWR_B_VREF_2
7	E4	3	IO_L54P_M3RESET

Table 2.2: Spartan6 interface pins

Indicator
The Spartan6 pin names for the LED indicators on the board is listed in the table below

Indicator LEDs	**LOC**	**BANK**	**Pin name**
0	U18	1	IO_L52N_M1DQ15
1	M14	1	IO_L53P
2	N14	1	IO_L53N_VREF
3	L14	1	IO_L61P
4	M13	1	IO_L61N
5	D4	0	IO_L1P_HSWAPEN_0
6	P16	1	IO_L74N_DOUT_BUSY_1
7	N12	2	IO_L13P_M1_2

Table 2.3: Indicator Spartan6 pins

Analog to digital converter
On the Spartan6 board there is a 8 pin connector that shall be used for the analog to digital converter. The pins for the connector is shown in the table below

ADC pins	**LOC**	**BANK**	**Pin name**
0	T3	2	IO_L62N_D6
1	R3	2	IO_L62P_D5
2	P6	2	IO_L64N_D9
3	N5	2	IO_L64P_D8
4	V9	2	IO_L32N_GCLK28
5	T9	2	IO_L32P_GCLK29
6	V4	2	IO_L63N
7	T4	2	IO_L63P

Table 2.4: Analog to digital converter connector

Host controller
The pin connection between the Spartan6 and ARM7 board is listed in the table below

IO	LOC	EMC pin	ARM7 pin
1	U16	OE	BOE
2	U15	A7	BA7
3	U13	A5	BA5
4	M11	A3	BA3
5	R11	A1	BA1
6	T12	CS2	DBUS_EN
7	N10	D15	BD15
8	M10	D13	BD13
9	U11	D11	BD11
10	R10	D9	BD9
11	U10	D7	BD7
12	R8	D5	BD5
13	M8	D3	BD3
14	U8	D1	BD1
15	V16	D14	BD14
16	V15	D12	BD12
17	V13	D10	BD10
18	N11	D8	BD8
19	T11	D6	BD6
20	V12	D4	BD4
21	P11	D2	BD2
22	N9	D0	BD0
23	V11	A2	BA2
24	T10	A4	BA4
25	V10	A6	BA6
26	T8	WE	BWE
27	N8	INT	P0.22
28	V8	RESET_IN	RESET_IN

Table 2.5: Address data bus between the Spartan6 and AMR7

With the module design, the design of the different parts is straight forward. The module design show what shall be hardware and software, plus it shows witch pins is used for the specific part in software and hardware.

3 Time box 3

3.1 Time box planning

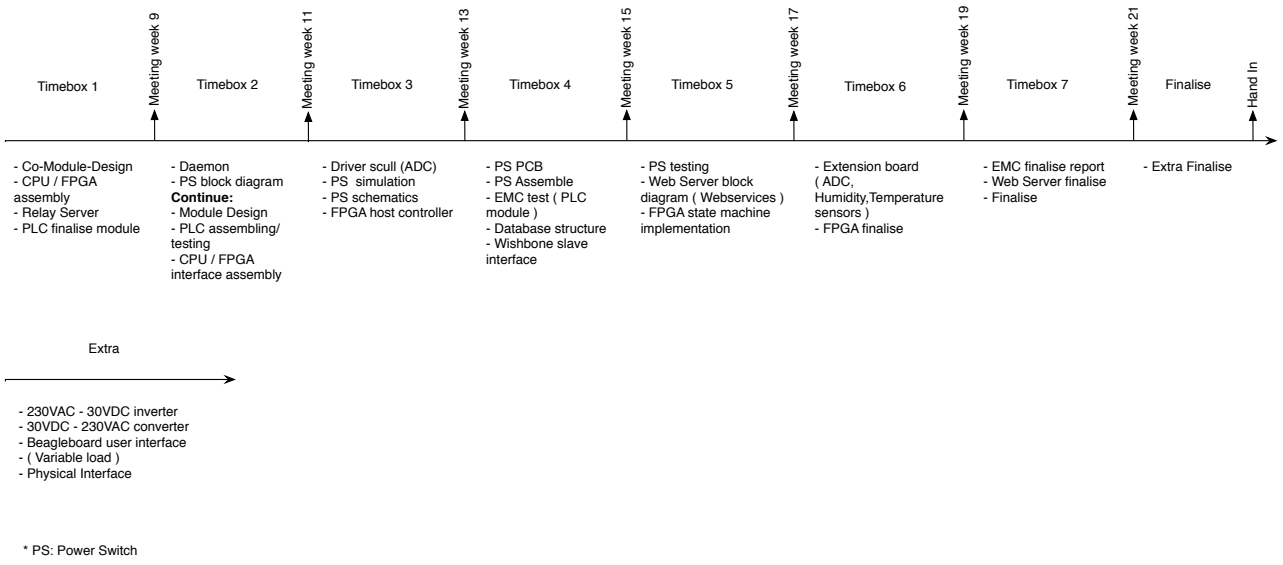


Figure 3.1: Updated time-box

3.1.1 Work to be done in this time box

- Device driver
  - Create a device driver framework from the scull example and the device drivers included in the distribution.
  - Convert the ADC driver into a device driver.
- Power switch
  - Schematic
  - Printed circuit board
  - Mount component
  - Power Switch Test
  - Further Implementations
- Webserver Communication
  - Data Flow
  - Setting the Server
  - Further implementations
- Host controller
  - Master wishbone
  - CPU interface

Description:

Power switch A single switch port board for the power switch is made, as an essential part for the power switch.

Host controller is made in the FPGA, and is responsible for the communication between the Spartan6 and the ARM7, it shall be made with wishbone master interface

3.1.2 Time planning

	Power Switch	Web server	Host Controller	Device Driver
Estimation	8	4	8	10
Actual	10	6	12	13

Table 3.1: Estimation and actual time used on the project

3.2 Power switch - Paulo

Power switch module have to be able to switch the power direction. A battery after fully charged can be used to power other modules or systems.

The power switch module have to be able to switch the power direction so for instance a battery fully charged can be used to power other modules or systems but also to protect the different modules connected to the hub.

3.2.1 Schematics

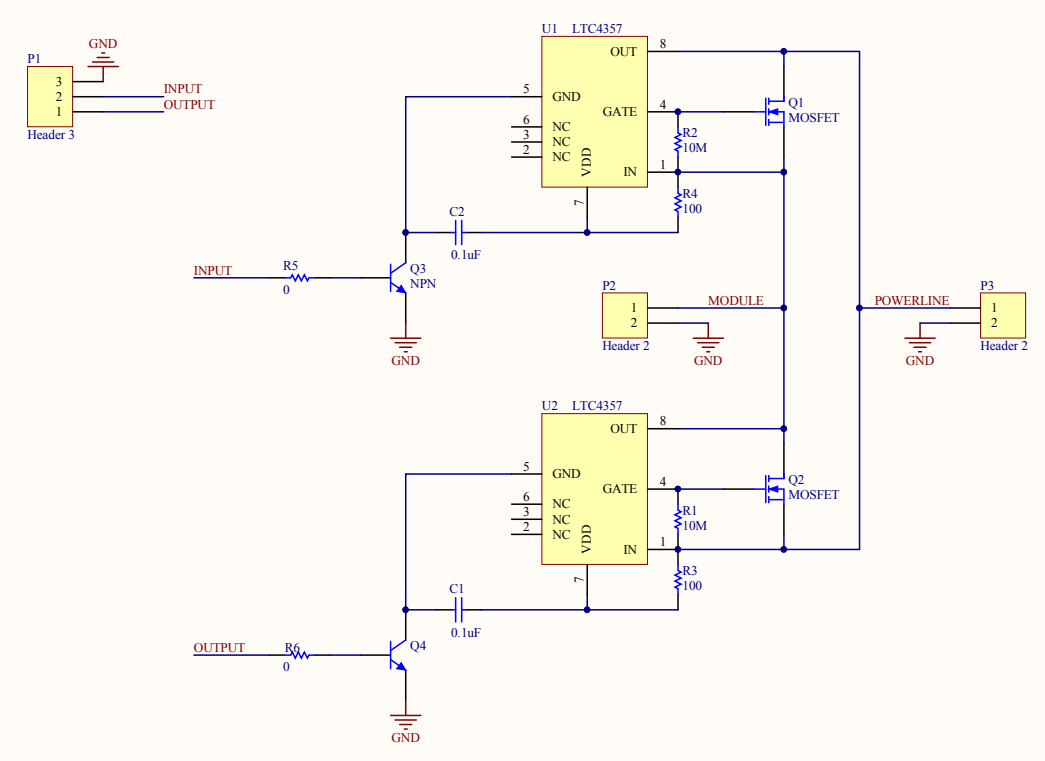


Figure 3.2: Schematic of the switch interface

3.2.2 Printed circuit board

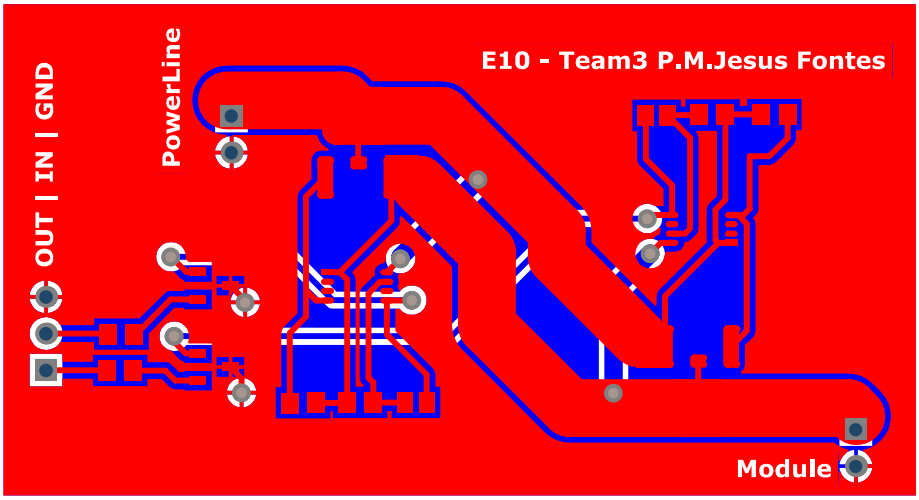


Figure 3.3: PCB layout

3.2.3 Mount component

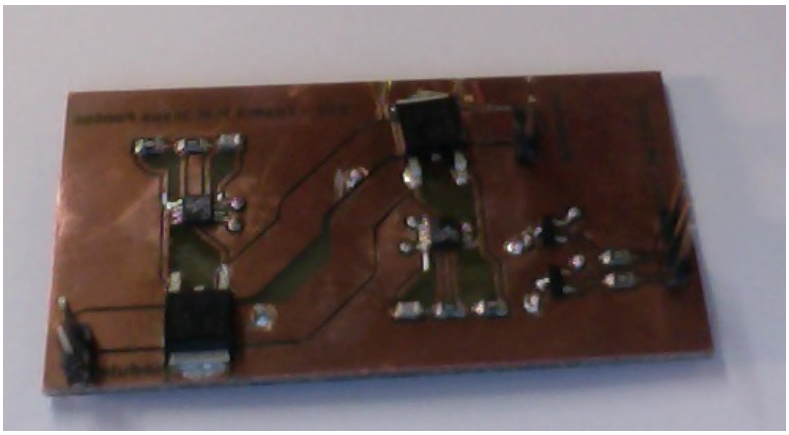


Figure 3.4: Mounted PCB

3.2.4 Power Switch Test

To test our circuit a load is applied on the module side and a source of 12V in the power-line connection. In a second scenario the opposite is done where a load is connected to the power-line and a source of 12V to the module side. With this configuration is possible to test the current flow in both directions using the switch component of this circuit.

Test overview Scenario 1:

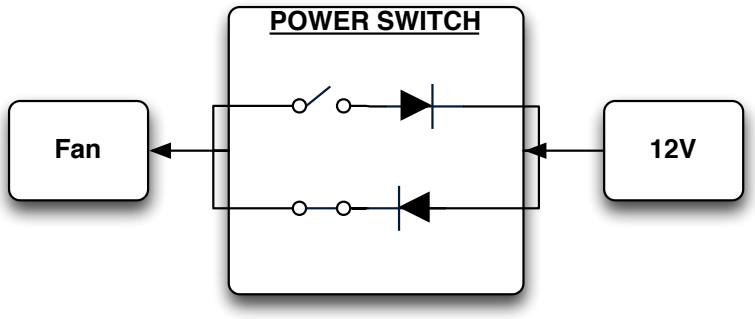


Figure 3.5: Current flow from the power-line to the module (fan)

Scenario 2:

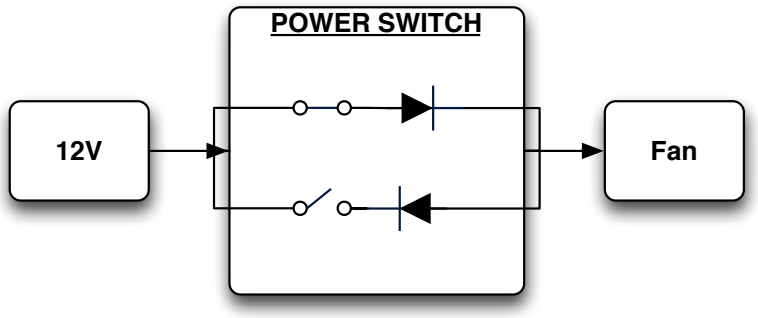


Figure 3.6: Current flow from the module to the power-line (fan)

3.2.5 Further Implementation

- The power switch module does not work as expected, unfortunately current is also flowing in the opposite direction. Further tests have to be done in order to find a solution for this situation.

- A current sensor have to be add to each switching port (10 ports), to surveillance the connected modules and react if a high/low current is measured. From the measurements it is also possible to verify the efficiency of the green system.

- The module will connect to a main board with a card edge connection, so it will be easier to substitute in case of a malfunction and keep the housing free of many wires.

3.3 Webserver Communication - Paulo

3.3.1 Data Flow

In the implementation of this system the user is able in an interactive way to command the power-switch. With this example is possible to send any command to the uClinux distribution running on the ARM7. A webpage has been made with 4 check lists, each one corresponding to each LED on the board, on click the LEDs are switched on and off.

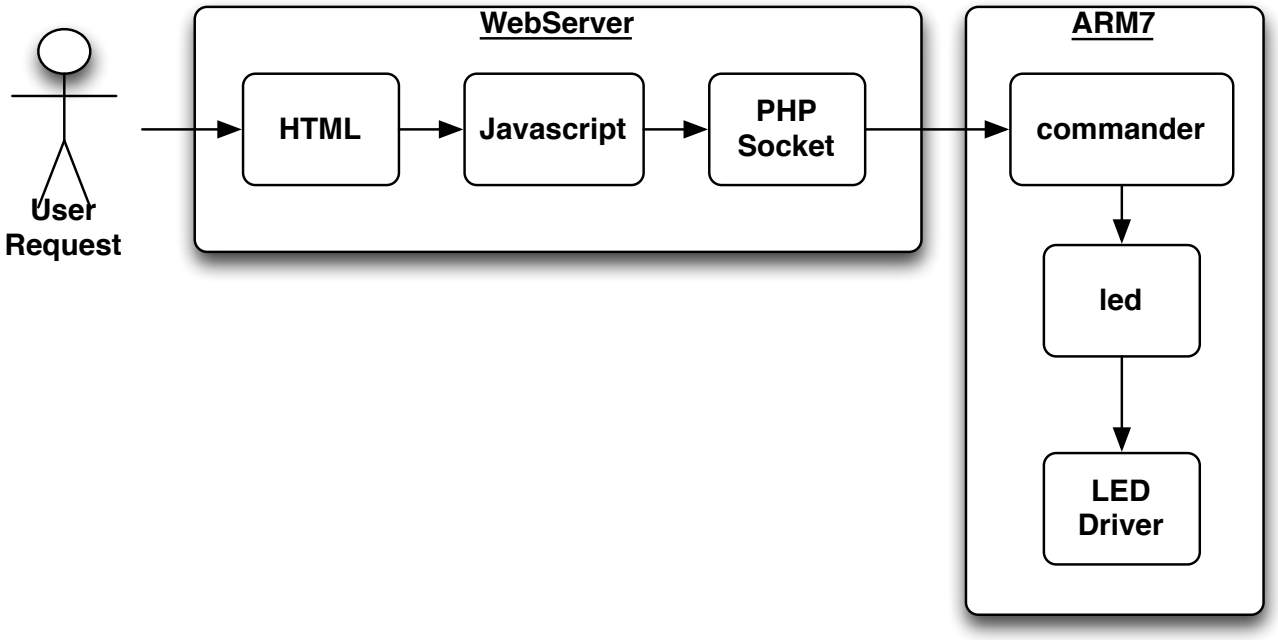


Figure 3.7: Data Flow between user and ARM 7

3.3.2 Setting the Server

The server was set on the development environment and can be access at 10.1.18.223, no online status is given from the board. This webpage is used as verification of the development process with a low level interaction.

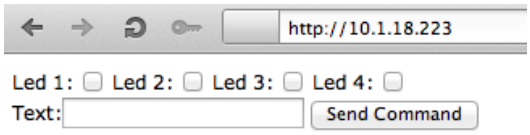


Figure 3.8: Low level web interface

The PHP code creates a socket and connects to the ARM7 commander application, a TCP server listening to port 5555. The command is send to this application and then send to the shell, executing the application led from embedded artists:

Connection: connect.php

```
<?php

  function con(){

    require_once("globals.php");

    if ($_SESSION['socket']=socket_create(AF_INET, SOCK_STREAM, SOL_TCP)){
      echo "Socket created <br>";
    }

    if (socket_connect($_SESSION['socket'],$device_ip,$device_port)){
      echo "Connection stablished<br>";
    } else {
      exit (socket_strerror(socket_last_error()));
    }
    echo socket_read($socket,1024)."<br>";
  }
?>
```

Command Handler: cmd.php

```
<?php

  session_start();

  include("connect.php");

  con();

  //Construct command
  $str = $_POST["cmd"];

  socket_write($_SESSION['socket'],$str);

  echo socket_read($_SESSION['socket'],1024);

  socket_close($_SESSION['socket']);

  $_SESSION["alive"]=false;

?>
```

Global Setting such as device IP and Port, the device IP in further implementation will be stored in the MYSQL database since the DHCP is activated on the board so we can have connection to the board even when a dynamic ip address is given.

globals.php

```
<?php

  //$device_ip = "127.0.0.1";
  $device_ip = "10.42.26.130";
  $device_port = 5555;

?>
```

The use of AJAX(Asynchronous Javascript And XML), allows the interface to be more dynamic and don't have to reload each time some action is taken, this is made in background by the XMLHttpRequest(). The code can be seen bellow, this use the method post to send variables and values.

```
function sendRequest(str){
```

```
  var req;
  if(window.XMLHttpRequest){
    req = new XMLHttpRequest();
  } else {
    req = ActiveXObject("Microsoft.XMLHTTP");
  }

  req.open("POST","cmd.php",false);
  req.setRequestHeader("Content-type","application/x-www-form-urlencoded");

  req.send("cmd="+str);
  alert(req.responseText);

}
```

3.3.3 Further Implementations

- Implement the user interface generated in the Interactive Design sessions.
- Both direction communication is to be implemented, so the data can be retrieved from the board to the user.
- Website is active only when the board is on-line, otherwise the interface is inactive and a warning is sent to administrator.

3.4 Device driver skeleton - Dennis

The goal is to create a simple framework for creating device drivers for the LPC2478 platform. To do so, the scull example provided is used together with sample code provided with the uClinux distribution. The framework is tested by converting an ADC driver (running on the bare metal) into a device driver. Down below code snippets from the ADC is extracted an commented.

The module registration is found in the *file_operations* structure:

```
static struct file_operations adc_fops = {
  .owner    = THIS_MODULE,
  .read     = adc_read,
  .open     = adc_open,
  //.write    = none,
  .release = adc_close,
};
```

The structure contains pointer values to the given functions, where these functions can be accessed from user space by:

Events	User Functions	Kernel Functions
Load Module	insmod	module_init()
Open	fopen	file_operations: open
Read	fread	file_operations: read
Write	fwrite	file_operations: write
Close	fclose	file_operations: release
Remove	rmmod	module_exit()

Table 3.2: Function event table.

Loading and removing a module is not included in the file operation structure, but declared aside as pointers to the two functions used:

```
// Define init and exit functions
module_init(adc_mod_init);
module_exit(adc_mod_exit);
```

In the init function several different things are done:

- Setting up the registers (copy from the *bare metal* driver file).
- Registration of the device (no need for dynamic numbering, so the major number is a static number not used by other drivers.)
- Allocating the char driver and adding it to the environment.

Code pieces from setting up the ADC:

```
//Register the device
  dev = MKDEV(adc_major, adc_minor);
  result = register_chrdev_region(dev, NUM_ADC_DEVICES, "adc");
//Allocate and add the adc device.
  adc_cdev = cdev_alloc();
  cdev_init(adc_cdev, &adc_fops);
  adc_cdev->owner = THIS_MODULE;
  adc_cdev->ops   = &adc_fops;

  result = cdev_add(adc_cdev, dev, NUM_ADC_DEVICES);

static void adcInit(void){
  volatile u32 tmp = 0;
  m_reg_bfs(PCONP, (1 << 12)); //Power on ADC
  m_reg_write(AD0CR, 0);
  m_reg_write(AD0CR,
      (4<<8)      | //set clock division factor
      (0<<17)     | //CLKS  = 0, 11 clocks = 10-bit result
      (1<<21)     | //PDN   = 1, ADC is active
      (1<<24));     //start a conversion
  }
```

The exit function undoes the steps done in the initialization function.

Before reading the ADC channels, the channel shall be opened for reading. In the ADC code the pins are setup to be of type ADC. As the pins belongs to different PINSEL registers, a if else statement is used to set the right registers.

```
// inside the function:
static int adc_open(struct inode* inode,  struct file* file){

// Check if the channel has been verified
if(chRefCnt[channel] == 0){
  file->private_data = (void *) channel;
  if(channel >= 0 && channel <=3)
    m_reg_bfs(PINSEL1, enable_bits[channel]);
  else if(channel > 3 && channel < 6)
    m_reg_bfs(PINSEL3, enable_bits[channel]);
  else if(channel >= 6 && channel < 8)
    m_reg_bfs(PINSEL0, enable_bits[channel]);
}
chRefCnt[channel]++;
```

When finish reading, the file is closed again and the opposite operation is performed (deselect the ADC function for the proper IO pin).

When reading the ADC, the result retrieved is converted from decimal into a string before it is returned to the user. Reading and writing to and from the CPU registers is done through the kernel space functions *m_reg_read* and *m_reg_write*. So the register writes and reads in the driver file for the ADC is performed as parameters sent to these functions:

```
//start conversion now (for selected channel)
m_reg_write(AD0CR, ((1 << channel) | (1 << 24)));

//wait til done
while ((m_reg_read(AD0GDR) & (1<<31)) == 0);

//get result from global register and adjust to 10-bit integer
result = (m_reg_read(AD0GDR) >> 6) & 0x3FF;
```

3.4.1 Testing the driver - Dennis & Paulo

To proper test the driver, a small user space program has been written. The program opens the device, reads it and closes it again. The argument sent to the program selects which channel should be read (ad0, ad1, ad2 etc.).

```
#define ADC "/dev/adc"

  //Format string to "Analog value"
  //Ex 1023 -> 3.30
void stringFormat(int value){
  char str[10];
  if((value%100)>9)
    sprintf(str, "%i.%i",value/100,value%100);
  else
    sprintf(str, "%i.0%i",value/100,value%100);

  printf("%s\n",str);
}

int main(int argc, char *argv[]) {
  FILE *fp;
  char read[10];

  if ((fp = fopen(strcat(ADC,argv[1]),"r"))==NULL){
    printf("Cannot open file.\n");
    exit(0);
  }

  fread(read,1,10,fp);
  stringFormat(atoi(read)*330/1023);
  fclose(fp);
  return 0;
}
```

The application is compiled with the uClinux image using the tutorial at Klaus Kolles wiki[8]

3.5 Host controller - Theis

The host controller takes commands from the ARM7, and controlling blocks in the Spartan6. This controller is made in hardware and implemented in the Spartan6.

[8] http://klaus.ede.hih.au.dk/index.php/How-to_add_a_user_space_application_to_uClinux

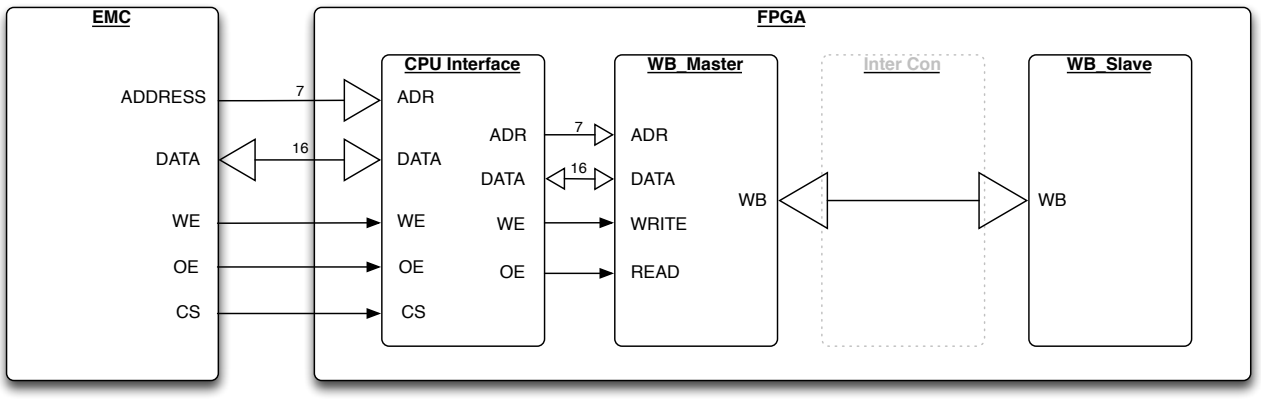


Figure 3.9: Host controller

3.5.1 External memory controller

The EMC is a memory controller peripheral that support asynchronous static memory device such as RAM, ROM and flash. The chip select is used to select which memory device the ARM7 wants to communicate with. A block diagram of the EMC is shown below.

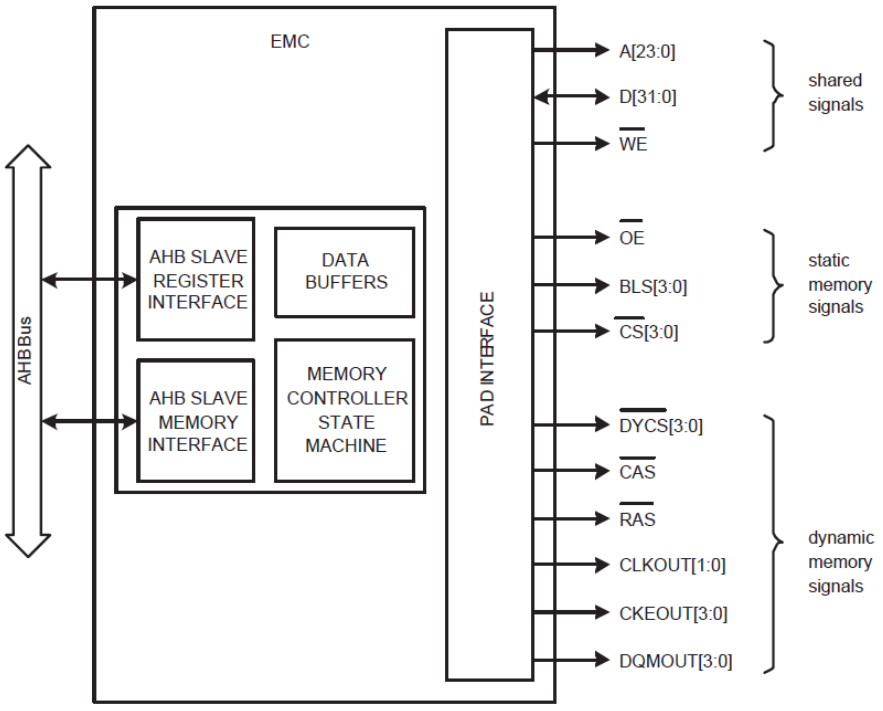


Figure 3.10: EMC block diagram

The EMC is used to communicate with the Spartan6 board, with a 7 bit address and a 16 bit data bus, plus a read, write and chip select signal to indicate if it wants to read or write data, and select that it is the Spartan6

it would like data from. In this project the EMC is only used in static mode without the byte lane selects. A diagram of the EMC block is shown below.

3.5.2 CPU interface

The CPU interface communicate with the EMC on one site and with the master wishbone on the other site. The purpose of this interface is to determine if the EMC wants to communicate with the Spartan6, and if the ARM7 wants to read or write data.

```
process (Clk,Rst)
  begin
    if Rst = '1' then               --Reset set everything to 0
      Wr_o  <= '0';
      Rd_o  <= '0';
      A_o   <= (others => '0');
      D_o   <= (others => '0');
      CpuD_o  <= (others => '0');
    elsif (Clk'event and Clk = '1') then
      if CpuCs_i = '1' then         --Check chip select
        A_o <= CpuA_i;              --Adress routing
          if CpuRd_i = '1' then     --Reading
            Rd_o  <= CpuRd_i;
            Wr_o  <= CpuWr_i;
            CpuD_o  <= D_i;         --Wishbone data out to Cpu data input
          elsif CpuWr_i = '1' then  --Writing
            Wr_o  <= CpuWr_i;
            Rd_o  <= CpuRd_i;
            D_o   <= CpuD_i;        --Cpu data output to wishbone data input
          else
            Wr_o  <= CpuWr_i;
            Rd_o  <= CpuRd_i;
            D_o   <= CpuD_i;
            CpuD_o  <= (others => '0');
          end if;
      else                          --If chip select not high everything is set to 0
        Wr_o  <= '0';
        Rd_o  <= '0';
        D_o   <= (others => '0');
        A_o   <= (others => '0');
        CpuD_o  <= (others => '0');
      end if;
    end if;
end process;
```

The block is synchronous with the clock and have an asynchronous reset. If the reset is pressed everything is set to zero to tell the master wishbone to not do anything. When the chip select is high, it tells the block that the ARM7 want to communicate with the Spartan6. After chip select is activated, the block checks if the ARM7 want to read or write data from or to the Spartan6. If the ARM7 wants to read data, the interface block routes the data from master wishbone to the data vector on the EMC. Otherwise if the ARM7 wants to write data, the data vector on the EMC is routed to the data input for the master wishbone. In any case of read or write the read and write output from the EMC is routed to the master wishbone. The address input from the EMC is routed as soon as the chip select is activated. If the chip select is not active, everything is set to zero, to secure that the master wishbone do not write any data. The data output to the EMC is tri-stated in the wrapper code if the chip select is not active, to not disturb other communication on the EMC bus.

Test bench The test bench is made to test if the host controller is responding correct to different signals, image of the test bench signals is shown below.

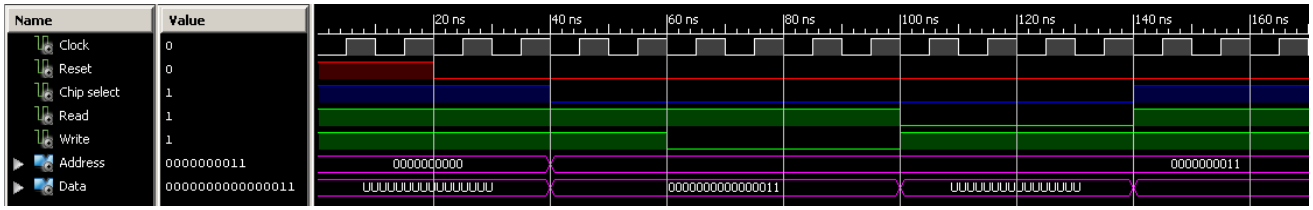


Figure 3.11: Host controller test bench

The test bench shows that after the reset state and when the chip select is active, a write cycle is made,to address "0x03" and the data that is written is "0x03". After the write, a read cycle on address "0x03" and the data read is uninitialized, this is because the master wishbone do not get any data from slave, so it is not possible to send any valid data to the ARM7, so the uninitialized indicate so far that the read cycle works. And after the chip select deactivate the data is equal to the data from the ARM7.

3.5.3 Wishbone

Wishbone is a computer bus for integrated circuit communication. It sets up some standard communication rules to use when designing IP cores. This makes it easy to reuse code on different hardware and in different systems. The wishbone bus is a logic bus, this means that there is no rules for voltage levels, it only works with ones and zeros. In this project the wishbone is used as illustrated on the picture below.

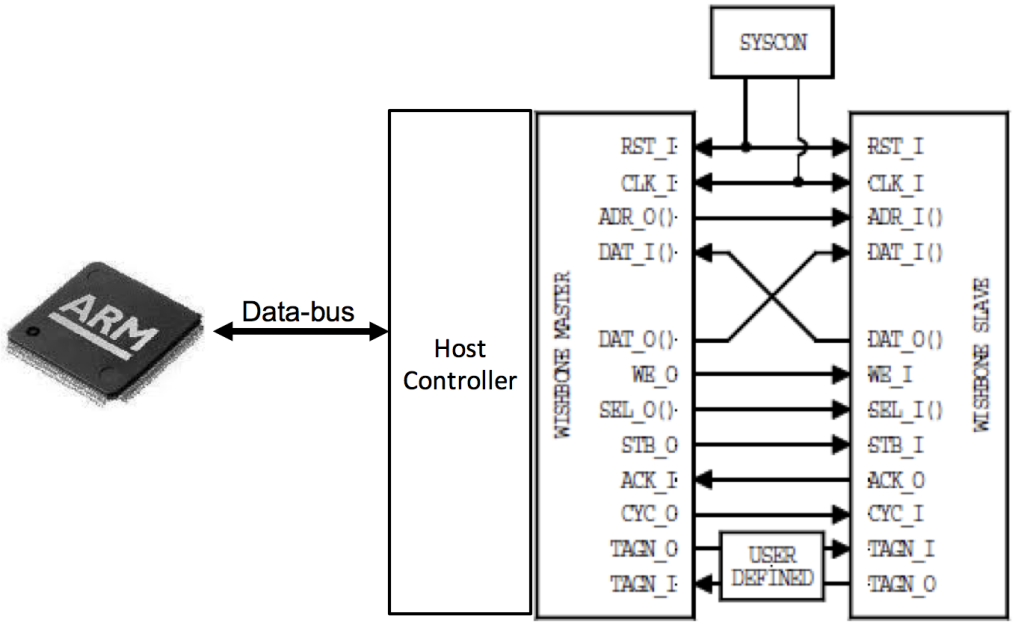


Figure 3.12: Wishbone connection for the host controller

The EMC on the ARM7 is controlling the wishbone master, which is controlling all the wishbone slaves that is in the system. This make it easier to write a driver for the ARM7 that controls the master wishbone through the EMC. Then the master wishbone control every IP core with at wishbone slave interface connected. In this project the wishbone is going to use single read and write cycles, the single read cycle timing diagram is shown

below.

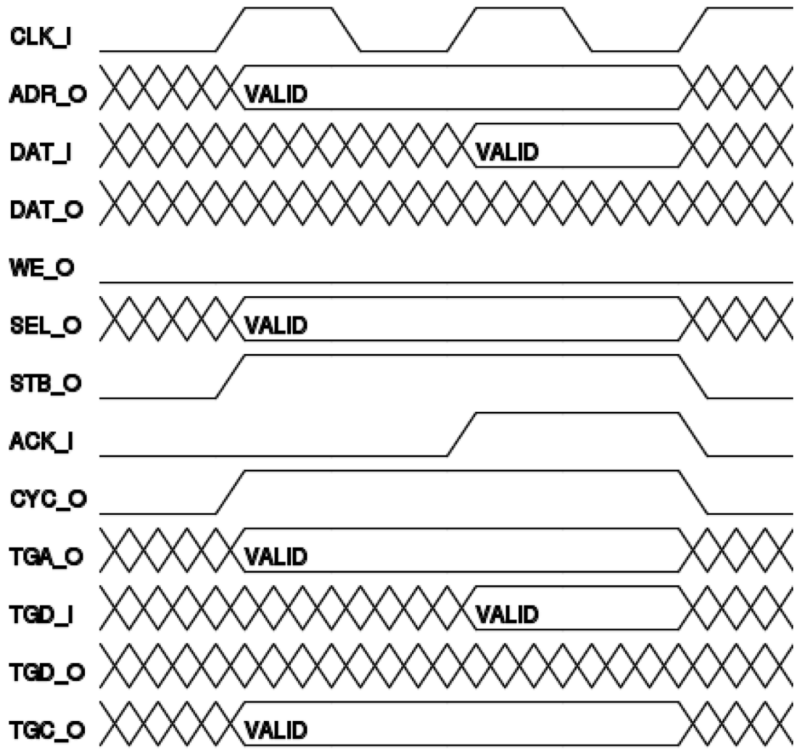


Figure 3.13: Single read cycle

When a single read cycle is started, the master wishbone presents a valid address, on the address register, it sets the write enable to zero to show that this is a read cycle, the SEL_O is set, to show where on the data input it expect the data to be read, it also sets the CYC_O and STB_O high to indicate the start of a cycle and a phase.

On the next raising clock edge, the slave decodes the input and sets the acknowledge bit high in response to the SEL_O. It also presents valid data on the slave data output. The master monitors the acknowledge and prepares to latch data.

On the third raising clock edge the master wishbone latches the data, and set the STB_O and CYC_O to zero to end the cycle and the phase, the slave set the acknowledge bit low again as response to STB_O.
The timing diagram for the single write cycle is shown below.

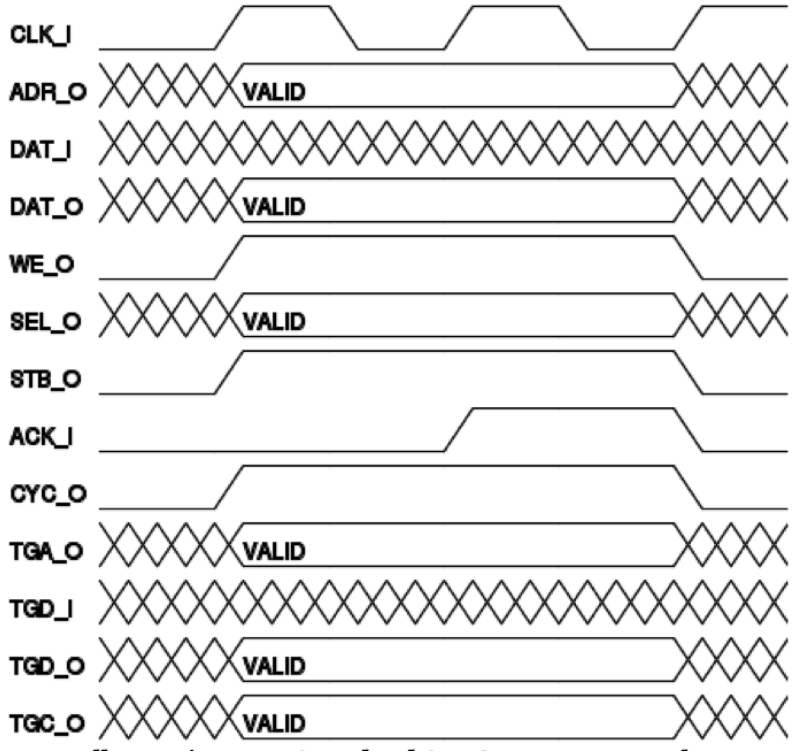


Figure 3.14: Single write cycle

When the master wishbone wants to write to the slave, it present valid address and data on the address register and data output, it also sets the write enable high to indicate the cycle is write. As with the read the STB_O and CYC_O is set high to indicate cycle and phase start, all this is done on the first raising clock edge.

On the second raising clock edge the slave decode the input and set the acknowledge bit high in response to the SEL_O. It also prepares to latch data from the master. The master monitors the acknowledge bit and prepare to terminate the cycle.

On the last raising clock edge the slave latches the data, the master set the STB_O and CYC_O to zero to end the cycle and the phase, the slave set the acknowledge bit low again as response to STB_O.

3.5.4 Further Implementation

The host controller work in theory but still needs to be tested in practice. The handed out code, was well commented to give a good overview. This made it easier to make the code for CPUinterface.

3.6 Post deployment

A short evaluation of the time box. The time plan has almost been on track, only a few extra hours have been used.

3.6.1 Power switch

Unfortunately the power switch implementation does not work as expected. Fixes for how to solve the problem shall be investigated and possible an alternative method found.

3.6.2 Host Controller

The controller is running on test-bench level and needs now to be physically implemented in the FPGA.

3.6.3 Webserver Communication

The web server is up and running and a small script to communicate with the ARM board has been made.

3.6.4 Device driver

A device driver for the ADC is running, this framework will further on be used to implement the external memory controller.

4 Time box 4

4.1 Time box planning

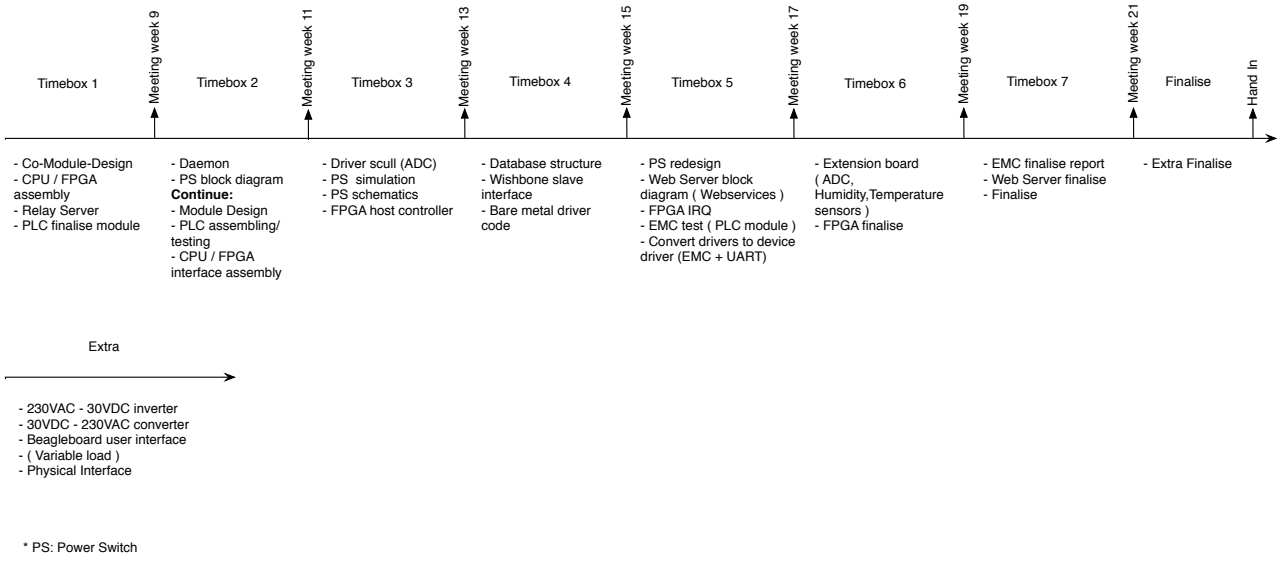


Figure 4.1: Updated time-box

4.1.1 Work to be done in this time box

- Driver
  - UART and EMC bare-metal drivers.
- Database
  - Database design and implementation
  - Webserver implementation
- Physical interface
  - Wishbone slave interface
  - Switch driver
  - LED driver

Description:

Driver UART driver to run on the bare-metal of the ARM7 to verify the PLC module. EMC bare metal test code to verify the code implemented in the FPGA.

Database stores the measurements retrieved my the modules in a efficient way.

Physical interface is the light indication and the switches that shall be on the physical hub interface

4.1.2 Time planning

	Driver	Database	Physical interface
Estimation	5	10	12
Actual	3.5	9	15
Developer	Dennis	Paulo	Theis

Table 4.1: Estimation and actual time used on the project

4.2 Driver - Dennis

In order to implement the UART and the EMC (external memory controller) as a device driver in the linux kernel, it is easy to verify the initialization, read and write functions by making a small driver which runs on the bare metal of the CPU.

4.2.1 Design

No diagrams are made, as the drivers is only implemented with a init, write and a read function.

4.2.2 Implementation

External Memory controller initialization
From the framework.c file, the initialization of the EMC is already made, as the controller is already setup and uses by external NOR flash (CS0, 4MByte in size, address 0x80000000-0x80FFFFFF) and a NAND flash (CS1, 128MByte in size, address 0x81000000-0x81FFFFFF).

```
// In the lowLevelInit function
...
    // Enable EMC
  EMC_CTRL = 0x00000001;
    // Turn on the EMC (power up)
  PCONP |= 0x00000800;
    // Enable p2.14 as Chip Select2.
  PINSEL4 = 0x10000000;
    // Data pins setup (D0-D15, 16 bit)
  PINSEL6 = 0x55555555;
    // Address pins setup (A0-A15, 16 bit). Note, only A1-A7 is routed to the FPGA.
  PINSEL8 = 0x55555555;
    // OE, WE + pins used by the two flash devices setup (BLS, cs0, cs1, A16-A19).
  PINSEL9 = 0x50555555;
    // Select 16 bit data width
  EMC_STA_CFG2 = 0x00000081;

    //Timing registers
  EMC_STA_WAITWEN2  = 0x2;
  EMC_STA_WAITOEN2  = 0x1;
  EMC_STA_WAITRD2   = 0x03;
  EMC_STA_WAITPAGE2 = 0x1f;
  EMC_STA_WAITWR2   = 0x1f;
  EMC_STA_WAITTURN2 = 0xf;
```

Reading a value
To read a value from a register in the FPGA, the correct address for CS2 range can be read as if it was just a variable read.

```
#define EXT_HW_BASE_ADDR  0x82000000
#define HW_SW_REG     (*(volatile unsigned short *)(EXT_HW_BASE_ADDR + 64))

...
  // Reading the FPGA's switch register
Sval = HW_SW_REG;
```

To print the read value as a binary value, a simple function has been made, which goes through all 16 bits in the string, checks if they are 0 or 1 and prints it out.

```
  // Write out the read 16 bit value on binary form.
bin_prnt_byte(Sval);

void bin_prnt_byte(short x){
   int n;
   for(n=0; n<16; n++){
      if((x & 0x8000) !=0){ //Verify if MSB is high or low and print result
         rprintf("1");
      }
      else{
         rprintf("0");
      }
      if(n==3 || n==7 || n==11){
         rprintf(" "); /* insert a space between nibbles */
      }
      x = x<<1; //Shift value one to the left
   }
   rprintf("\n\r");
}
```

Writing a value

Writing a value to the FPGA is equal to reading a value. When a define of the wanted address is made, writing to this address automatically sets CS2, WR, the address pins and reads the data.

```
#define EXT_HW_BASE_ADDR  0x82000000
#define HW_LED_REG    (*(volatile unsigned short *)(EXT_HW_BASE_ADDR + 192))

...
  // Writing to the FPGA's diode register
  HW_LED_REG = Sval;
```

Initializing the UART

Initialization of UART2 used to communicate with the PLC module.

```
...
  // Calling the init UART2 and set Baud rate 9600.
  initUart(UART2, B9600(Fpclk), UART_8N1);
...

void initUart(unsigned char uart, unsigned short div_factor, unsigned char mode){
  switch (uart){
  ...
    case UART2:
    //enable UART pins in GPIO (P0.10 = TxD0, P0.11 = RxD0)
    PINSEL0 |= ((1 << 20) | (1 << 22));

    //set the bit rate = set uart clock (pclk) divisionfactor
    U2LCR = 0x80; //enable divisor latches (DLAB bit set, bit 7)
    U2DLL = (unsigned char) div_factor; //write division factor LSB
    U2DLM = (unsigned char) (div_factor >> 8); //write division factor MSB

```

```
18      //set transmissiion and fifo mode
19      U2LCR = (mode & ~0x80); //DLAB bit (bit 7) must be reset
20      break;
21    ...
22    }
23  }
```

UART write

Put a character in the transmit buffer and wait for it to end transmission before returning.

```
1  int sendchar(unsigned char uart, int ch){
2    int count = 0;
3    switch (uart){
4    ...
5      case UART2:
6      if(ch == '\n'){
7        while(!(U2LSR & 0x20));
8        U2THR = '\r';
9      }
10     while(!(U2LSR & 0x20));
11     return (U2THR = ch);
12     break;
13   ...
14   }
15 }
```

UART read

Read the buffer and return the value.

```
1  int getkey(unsigned char uart){
2    switch (uart){
3    ...
4      case UART2:
5      return (U2RBR);
6      break;
7    ...
8    }
9  }
```

UART main function

The main function in the small UART program checks UART2 (PLC module) and UART0 (connected to the PC). If a character is received from the PC, the character is sent to the power line. The power line has a default loop back function, so when a character is sent to the power line it is looped back to UART2. When something is received on UART2, it is sent to the PC, and printed.

```
1  ...
2    while(1){
3      if((U0LSR & 0x1) > 0){    // If UART0 buffer is not empty
4        c = getkey(UART0);    // Read the buffer
5        sendchar(UART2, c); // Send the character to UART2
6      }
7      if((U2LSR & 0x1) > 0){    // If UART2 buffer is not empty
8        c = getkey(UART2);    // Read the buffer
9        sendchar(UART0, c); // Send the character to UART0
10     }
11   }
12 ...
```

4.2.3 Verification

UART verification The UART code has been verified by connecting it to the power line line module, sending a character and receiving the same character again on the PC. Full documentation can be found in the *EPRO 3 & 4 PLC - Hardware Interface* document.

EMC verification Together with the Physical Interface code this driver has been verified. When the center button on the ARM board is pressed, the register containing information about the switches on the FPGA board is read, whereafter this value is sent to the register in the FPGA that updates the diodes next to the switches. A video have been uploaded to youtube showing the verification of both the EMC driver and the Physical interface driver in the FPGA[9]

4.3 Database - Paulo

Each module in the system have sensors which send data to be stored and shown to the user. As such, data have to be stored in a convenient and consistent way. A MySQL database is used. All data stored have to be present in a web interface the integration between this type of data base and a server-side scripting (PHP) keeps the development faster and completely open source. A database have to be consistent, flexible and efficient so no data is lost or repeated when a value is send to it.

4.3.1 Verification specification

A LAMP (Linux Apache MySQL PHP) web server is implemented at the development environment used to build the uClinux image. This use a linux distribution CENTOS virtual machine in which the web server was built (how to build the web server explanation further in this documentation). For an easy and fast verification and development of the database a phpMyAdmin web application is installed at the server. The verifications can be found at the end of this section.

4.3.2 Analysis

The analysis of the system information and requirements is fundamental to the development of a database.

Use Cases

- Jan is looking at the web interface for the energy hub. From where he can see the status for each modules connected to it in a graphically way.
- A new energy module is connected to the hub, Jan opens the web interface for the energy hub, he logs in the administrator section of the system to start, stop or see a more detailed overview of each module.
- Jan arrives at the university in the morning and an email was send to him reporting a failure in the green energy system, he login to the administrator web interface, and he can see what the problem might be, and if it's possible to solve it directly on the interface.

From this use cases given by the customer the relevant information for the database development is filtered:

[9] `http://www.youtube.com/watch?v=3MUK6qbgORk`

- The status of each module have to be show.
- A detailed overview of each module such as current production (Voltage, Current, Power, etc.), efficiency of the module, etc.
- Errors have to be reported to the system administrator / maintainer.

In a deeper analysis of the information system for the power energy hub, some technical data is needed to be stored such as:

- Each module have is unique id, this is similar to the MAC address, this way when a module is connected is possible to check if the module have been connected before, so the data can be stored for the same module instead of instantiate a new module which can generate ambiguous data.
- Three types of modules are defined: input, output and bidirectional. This are seen from the power hub point of view where inputs are producers (solar panels, wind turbine, ...), output are consumers (Inverter, 30V output socket) and bidirectional (C.A.E.S, batteries, ...).

4.3.3 Design

The design of the database as part of its development is fundamental for a overview of the system, this is done with the use of data sets and data models. With data sets a detailed description of each table in the

Data sets
From the system information analysis a basic tabular structure is designed for a better understanding and low level overview of the database, this is called data set structure.

TYPE(ID_TYPE, TYPE);

ID_TYPE	Auto increment integer, a new id number is generated when a different type is needed to the system.
TYPE	Describe the type name for example: Input, output, bidirectional, etc.

STATUS(ID_STATUS, STATUS);

ID_STATUS	Auto increment integer, a new id number is generated when a different status is needed to the system.
STATUS	Describe the status name for example: Running, stopped, warning, etc.

MODULES(ID_MODULE, UNIQUE_ID, ID_TYPE, ID_STATUS, NAME);

ID_MODULE	Auto increment integer, a new id number is generated every time an unknown module is connected.
UNIQUE_ID	Module unique ID, this id as primary key ensures that no module with is repeated in the database.
ID_TYPE	This is a foreigner key for the table type, this way if some other type of module is needed it can be dynamical add.
ID_STATUS	This is a foreigner key for the table status, this way if some other status to the module is needed it can be dynamical add.
NAME	The name of the module for example, Solar Panel, wind turbine, battery.

As a requirement the database have to store the measurement from different modules, the common and minimal measurements needed from the modules are current and voltage. So a log data set is created.

LOGS(ID_LOG, ID_MODULE, DATE_TIME, PORT, CURRENT, VOLTAGE);

ID_LOG	Auto increment integer, a new id number is generated every time a measurement is add.
ID_MODULE	This is a foreigner key for the table modules, this allow the system to know from which module correspond the measurement .
DATE_TIME	Date and time of the measurement is saved so it can be plotted or in case of a lower efficiency a detailed history can analysed.
PORT	Defines in which port or the energy hub the module is connected in the time of the measurement.
CURRENT	Current measurement.
VOLTAGE	Voltage measurement.

At this point the database can initialise a new module or identified if the module where connected before, change the status for each module and add new measurements for each module. (Verification 1, 2 and 3)

Measurements are add to the database constantly, which in a non-stop system database size could be a problem, to avoid this situation a wrapper data set is created. This will store an average of values between a time span for each module, allowing the customer either to erase the values from the log or export them out from the database. Crucial data can be lost in this step for example the port history in case of troubleshooting, and precise date and time which would not allow the precise plot of data.

AGGREGATE(ID_WRAP, ID_MODULE, DATE_FROM, DATE_TO, AVG_CURRENT, AVG_VOLTAGE);

ID_AGGREGATE	Auto increment integer, a new id number is generated every time an wrap is needed.
ID_MODULE	This is a foreigner key for the table modules, this allow the system to know from which module correspond the values .
DATE_FROM	Date and time of the time span start.
DATE_TO	Date and time of the time span end.
AVG_CURRENT	Average current measurements.
AVG_VOLTAGE	Average voltage measurements.

The system is now able to decrease the amount of space needed using a time span and average values. (

Verification 4)

During the verification process some problems related to the flexibility of the database were found:

- Problem 1: The Photovoltaic and Wind Turbine modules have more measurements to be stored then only current and voltage, for example velocity and direction of the wind, sun position, etc.

Problem 1 In the design of this database it was assumed that all the modules would only send to the database the current and voltage. But during the verification process, the designed database was not flexible enough by the type of measurements being restrict to only current and voltage.

- How to make the database more flexible so it would allow inserting different measurements for different modules?

Two different approaches were made:

- Option 1: Create a new table for each new module that have extra measurements and add the ID_EXTRA to the logs table.
- Option 2: Give different unique ids to the sensors and relate them to their modules, so the log will store the sensor measurement.

Option 1 is not consistent, when a new module is connected a new table have to be created for that module with the needed measurement units.

Option 2 is the most reliable and the implemented one. A table Units and Sensors were created:

SENSOR(ID_SENSOR, ID_MODULE, ID_UNITS);

ID_SENSOR	Auto increment integer, a new id number is generated when a different sensor is needed to the system.
ID_MODULE	This is a foreigner key for the table sensors, this allow the system to know from which module correspond the sensor, being a primary key with the id_sensor, this way each sensor correspond to only one module .
ID_UNITS	This is a foreigner key for the table units, this allow the system to know which units the sensor is measuring.

UNITS(ID_UNIT, UNIT);

ID_UNIT	Auto increment integer, a new id number is generated when a different unit is needed to the system.
UNIT	Describe the unit name for example: A, V, deg ,m/s,etc. (Ampere, Volt, Degrees, Velocity)

In table wrapper and logs the ID_MODULE was replaced with ID_SENSOR. The database becomes more flexible since at anytime a module with new an unknown sensor can be add. A new sensor can now be add to a existent module and the values will always be saved and retrieved to the user with the appropriate measurement units.

Data Model
Data model is a high-level overview of the database structure. At this stage data sets are translated to a logic structure with the relationship between tables.

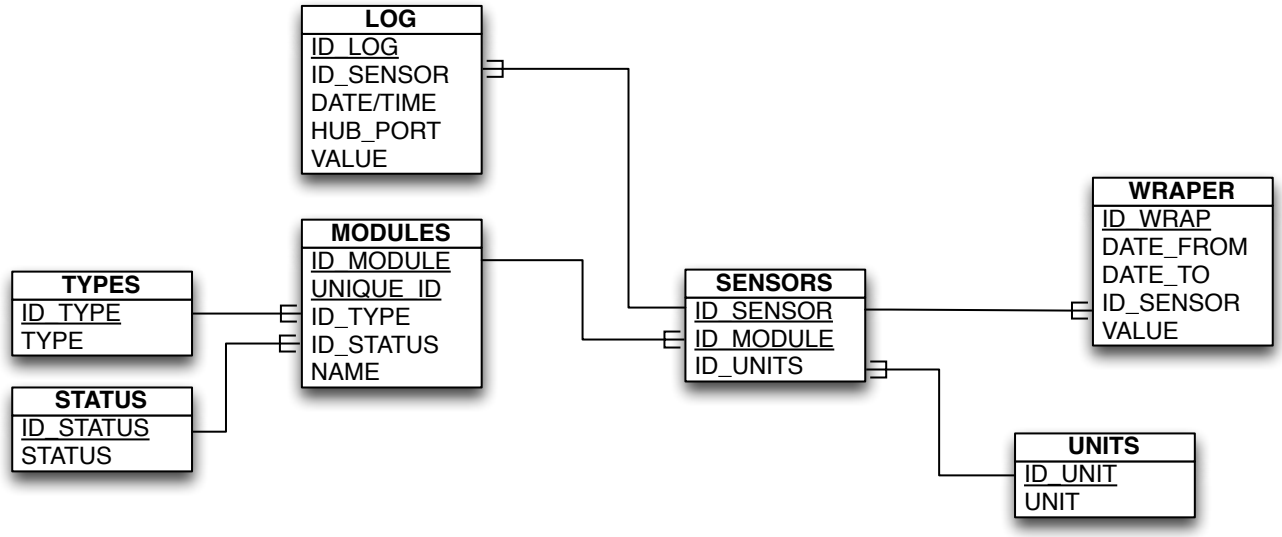


Figure 4.2: Data Model Structure

4.3.4 Implementation

Using the development environment virtual machine with the linux distribution CENTOS, a AMP (Apache MySQL PHP) web server is implemented, this way the web application phpMyAdmin is installed which simplifies the implementation of the database.

Webserver Implementation
A LAMP (Linux Apache MySQL PHP) web server is implemented, this is explains step by step how to implement this kind of web server on a linux CENTOS OS.

- Installing Apache and bring the server up.
  Install daemon that is going to handle HTTP requests on port 80. (Apache)

```
# yum install httpd
```

Start Apache in background.

```
# /etc/init.d/httpd start
```

Testing the web server. (Verification 5)

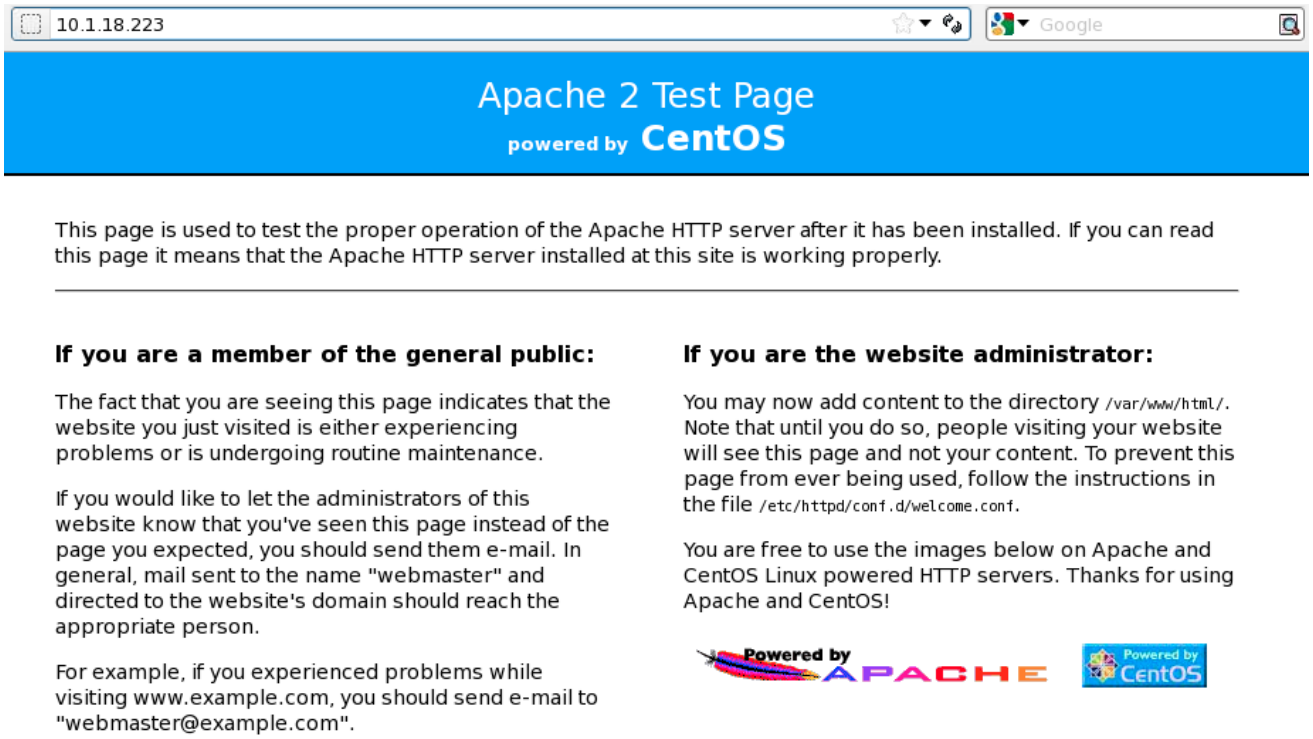


Figure 4.3: First page apache server running

- Installing MySQL and bringing the server up
  Install MySQL server and client.

```
# yum install mysql-server mysql
```

Start MySQL server in background.

```
# /etc/init.d/mysqld start
```

Testing the MySQL server (Verification 6)

```
# mysql
```

Server up and running when the follow response is given.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 5.0.95 Source distribution

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

At this moment the database is set with username root and an empty password. For security reasons a password is set for the MySQL server.

Changing MySQL root password

```
mysql> USE mysql;
mysql> UPDATE user SET Password=PASSWORD('new password') WHERE user='root';
mysql> FLUSH PRIVILEGES;
```

With the instruction 'USE mysql' the database directory mysql is open, 'UPDATE user SET Password=PASSWORD ('new password') WHERE user='root';' the password for the user 'root' is updated to the desired one. 'FLUSH PRIVILEGES' erases the privileges in cache and reloads the privileges in the MySQL server.

- Installing PHP
  Install PHP scripting language and necessary modules for MySQL server.

```
# yum install php php-mysql
```

php: scripting language compiler
php-mysql: mysql functions to be called from a PHP script

Testing PHP by creating a file in the web server documents root. /var/www/html/test.php. (Verification 7)

```
//test.php
<?php
 // Shows the in use PHP configuration file.
 phpinfo();
?>
```

Accessing in the web browser to the web server ip address http://10.1.18.223/test.php

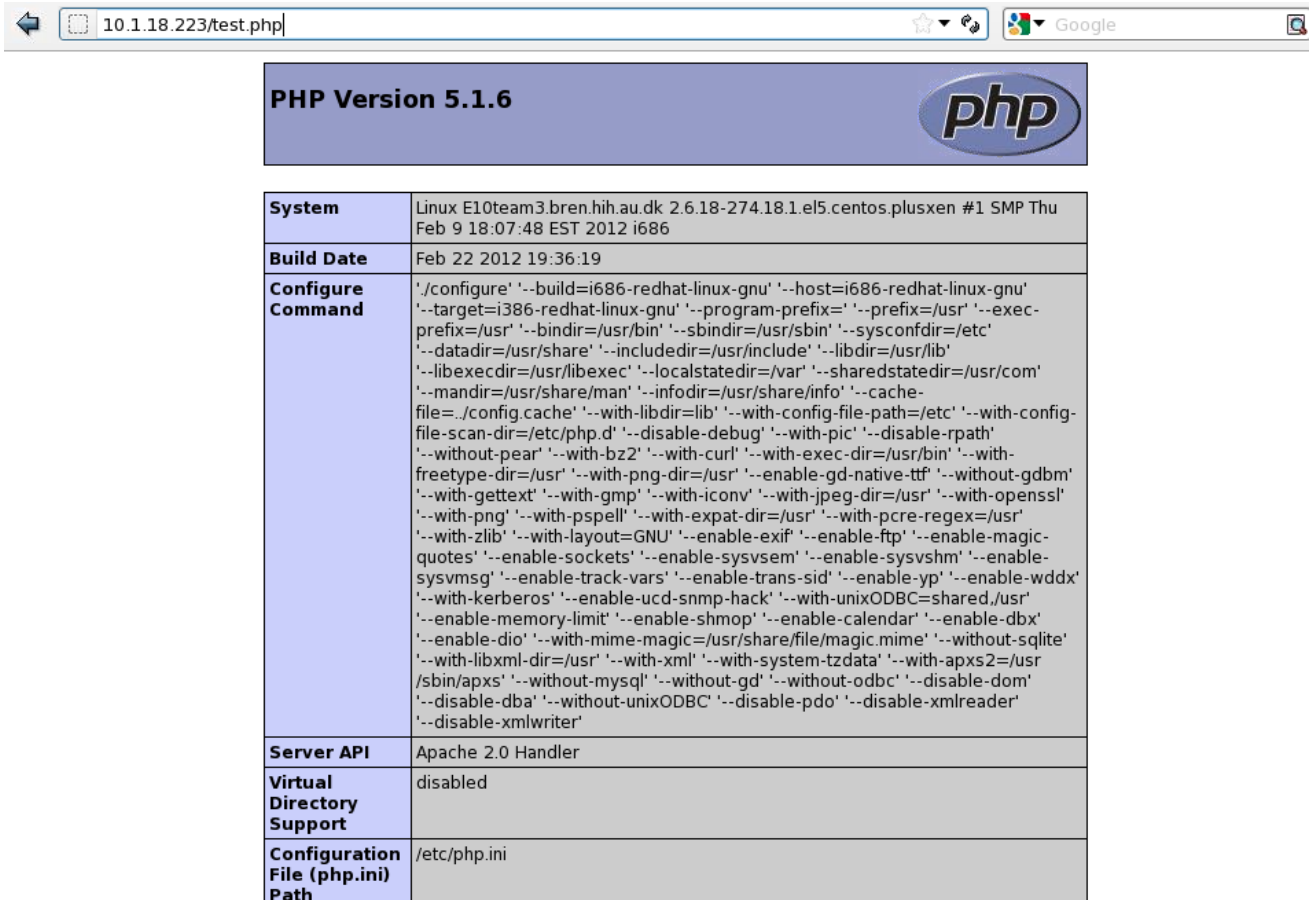


Figure 4.4: test.php - PHP configuration file

- Installing phpMyAdmin
  phpMyAdmin is a free web based MySQL database administration environment, without this tool the development of a MySQL database have to be made using the command line, it has become a fundamental tool for most web masters.

```
# yum install phpmyadmin
```

The Apache have to be restarted so it can assume the symbolic link to phpMyAdmin tool.

```
# /etc/init.d/httpd restart
```

Testing phpMyAdmin by pointing in the web browser to the address http://10.1.18.223/phpmyadmin. (Verification 8)

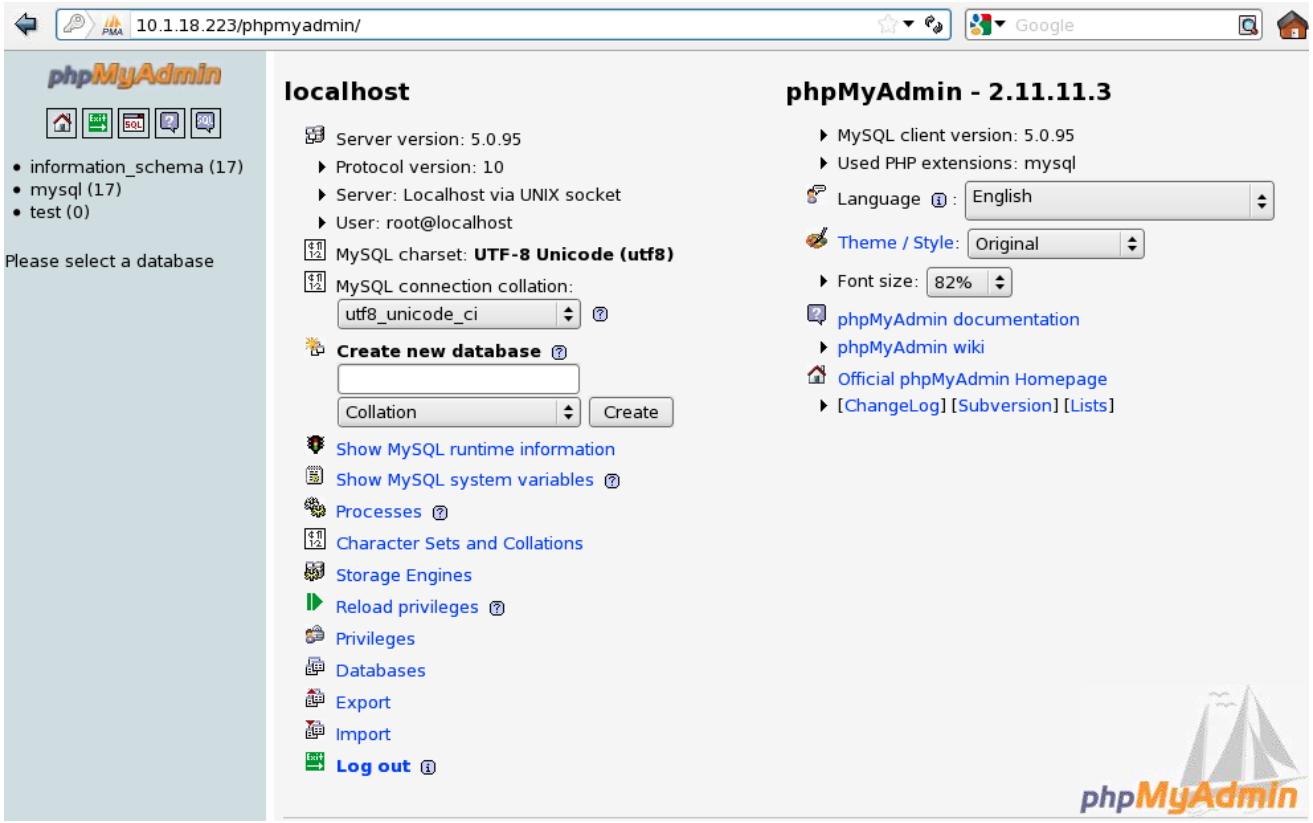


Figure 4.5: phpMyAdmin front page

Database Implementation

The final step of the database development is the physical implementation. The tool phpMyAdmin is used to implement the database at the MySQL server the code blocks generated by this tools are described bellow. The following blocks of code create the database and tables in it.

Database

To create tables that can be used to store data, a database directory have to be created.

```
CREATE DATABASE `powersystem`;
```

With the database directory created, the tables can now be created inside this directory.

Tables

The SQL syntax is going to be explain for only one table since the other tables will follow the same syntax just using different names.
MODULES

```
CREATE TABLE `powersystem`.`MODULES` (
 `ID_MODULE` INT NOT NULL AUTO_INCREMENT ,
 `UNIQUE_ID` INT NOT NULL ,
 `ID_TYPE` TINYINT NOT NULL ,
 `ID_STATUS` TINYINT NOT NULL ,
```

```
    `NAME` VARCHAR( 50 ) NOT NULL ,
    PRIMARY KEY ( `ID_MODULE` , `UNIQUE_ID` )
   )
```

This block of code will create the table MODULES in the database directory 'powersystem' with the columns ID_MODULE and UNIQUE_ID as primary key for this table. Primary keys are columns where one and only one value is allowed, in this table UNIQUE_ID is important to exist only once. ID_TYPE and ID_STATUS are foreigner keys form table TYPES and STATUS and NAME is the name of the module as explained in the analysis phase. TYPES

```
CREATE TABLE `powersystem`.`TYPES` (
`ID_TYPE` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`TYPES` VARCHAR( 50 ) NOT NULL
)
```

STATUS

```
CREATE TABLE `powersystem`.`STATUS` (
`ID_STATUS` INT NOT NULL  AUTO_INCREMENT,
`STATUS` VARCHAR( 50 ) NOT NULL ,
PRIMARY KEY ( `ID_STATUS` )
)
```

SENSORS

```
CREATE TABLE `powersystem`.`SENSORS` (
`ID_MODULE` INT NOT NULL AUTO_INCREMENT ,
`ID_SENSOR` INT NOT NULL ,
`ID_UNITS` INT NOT NULL ,
PRIMARY KEY ( `ID_MODULE` , `ID_SENSOR` )
)
```

UNITS

```
CREATE TABLE `powersystem`.`UNITS` (
`ID_UNIT` INT NOT NULL ,
`UNIT` VARCHAR( 50 ) NOT NULL ,
PRIMARY KEY ( `ID_UNIT` )
)
```

WRAPPER

```
CREATE TABLE `powersystem`.`WRAPPER` (
`ID_WRAP` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`DATE_FROM` DATETIME NOT NULL ,
`DATE_TO` DATETIME NOT NULL ,
`ID_SENSOR` INT NOT NULL ,
`VALUE` FLOAT NOT NULL
)
```

LOGS

```
CREATE TABLE `powersystem`.`LOGS` (
`ID_LOG` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`ID_SENSOR` INT NOT NULL ,
`DATE_TIME` DATETIME NOT NULL ,
`HUB_PORT` INT NOT NULL ,
`VALUE` FLOAT NOT NULL
)
```

Populate Tables

The tables are populated using phpMyAdmin since the easy implementation with this tool. All the SQL syntax generated by this tools is shown and explained bellow. Tables are populated in the beginning with initial values known values. PHP scripts will allow later on to add new data to this tables, allowing the system to be more flexible to unknown data.

STATUS: This SQL code block insert tree value into the table status in the powersystem database directory.

```
INSERT INTO `powersystem`.`STATUS` (`ID_STATUS` ,`STATUS` )
VALUES (NULL , 'Running'),
(NULL , 'Stoped'),
(NULL , 'Warning');
```

TYPES

```
INSERT INTO `powersystem`.`TYPES` (`ID_TYPE` ,`TYPES`)
VALUES (NULL , 'Input'),
(NULL , 'Output'),
(NULL , 'Bidirectional');
```

UNITS

```
INSERT INTO `powersystem`.`UNITS` (`ID_UNIT` ,`UNIT`)
VALUES (NULL , 'V'),
(NULL , 'A'),
(NULL , 'deg'),
(NULL , 'm/s'),
(NULL , 'km/h'),
(NULL , 'N'),
(NULL , 'P');
```

4.3.5 Verification

Verification	Description	Acceptance
1	Initialise a new module or identify if the module was connected before	TimeBox5
2	Change the status for each module	TimeBox5
3	Add new measurements for each module	TimeBox5
4	Decrease the database size using a time span and average values.	TimeBox5
5	Web server running and accessible	Ok
6	MySQL server running and accessible)	Ok
7	PHP script language testing	Ok
8	phpMyAdmin tool installed and running	Ok
9	Database created	Ok
10	Tables created	Ok
11	Tables populated	Ok

4.3.6 Database Structure

This documentation is generated by phpMyAdmin running on linux CENTOS server.

```
-- phpMyAdmin SQL
-- version 2.11.11.3
-- http://www.phpmyadmin.net
--
-- Host: localhost
-- Generation Time: Mar 29, 2012 at 04:11 PM
-- Server version: 5.0.95
-- PHP Version: 5.1.6
--
-- Database: `powersystem`
--
-- --------------------------------------------------------
--
-- Table structure for table `LOGS`
--
CREATE TABLE `LOGS` (
  `ID_LOG` int(11) NOT NULL auto_increment,
  `ID_SENSOR` int(11) NOT NULL,
  `DATE_TIME` datetime NOT NULL,
  `HUB_PORT` int(11) NOT NULL,
  `VALUE` float NOT NULL,
  PRIMARY KEY  (`ID_LOG`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
--
-- --------------------------------------------------------
--
-- Table structure for table `MODULES`
--
CREATE TABLE `MODULES` (
  `ID_MODULE` int(11) NOT NULL auto_increment,
  `UNIQUE_ID` int(11) NOT NULL,
  `ID_TYPE` tinyint(4) NOT NULL,
  `ID_STATUS` tinyint(4) NOT NULL,
  `NAME` varchar(50) NOT NULL,
  PRIMARY KEY  (`ID_MODULE`,`UNIQUE_ID`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
--
-- --------------------------------------------------------
--
-- Table structure for table `SENSORS`
--
CREATE TABLE `SENSORS` (
  `ID_MODULE` int(11) NOT NULL auto_increment,
  `ID_SENSOR` int(11) NOT NULL,
  `ID_UNITS` int(11) NOT NULL,
  PRIMARY KEY  (`ID_MODULE`,`ID_SENSOR`),
  KEY `ID_MODULE` (`ID_MODULE`)
```

```
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
--
-- --------------------------------------------------------
--
-- Table structure for table `STATUS`
--
CREATE TABLE `STATUS` (
  `ID_STATUS` int(11) NOT NULL auto_increment,
  `STATUS` varchar(50) NOT NULL,
  PRIMARY KEY  (`ID_STATUS`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;
--
--
INSERT INTO `STATUS` (`ID_STATUS`, `STATUS`) VALUES
(1, 'Running'),
(2, 'Stoped'),
(3, 'Warning');
--
-- --------------------------------------------------------
--
-- Table structure for table `TYPES`
--
CREATE TABLE `TYPES` (
  `ID_TYPE` int(11) NOT NULL auto_increment,
  `TYPES` varchar(50) NOT NULL,
  PRIMARY KEY  (`ID_TYPE`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;
--
--
INSERT INTO `TYPES` (`ID_TYPE`, `TYPES`) VALUES
(1, 'Input'),
(2, 'Output'),
(3, 'Bidirectional');
--
-- --------------------------------------------------------
--
-- Table structure for table `UNITS`
--
CREATE TABLE `UNITS` (
  `ID_UNIT` int(11) NOT NULL auto_increment,
  `UNIT` varchar(50) NOT NULL,
  PRIMARY KEY  (`ID_UNIT`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=8 ;
--
--
INSERT INTO `UNITS` (`ID_UNIT`, `UNIT`) VALUES
(1, 'V'),
(2, 'A'),
(3, 'deg'),
(4, 'm/s'),
(5, 'km/h'),
(6, 'N'),
(7, 'P');
-- --------------------------------------------------------
--
-- Table structure for table `WRAPPER`
--

CREATE TABLE `WRAPPER` (
  `ID_WRAP` int(11) NOT NULL auto_increment,
  `DATE_FROM` datetime NOT NULL,
  `DATE_TO` datetime NOT NULL,
  `ID_SENSOR` int(11) NOT NULL,
  `VALUE` float NOT NULL,
  PRIMARY KEY  (`ID_WRAP`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
--
--
```

4.4 Physical interface - Theis

The interface shall be made to fulfil the requirements *NF-1.5*[10] and *B-1.2*[11].

ID	Requirement	Description	Comments
NF-1.5	HW Interface	1 start button for the hub. 10 buttons to each start a module.	This is shown with 8 switches on the Spartan 6 board
B-1.2	Physical	1 diode showing when a module is off and one showing when it is on.	This is shown with 8 LEDs on the Spartan 6 board

4.4.1 Analysis

The physical interface is where the user interact directly with the hub, it is placed on the hub cabinet. It shall be possible for the user to start the hub and modules from here. There shall also be an indication of the hub and module status.
For turning on a module there shall be a switch for each module, and one for turning on the hub. To indicate the status there shall be a LED for every module and one to indicate the hub status.
The interface is made on the Spartan6 with an internal wishbone interface, and it is communicating with the ARM7 through the external memory controller in the ARM7.

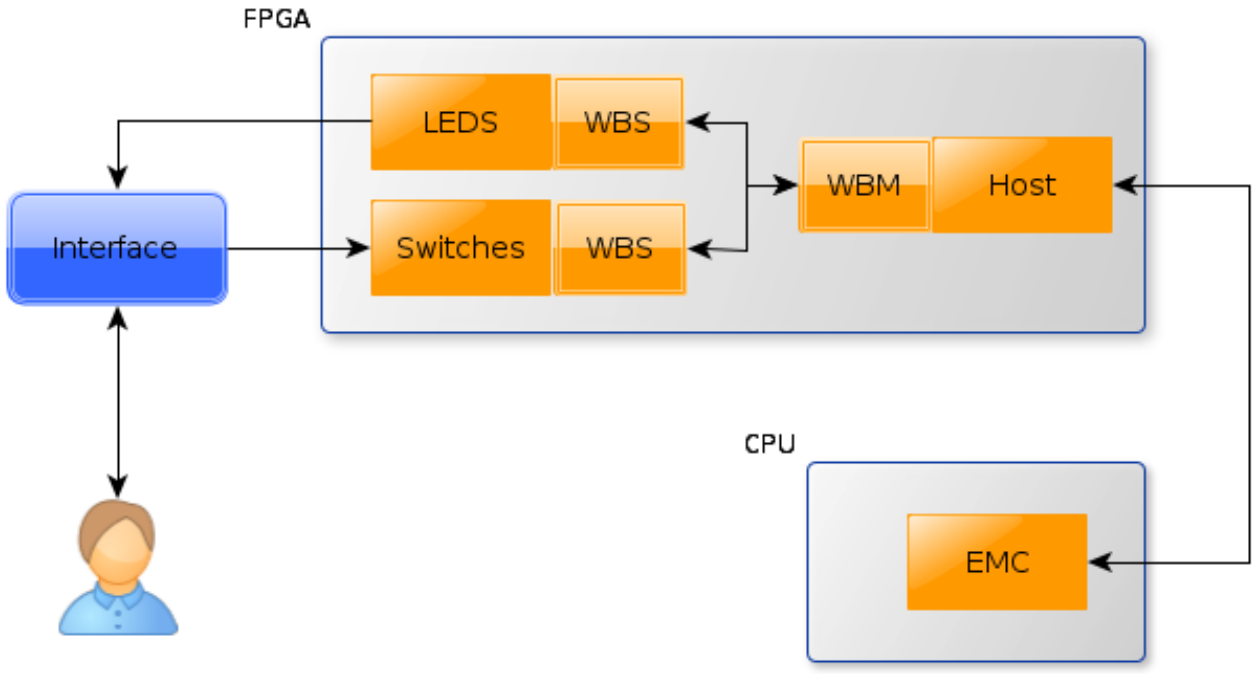


Figure 4.6: User-system interface interaction

[10]Requirement is found in table 3.2 in EPRO 3 project energy-hub
[11]Requirement is found in table 3.4 in EPRO 3 project energy-hub

4.4.2 Design

The physical interface is made with 8 switches and 8 LEDs, and is controlled through the Spartan 6. The block diagram below shows how the interface is made inside the Spartan 6, with the wishbone interface to the master, the master wishbone is discussed in Time box 3 section 3.5.3.

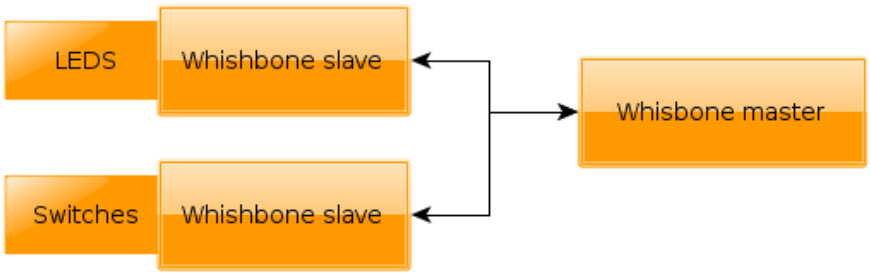


Figure 4.7: Block diagram for the interface in the Spartan 6

4.4.3 Implementation

The implementation is made in the Spartan 6 board. There is 8 LEDs and 8 switches located on the development board, that is used for the interface. The interface is written in VHDL and loaded on to the Spartan 6 FPGA.

LED interface The LEDs is controlled with the LED block in the Spartan 6, and it is able to send and receive data with a wishbone interface. The data output is the status of the 8 LEDs, the data input is used to turn on and off the LEDs.

In the code below, the entity of the LED controller is shown, the wishbone interface is defined from line 5 to line 16, these lines is common for all wishbone slaves. In line 19 the output for the 8 LEDs is defined as a 8 bit vector.

```
1  ...
2  entity WBS_leds is
3    port  (
4       --! Wishbone Slave interface
5       --Input
6       clk_i      : in     std_logic;     --! Clock input from SYSCON
7       rst_i      : in     std_logic;     --! Reset for Wishbone interface
8       cyc_i      : in     std_logic;     --! cycle input, asserted when cycle is in progress
9       stb_i      : in     std_logic;     --! Strobe input, similar to Chip Select
10      we_i       : in     std_logic;     --! Write Enable: High=WR, Low=RD
11      dat_i      : in     wb_dat_typ;    --! data from host
12      adr_i      : in     wb_lad_typ;    --! Low address input
13      --Ouput
14      ack_o      : out    std_logic;     --! Slave acknowledge output, asserted after sucessful cycle
15      err_o      : out    std_logic;     --! Error output, abnormal cycle termination
16      rty_o      : out    std_logic;     --! Retry output, slave not ready
17      dat_o      : out    wb_dat_typ;    --! data to host
18      ---------------------------------
19      --! Output port
20      leds       : out    std_logic_vector(7 downto 0)
21      );
22 end WBS_leds;
23 ...
```

Here the slave is responding the master, the error and retry signal is not supported in this slave, they are assigned with zeros, the acknowledge bit is set high if the strobe and cycle is assigned by the master. The Q signal is assigned to the LEDs and the data output. If the master reads data, it will get the status of the LEDs.

```
...
  signal Q        : std_logic_vector(7 downto 0);

begin

--  Concurrent assignments
--  Wishbone cycle acknowledge
  err_o <= '0'; --error signal
  rty_o <= '0'; --retry signal
  ack_o <= stb_i and cyc_i;  --! asynchronous cycle termination is OK here.
--  Data
  dat_o(7 downto 0) <= Q;
  leds <= Q;
...
```

In the code below the writing to the LEDs is made. If the cycle, strobe and the write enable inputs is high, the master wants to write data to this slave, then the process checks the address that the slave is sending data to. To write data to the LEDs the address has to be five zeros, then the code will assign Q with the data input from the master.

```
...
--!  Processes                                          --
--! Wishbone write to Q register
  Reg : process(clk_i)
    begin
      if(clk_i'event and clk_i = '1') then
        if (rst_i = '1') then
          Q <= Revision_c(7 downto 0);          --! Revision readable at reset
        else
          if ((cyc_i and stb_i and we_i) = '1') then
            case adr_i is
              when A_WBO_REG1 =>
                Q <= dat_i(7 downto 0);
              when others =>
              --Ack_o <= '0';
            end case;

          else
            Q <= Q;
          end if;
        end if;
      end if;
  end process Reg;
...
```

Switch interface The switches is read from the switches block through wishbone interface in the Spartan 6. This block is read only, and the data output is the status of the 8 switches.

The wishbone part of the code i common to the entity code in the LED block. For the switches an input vector is made with 8 bit.

```
...
entity WBS_switches is
  port  (
      --! Wishbone Slave interface
      --Input
      clk_i     : in    std_logic;    --! Clock input from SYSCON
      rst_i     : in    std_logic;    --! Reset for Wishbone interface
      cyc_i     : in    std_logic;    --! cycle input, asserted when cycle is in progress
      stb_i     : in    std_logic;    --! Strobe input, similar to Chip Select
```

```
      we_i      : in    std_logic;    --! Write Enable: High=WR, Low=RD
      dat_i     : in    wb_dat_typ;   --! data from host
      adr_i     : in    wb_lad_typ;   --! Low address input
      --Ouput
      ack_o     : out   std_logic;    --! Slave acknowledge output, asserted after sucessful cycle
      err_o     : out   std_logic;    --! Error output, abnormal cycle termination
      rty_o     : out   std_logic;    --! Retry output, slave not ready
      dat_o     : out   wb_dat_typ;   --! data to host
      ------------------------------------------------
      --! Intput port
      sw        :in     std_logic_vector(7 downto 0)
      );
end WBS_switches;
...
```

The acknowledge bit for the master is set here with the high strobe and cycle input.

```
...
--  Concurrent assignments
--  Wishbone cycle acknowledge
  err_o <= '0'; --error signal
  rty_o <= '0'; --retry signal
  ack_o <= stb_i and cyc_i;  --! asynhronous cycle termination is OK here.
...
```

In the code below, the switches status is written to the master wishbone. The process write the switches status to the lowest 8 bit in the data string and fill the rest with zeros, if the cycle and strobe input are high and the write enable is low to indicate that the master wants to read data.

```
...
--!  Processes                                          --
  Reg : process(clk_i)
    begin
      if(clk_i'event and clk_i = '1') then
        if (rst_i = '1') then
          dat_o <= Revision_c;          --! Revision readable at reset
        else
          if ((cyc_i and stb_i and not we_i) = '1') then
            case adr_i is
              when A_WBO_REG1 =>
                dat_o(7 downto 0) <= sw;
                dat_o(15 downto 8)  <= (others => '0');
              when others =>
              --Ack_o <= '0';
            end case;

          else
            dat_o <= dat_o;
          end if;
        end if;
      end if;
  end process Reg;
...
```

The rest of the code is made by Morten Opprud Jakobsen, and is not in the scope of this section.

4.4.4 Verification

The verification is made on the computer by simulating a test bench for writing to the LEDs and reading from the switches, and a test program on the ARM7 is made, to read the switches and write the status to the LEDs. The test with the ARM7 was recorded and is available through this link `http://www.youtube.com/watch?v=3MUK6qbg0Rk`

Read switches In the start of the test, *reset* is active and the switch status is set. before a read cycle is assigned the *address* is assigned, then the *chip select* goes low, then *read (output enable)*. The test shows that the data is tri-stated until the *read* goes low, and one clock cycle after, the status on *switch* is loaded into *data*.

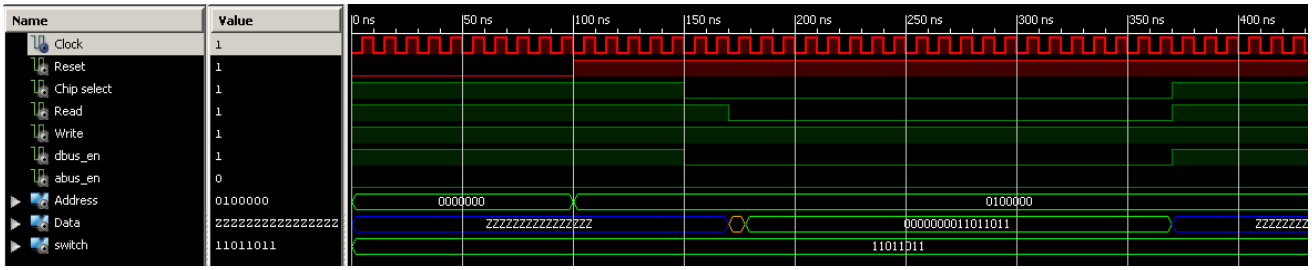


Figure 4.8: Test bench simulation of read

Write to LEDs In the write test, *leds* is uninitialized and *reset* is active, before a write cycle is assigned the *address* and *data* is assigned, then the *chip select* goes low, then *write (write enable)*. The test show that one clock cycle after *write* goes low the *data* is loaded onto the *Leds*.

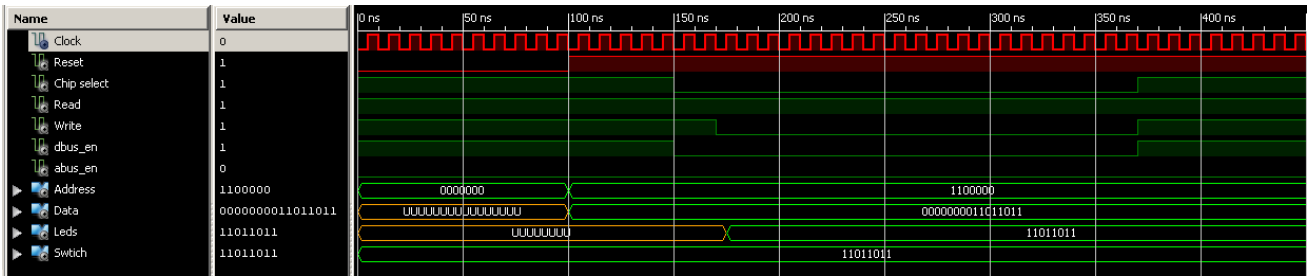


Figure 4.9: Test bench simulation of write

4.4.5 Conclusion

The test verify that the implementation of the interface design is working properly. The switches block is not completely finished, right now to register changes in the switches the ARM7 shall read the status to often, to prevent this an interrupt function is to be implemented, in that way the ARM7 is only reading the status when the switches changes. The interrupt function is not on the scope of this time box.

4.5 Deployment

Database Database is still being tested, and with new functionalities being add to the project the database is being constantly tested to fit all the changes. In this primary version the database is able to handle new or existent modules, dynamically add sensors to each module and add new measurements from this sensors.

Physical interface The physical interface delivered for this time box is revision 0.001.5, this version can send data from switches and receive data to turn on and off the LEDs.

5 Time box 5

5.1 Time box planning

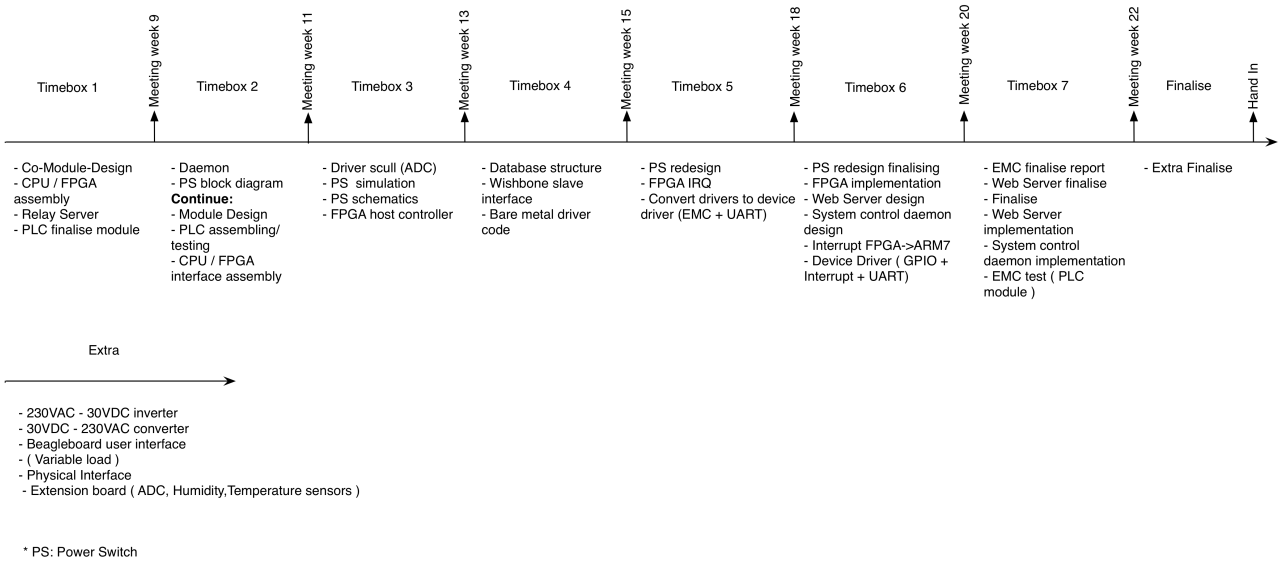


Figure 5.1: Updated time-box

5.1.1 Work to be done in this time box

- Power Switch module redesign
  - Improvements
  - Simulation
  - Redesign
- UART Device driver
  - Convert UART bare-metal driver into Linux device driver
- Interrupt register
  - Wishbone slave interface
  - Interrupt signals
  - Interrupt handling

Description:

UART Device driver Convert the bare-metal UART driver from the last time box into a device driver, in order to be able to access the UART hardware from user space in the uClinux environment.

Power Switch Main component of the energy hub system, controls the ports for each module allowing energy to flow in both directions.

Interrupt register is a block in the Spartan 6, that is allow to interrupt the ARM7 directly. The register is handling other interrupts from the Spartan 6 and tell the ARM7 where the interrupt came from.

5.1.2 Time planning

	UART Device driver	PS redesign	Interrupt Register	Platform setup
Estimation	8	17	15	0
Actual	22	20	25	11
Developer	Dennis	Paulo	Theis	Dennis

Table 5.1: Estimation and actual time used on the project

5.2 Power Switch redesign

The prototype made in time box 3 for the power switch module didn't work as expected. New analyses and contact with Linear Technologies revealed that the IC being used (LTC4357) is not the adequate device for this functionality.

Requirements

- Voltage input to the Energy hub shall be at 30V ±10%
- Ampere input to the Energy hub shall be max 30A.
- Power-line communication is used, for communication between the different devices and the hub.

Verification

- The system is able to switch the voltage as input or output. (Using a Multimeter on both POWERLINE and MODULE connection)
- Over voltage and under voltage are keep as set in common requirements 30V ±10% (Using a Multimeter on both POWERLINE and MODULE connection)
- Measurement of the current sensor for calculation of efficiency and security, max 30A. (Using a Multimeter on the current sensor output pin.)
- Create test-bench for 2 PCB switching system. (Fast prototyping using the mBed)
- Test Power Line Communication through the switching system, using development boards from Yamar and mBed.

5.2.1 Analysis

Power switch control
One of the problems found on the first prototype was that the LTC4357 device don't have a control pin to turn

the MOSFETs on/off, this device will work only as an ideal diode, making it a good application for photovoltaic harvesting method.

In contact with Linear Technologies a new device was suggested, the LTC4365, this device is a N-Channel MOSFET controller that protects the load using a higher and lower threshold voltage, along with a shut-down pin that drives the MOSFETs gate high or low.

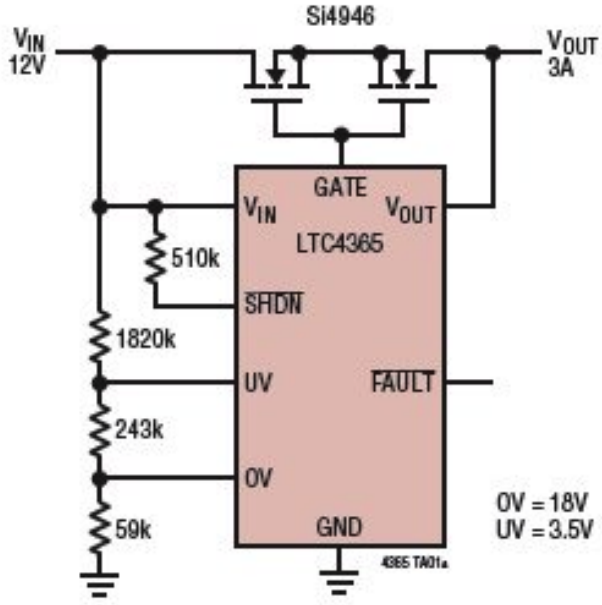


Figure 5.2: LTC4365 application notes.

In this application the shut-down pin is always high, energy will flow through the MOSFETs when the input voltage is between the under-voltage and over -voltage threshold selected values, in this case 3.5V and 18V. It's necessary to define the over and under-voltage threshold values for the power switch system, this is specified in the common requirements.

Common requirements:

$V_{in} = 30V$
$Offset = \pm 10\%$

From the datasheet:

$I_{UV} = 10nA$
$V_{OS} = 3mV$

V_{OSUV}: maximum tolerable offset (error)
I_{UV}: worst case scenario current leakage.

Under voltage requirement:

$U_V = 30V - (30 * 10\%) = 27V$

Over voltage requirement:

$O_V = 30V + (30 * 10\%) = 33V$

For working conditions of the device the bellow equation have to be true:

$R_3 + R_4 = \frac{V_{OSUV}}{I_{UV}}$

Using the formulas given in the datasheet to calculate the value of the 2 resistive dividers:

$R_2 = 2 * \frac{3mV}{10nA} * (27V - 0.5V) = 15.9M\Omega$ Approximate 16M Ω

$R_4 = \frac{\frac{V_{OSIU}}{I_{UV}} + R_3}{2 * O_V} = 246.97k\Omega$ Approximate 250k Ω

$R_3 = \frac{V_{OSUV}}{I_{UV}} - R_1 = 50k\Omega$

Simulations

Linear Technologies have a simulator for their devices, the LTSpiceIV, it's a free software that can be downloaded from the web site. Using LTSpice to draw a schematics, a test can be performed for the LTC4365 with the required resistor values to the power switch system.

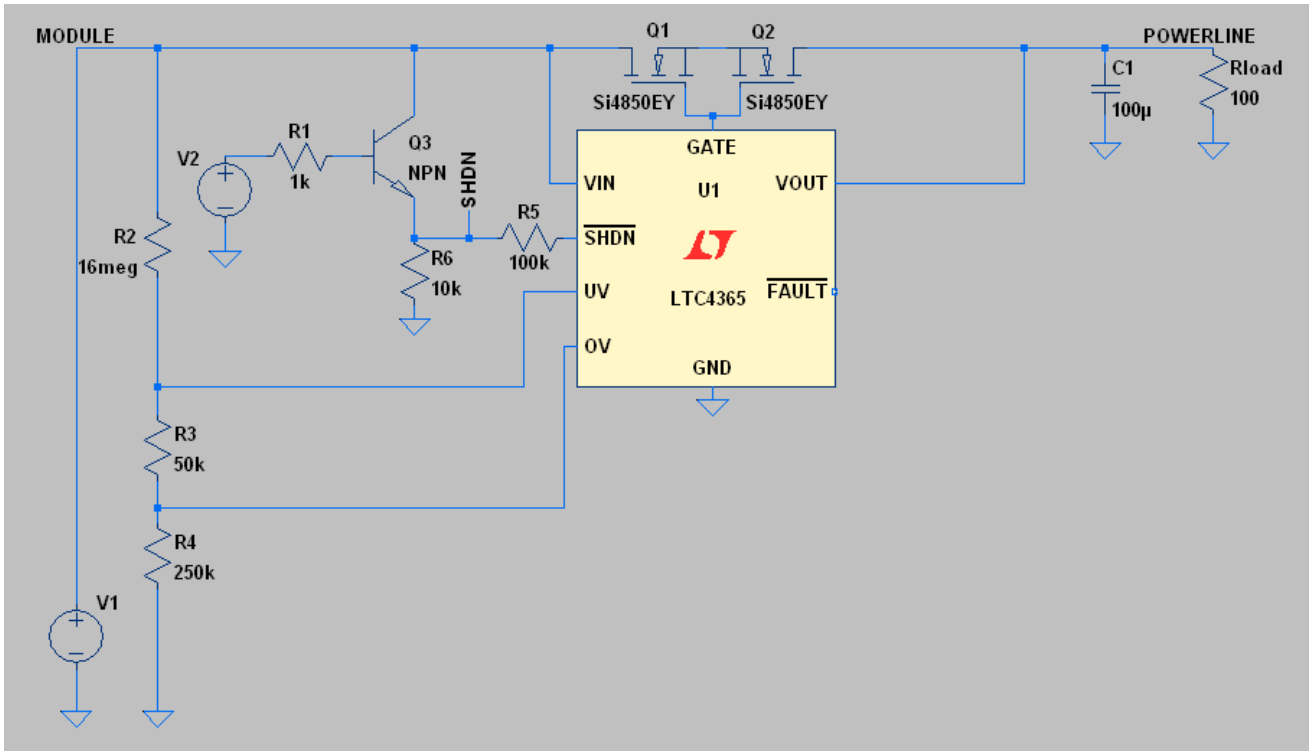


Figure 5.3: Schematics for the device LTC4365

A NPN transistor is add to drive the shut-down pin high and low. Two N-channel MOSFETs are connected

back to back to stop the voltage from passing to the load when the device is shut down.

Case Scenario 1 (Producer device connected to the MODULE side)

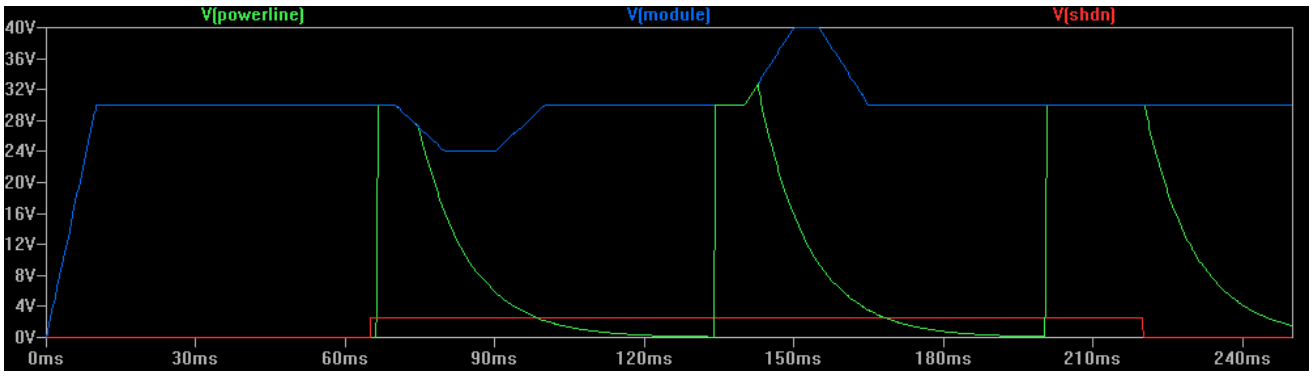


Figure 5.4: Simulation case scenario 1

In this case scenario the under voltage and over voltage can be tested along with the shut down pin.
The simulation has a total time of 250ms, the system takes 10ms to stabilized at 30V which will stabilize the device at that voltage.
At 65ms the NPN transistor is high, driven the input voltage to the shut down pin, with the shut down pin high the device will drive the gates for the two MOSFETs high and the current will flow through the MOSFETs from MODULE to POWERLINE as expected.
The voltage at the module at 80ms is changed to 24V, this will test the under voltage threshold value. From the calculations made in the analysis section, the device should sink the gates to ground when the voltage is under 27V, this happens with success at 75ms as it can be seen in the simulation.
To simulate test the over-voltage protection, the MODULE (source) is set to 30V so the device can stabilize again. At 140ms the voltage is pull up to 40V, which the device should sink the gates of the MOSFETs when the voltage is 33V, this happens with success at 142ms.
To test the control of the shut down pin, the voltage on the source side is set to 30V, getting the device to stabilize again. With the device in working conditions the NPN transistor is turned off at 220ms, the shut-down pin is then pull down by a 10kΩ resistor, pulling the MOSFETs gates to ground

Case Scenario 2 (Device off, no energy flow from MODULE to POWELINE and the other way around)
In the case scenario 1, there is no energy flowing when the device is off, is necessary now to simulate if the device is working as a diode, if no energy flows in the opposite direction. In this simulation the source and load positions are changed.

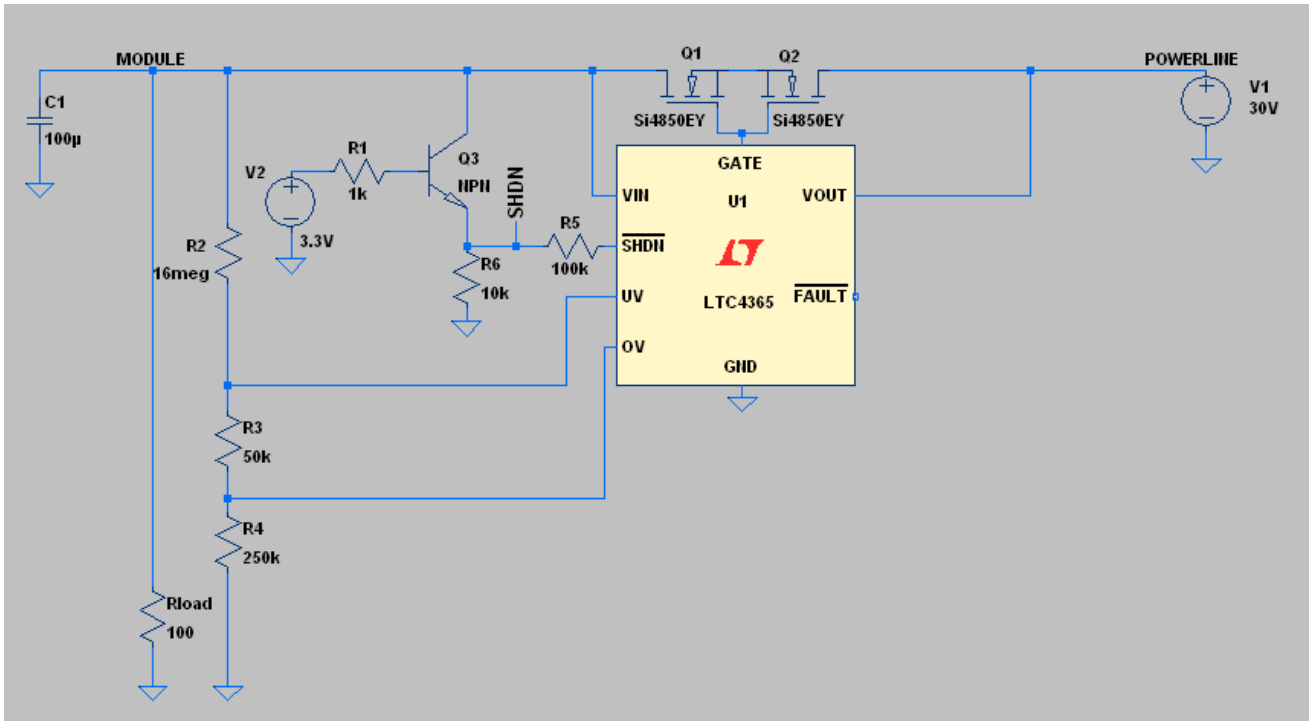


Figure 5.5: Schematics for the device LTC4365 with Case scenario 2 specifications

Being now the POWERLINE the source and the MODULE the load, this situation can occurs when a battery is fully charged and there is a different potential higher on the POWERLINE side. The device is going to insure that no energy flows in the opposite direction. This is successfully shown in the simulation bellow.

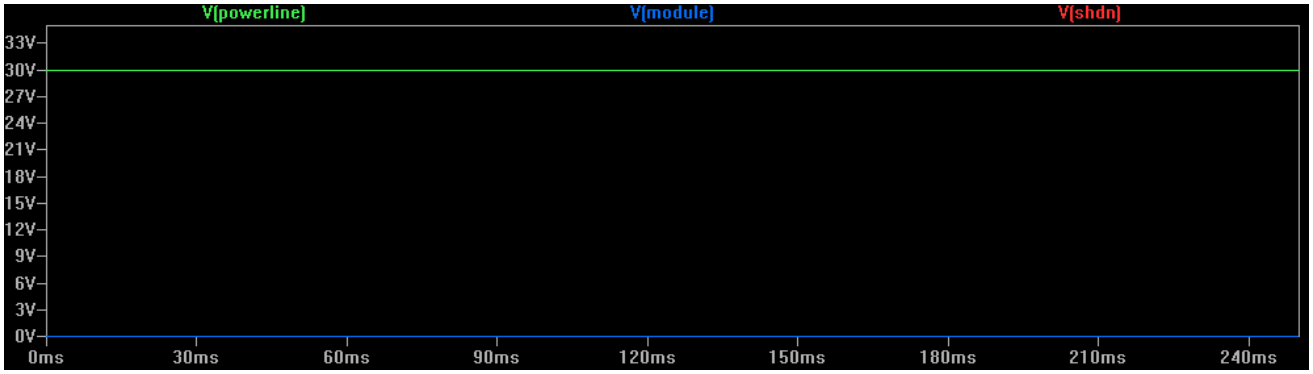


Figure 5.6: Simulation for case scenario 2

The voltage at the source (POWERLINE) is kept at 30V, and at the load (MODULE) the voltage is around 0V (1mV), in this configuration the circuit is behaving as expected, when the device is off, and the energy flow in the opposite direction the system behave as a diode, blocking the energy from flowing.

Current Sensors

Being part of a green energy system, a current sensor is implemented in this power switch system. The measure-

ment of current allows the system to calculate the efficiency of each module connected to the switch port. The current measurements are shown to the user on the web interface.
The sensor used is a hall effect linear current sensor from Allegro, the ACS756SCA-050B, a bidirectional hall effect sensor with range from -50A to +50A.

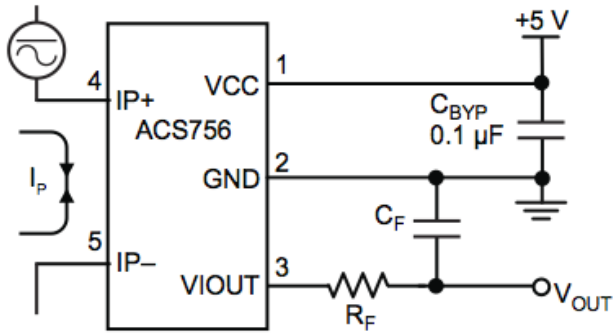
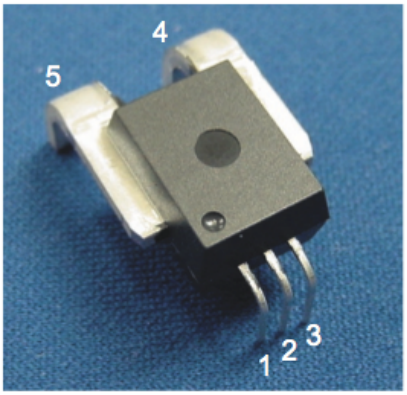


Figure 5.7: ACS756SCA Typical application

A test for this device is set on a bread board, the measurements are made using a voltmeter on the output pin. At a temperature around $24°C$ the output increments approximated 40mV for each ampere, this is consistent with the datasheet.
Since we can have a meaningful measure with the factory ranges, there is no need to amplify this signal.



Pin 1: V_{CC}
Pin 2: Gnd
Pin 3: Output
Terminal 4: I_{p+}
Terminal 5: I_{p-}

Figure 5.8: ACS756SCA-050 pins.

5.2.2 Design

The redesign of the power switch system required some more analysis since this is the main functionality of the energy hub.

Power Switch Overview
An overview of the system to be can be seen bellow.

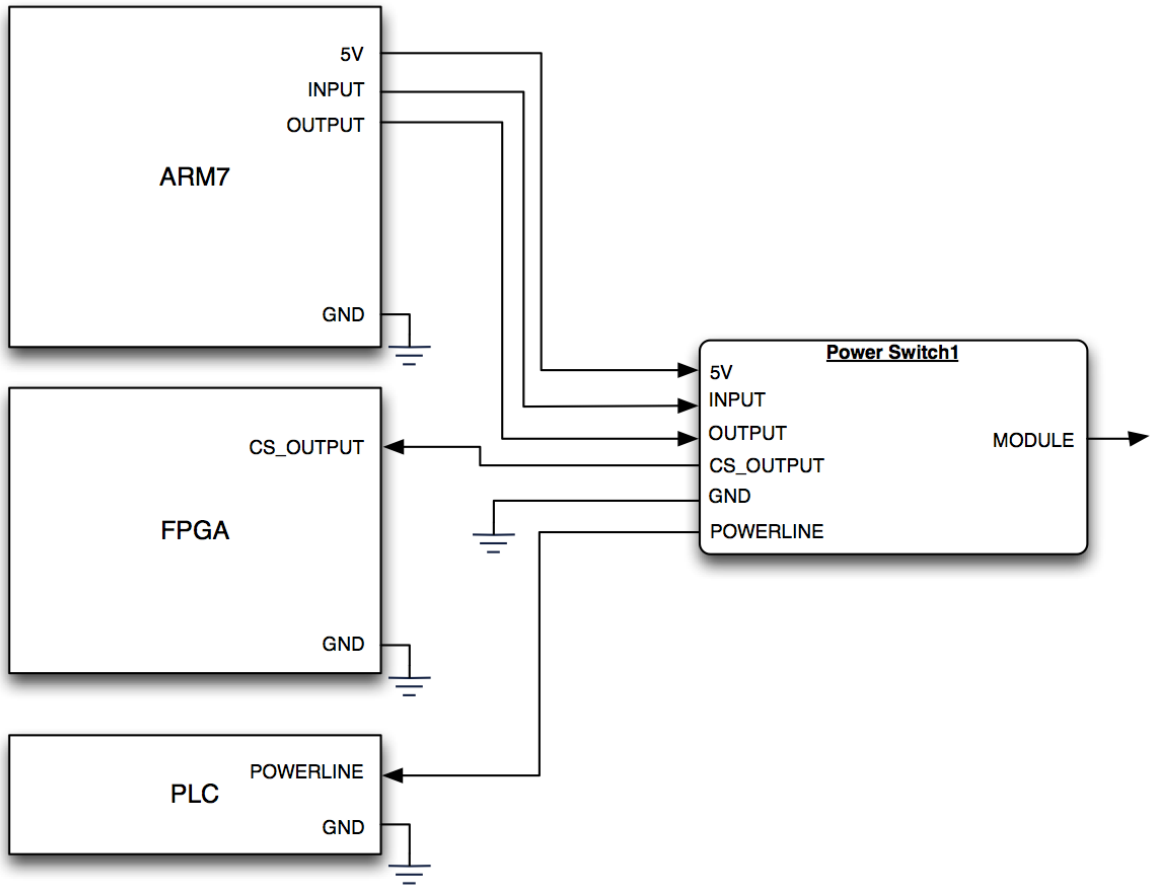


Figure 5.9: ACS756SCA-050 pins.

The ARM7 controls the input and output pins of the power switch system, a device driver will be implemented for this functionality later on this project. The CS_OUTPUT corresponds to the current sensor voltage output, with ranges from 0V(-50A) to 5V(50A), this is connected to one of the 8 ADCs implemented on the FPGA and retrieved to the core system (ARM7) through the EMC (External Memory Control) implementation.
POWERLINE pin from all power switch systems are connected to the same track into the PLC module. This will work as a sniffer for new communication arriving and to send new communication to the modules connected to the energy hub.
The MODULE pin is a connector for each module, in this prototype only 4 ports are built, since there's no more than 4 modules in development.

Schematics
Schematics of the power switch system.

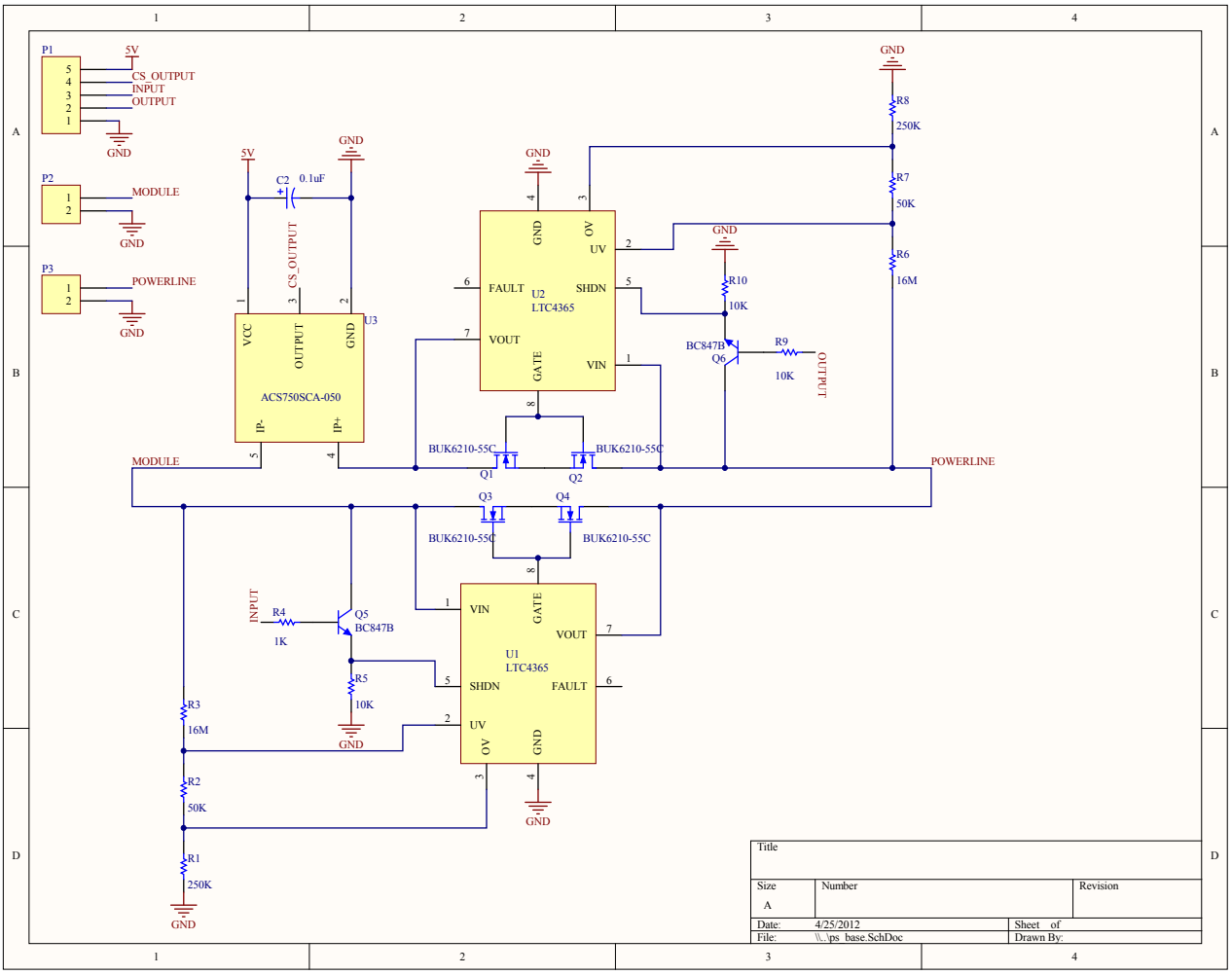


Figure 5.10: Power switch system schematics.

In this schematics the current sensor is add to the system a long with the new LTC4365 devices to control the two back to back N-channel MOSFETs. A NPN transistor is added between the V_{in} and the shut down pin, this will drive the pin to the input voltage or connect to ground.
In case off power failure the power switch systems are disconnected since the shut down pin is pull down to ground through a 10KΩ resistor.

PCB layout
A layout is made using Altium Designer software version 10.

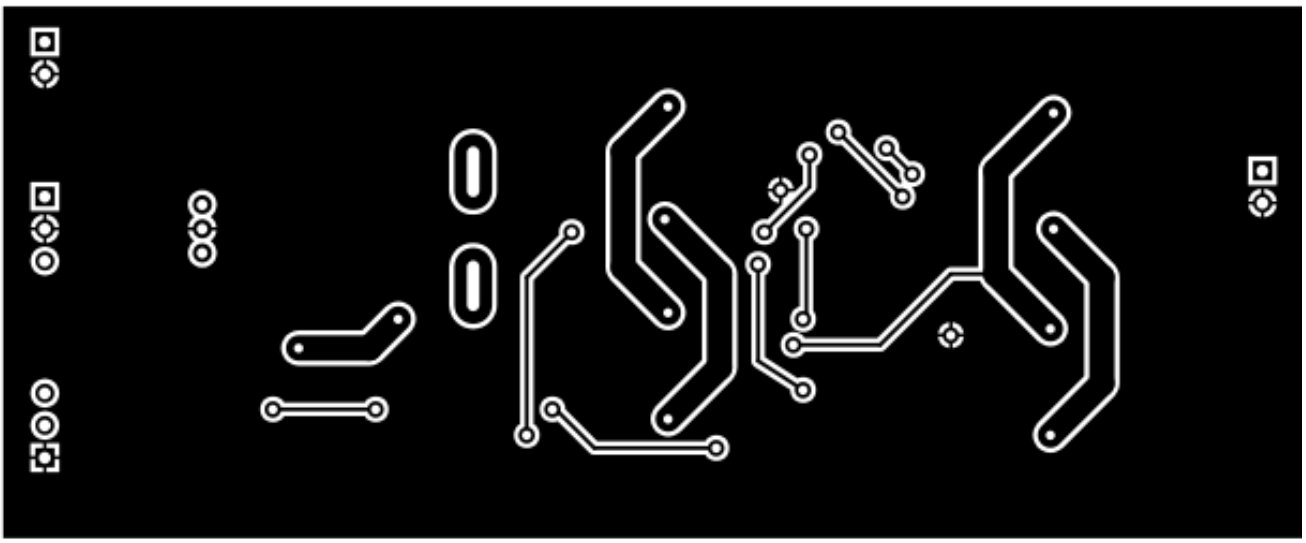


Figure 5.11: Power switch layout bottom layer.

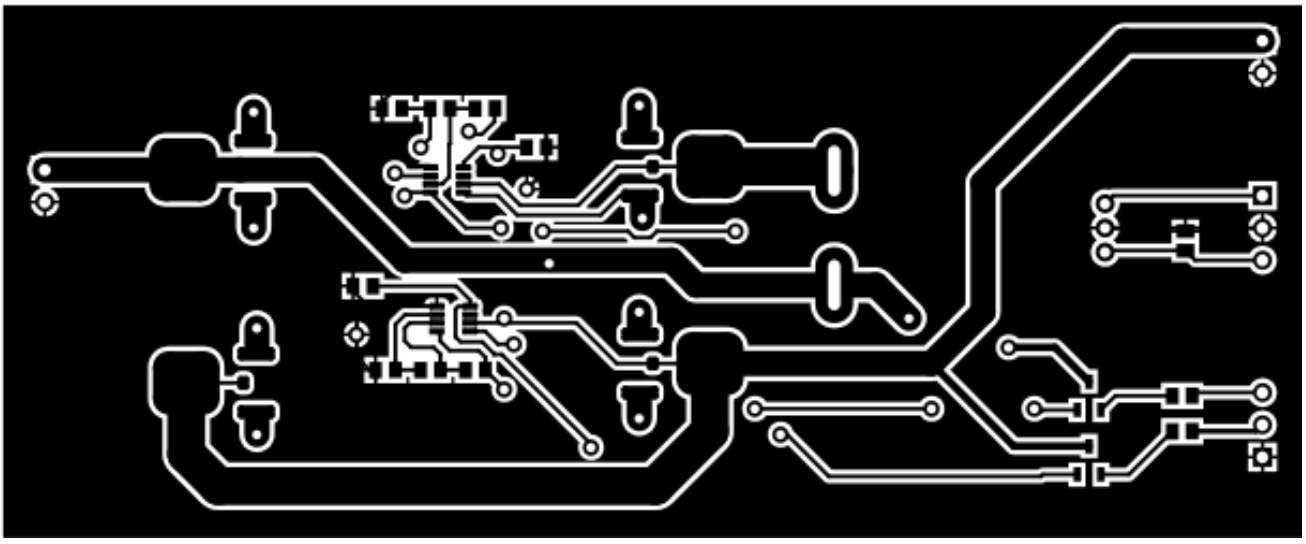


Figure 5.12: Power switch layout top layer.

This layout is a prototype for 1 power switch system, the tracks between the MOSFETs and the pin headers for the modules and power line are larger for eat reduction due to higher current flow.
The LTC4365 is as close as possible to the MOSFETs this is suggested in the device datasheet.
The board is going to be assembled and tested in time box 6.

5.2.3 Verification

All the verifications for the power switch system are changed to time box 7 since the PCB will not be ready for testing.

Verification	Description	Acceptance
1	Switch the voltage as input or output	TimeBox7
2	Over voltage and under voltage are kept 30V ±10%	TimeBox7
3	Current between maximum 30A	TimeBox7
4	Power Line Communication through the switching system	TimeBox7
5	2 PCB switching system	TimeBox7

5.3 Platform setup - Dennis

A virtual CentOS machine running on a machine at school has been assigned to each team in order to have a host machine where the boot loader (u-boot) can be compiled and modifications to the uClinux kernel can be made. Unfortunately the kernel compiled started to kernel panic. After undoing the different things done (removing files from the rc file and removing different kernel modules) the kernel kept panicking. Instead of using any more time on the machine, another similar virtual machine has been created on a local machine. The virtual machine is running CentOS 5.8 (like the one assigned), fortunately no kernel panic have been detected after compiling the kernel on the "new" local machine. Another benefit by having a local machine up and running is the flexibility as the Embedded Artist board should be connected to the schools network in order to access the tftp server where the board loads the kernel from throughout the development period. Now a tftp server has been set up on the local machine, where to the development board can be directly connected though an ethernet cable.

5.3.1 Network File system

In order to speed up the development process of user space applications and device drivers to the board, a NFS server has been setup on the local host machine using the OS X application *NFS Manager*. The NFS disk is then mounted on the board by issuing the command:
#mount -o nolock,rsize=4096,wsize=4096 -t nfs 192.168.1.1:/Users/Shared/nfs_mount /mnt/nfs,
which mounts the disk in */mnt/nfs*. When having the NFS server setup, there is no need to recompile the whole kernel when changes have been done to a kernel module or a user space application before it can be tested on the target.

5.4 UART Device driver - Dennis

As specified in the Technical platform section, the board used is LPC2478-32 from Embedded Artists. The on-board ARM processor is running uClinux, which is a Linux kernel for micro controllers without a memory management unit. In order to communicate with the UART hardware on the board a device driver has to be written, which is a piece of software that allows user-space applications in the kernel to interface with a hardware device (in this case the UART). The reason why implementing a UART driver, is to be able to communicate with the Power Line module which uses UART interface.

ID	Requirement	Description	Comments
F-1.2	Communication - System	The hub shall be able to communicate with a connected module through power line communication.	The interface to the power line module is UART.

5.4.1 Analysis

The UART used for communication to the Power line module is UART2, however instead of making a device driver especially for UART2, a more generic driver is build instead so other UART can be used in the future if

there is a need for it.

On the 32 bit LPC2478 processor the UART1 is not supported and UART0 is used as console, so during initialization it is very important not to change in any registers concerning these, to avoid kernel panic or loose communication with the console.

The initial framework for the driver was made in the last time box, which will be used as the skeleton together with the bare metal UART driver also made in the last time box.

5.4.2 Design

Before implementing the kernel module, it has to be thoroughly analyzed how to operate with the module from the user side of it. A small test program has been created in order to have some implementation guide lines, but also to have a verification script to all the time test against, to check if the module fulfills all the tests.

The script either opens the /dev/uart2 file and reads it, or opens it and writes the different commands available. After writing the file with one command, the content of the file is deleted (flushed) to prepare for a new operation.

```
#define UART "/dev/uart2"

int main(int argc, char *argv[]) {
        FILE *fp;
  //Write commands
  char buffer1[] = {"INIT:8N1"};
  char buffer2[] = {"BAUD:9600"};
  char buffer3[] = {"WR:55"};
  char buffer4[] = {"WR:0x55"};
  char buffer5[] = {"help"};
  //read buffer
  char read[10];
  //If read is not specified in the arguments do a write
  if(strncmp("read", argv[1], 4) != 0){ //Open the file
    if ((fp = fopen(UART,"wb"))==NULL){
                printf("Cannot open file.\n");
                exit(0);
          }
    //Initialize UART2 with 8N1 setup
    fwrite(buffer1, 1, sizeof(buffer1), fp);
    fflush(fp); //Empty the file
    //Define baudrate of devie
    fwrite(buffer2, 1, sizeof(buffer2), fp);
    fflush(fp);
    //Write a value decimal
    fwrite(buffer3, 1, sizeof(buffer3), fp);
    fflush(fp);
    //Write a value hexadecimal
    fwrite(buffer4, 1, sizeof(buffer4), fp);
    fflush(fp);
    //Write out different file options
    fwrite(buffer5, 1, sizeof(buffer5), fp);
          fclose(fp);
  }
  //Read the file
  else{
    if ((fp = fopen(UART,"rb"))==NULL){
                printf("Cannot open file.\n");
                exit(0);
          }
    //Read the file and print it if something available
    if(fread(read,1,10,fp) > 0) printf("\nVAL: %s\n",read);
    else printf("\nFail or nothing to read\n");

    fclose(fp);
  }
        return 0;
}
```

As it can be seen in the code, the user has the ability to:

- help - get a list of commands available in the kernel module.
- WR: - Write a character, which is transmitted on the UART module chosen. The value can be either hex or decimal.
- BAUD: - Change the baud rate of a UART to for instance 9600, 115200 or others.
- INIT: - Initialize the module with *Word length*, 5,6,7 or 8 bits, *stop bits*, 1 or 2 and *parity bit* on or off.

Note that the first time a device is opened after reset the INIT and BAUD shall be written to it in order to work as wanted.

5.4.3 Implementation

As the device has to be able to both read and write, the file descriptor shall at minimum look like below with a open, close, read and write function. Further implementations is discussed in the conclusion section.

```
static struct file_operations uart_fops = {
  .owner   = THIS_MODULE,
  .read  = uart_read,
  .write   = uart_write,
  .open    = uart_open,
  .release = uart_close,
};
module_init(uart_mod_init);
module_exit(uart_mod_exit);
```

The init and exit functions are similar to the one used in the ADC, except that the UART2 and UART3 power pins are disabled by default. If these pins are set and reset then the UART devices erases its settings. Therefore the power pins for UART2 and UART3 are set when inserting the module (UART0 and UART1 are enabled at reset).

```
m_reg_bfs(PCONP, 0x03000000);
```

The open and close functions are almost similar to the skeleton made, except that the pins for a module is set the first time a file is opened. When the file closes the pins are not reset to GPIO's as it should still be able to receive data and put it into its buffer.

```
static int uart_open(struct inode* inode, struct file* file){
  ---
  if(chRefCnt[num] == 0){
    file->private_data = (void *) num;
    if(num == 0 || num == 2 || num == 3){
      m_reg_bfs(PINSEL0, enable_pinsel[num]);
    }
    else if(num == 1){
      m_reg_bfs(PINSEL7, enable_pinsel[num]);
    }
  }
  chRefCnt[num]++;
}
```

As for now the read function is implemented as a dummy function which only returns a value when it is called. This implementation is similar to the one implemented in the bare-metal driver. This way of implementing a read call is very simple but also very inefficient. Further improvements of this is discussed in the conclusion section.

```
static ssize_t uart_read(struct file *p_file, char *p_buf, size_t count, loff_t *p_pos){
  ---
  if(endRead[num]){
    endRead[num] = 0;
    return 0;
  }

  if(!(m_reg_read(ulsr[num]) & 0x01)){
    return 0;
  }
  val = m_reg_read(urbr[num]);
  len = intToStr(val, p_buf+*p_pos, count, 10);
  if(len < count){
    p_buf[len++] = '\n';
  }
  endRead[num] = 1;

  return len;
}
```

When writing to the file, four different valid commands can be given. The *getCommand* function reads the string send and by comparing the string to a local array it returns a value to the write function, which handles the different commands given.

```
static int getCommand(const char __user* buf, size_t count, char** ppArg){
  int i = 0;
  char* pEnd = NULL;

  if(strncmp("HELP", buf, 4) == 0 || strncmp("help", buf, 4) == 0){
    return HELP;
  }

  pEnd = strchr(buf, ':');
  if(pEnd == NULL)
    return INVALID;
  *ppArg = pEnd+1;

  for(i = 0; i < sizeof(commands)/sizeof(char*); i++){
    if(strnicmp(commands[i], buf, pEnd-buf) == 0){
      return i;
    }
  }
  return INVALID;
}
```

When the command has been read by the *getCommand* function, the remaining length of the string is measured where after a function is called according to the return value of *getCommand*. The help function writes out the different options of writing the file to the user. The write command first checks if the user has send a decimal or a hexadecimal value and then reads the value. Then the *UART write busy flag* is checked to verify that the *write register* is empty. If the register is not empty a value is counted up and returns a *end of file* value (0) if the register is still not empty, otherwise it puts the value given into the *write register*. The baud rate setup and init functions are explained below.

```
static ssize_t uart_write(struct file *filp, const char *bufp, size_t count, loff_t *p_pos){
  ---
  cmd = getCommand(bufp, count, &arg);
  len = strlen(arg);

  switch(cmd){
  /******************************** HELP ******************************************/
  case HELP:
      help();
```

```
10        break;
11    /******************************** BAUD ***************************************/
12    case BAUD:
13        baud = strToInt(arg, len, base);
14        set_baud(num,baud);
15        break;
16    /******************************** INIT ***************************************/
17    case INIT:
18        wl = strToInt(arg,1,1); arg++;
19        pb = strToInt(arg,1,1); arg++;
20        sb = strToInt(arg,1,1);
21        init_uart(num,wl,sb,pb);
22        break;
23    /********************************  WR  ***************************************/
24    case WR:
25        if(strnicmp("0x", arg, 2) == 0){
26            arg += 2;
27            len -= 2;
28            base = 16;
29        }
30        val = strToInt(arg, len, base);
31        while(!(m_reg_read(ulsr[num]) & 0x20)){
32            ecnt++;
33            if(ecnt > 100)
34            return 0;
35        }
36        m_reg_write(uthr[num], val);
37        break;
38    default:
39        printk("warning: invalid argument'\n");
40    }
41    return count;
42 }
```

As written earlier, the init function has to be called the first time a file is opened after a reset (the same with the baud rate setup). The function enables the receive and transmit FIFO and sets up the: word length, amount of stop bits (1 or 2) and adds a parity bit if was requested.

```
1  void init_uart(int dev, int wl, int sb, int pb){
2    if(sb == 2) sb=1;
3    else sb = 0;
4    switch(dev){
5    ---
6    /********** uart2 **********/
7    case 2:
8        m_reg_write(U2LCR, 0x0);
9        m_reg_write(U2IER, 0x0);
10       m_reg_write(U2LCR, ( ((wl-5)<<0) |(sb<<2) | (pb<<3)  | (1<<7)));
11       m_reg_write(U2FCR, 0x07);
12   break;
13   ---
14   }
15 }
```

According to the lpc24xx data sheet[12] the UART clock must be 16 times the desired value of UART. From the drawing below it can be seen that the clock to the UART module is the same as the ARM7 clock (57.6MHz) as it is divided by one (these values are assigned to the PCLKSEL0 and PCLKSEL1 registers during low level initialization in u-boot).

[12] lpc24xx.user.manual.pdf, page 427

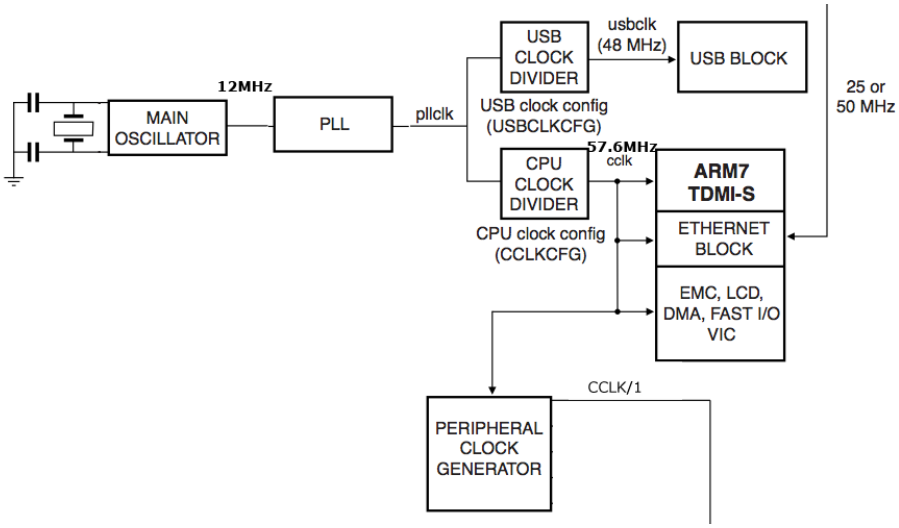


Figure 5.13: Peripheral clock to UART

In order to get the desired clock speed to the UART module, a scale value must be written to the *Divisor Latch Register*.
If a baud rate of 9600 is wanted, the value written to the register must be:
$57.600.000/(16 * 9600) = 375$
As seen in the code below, the divisor register is split into two parts (a low and a high part).

```
void set_baud(int dev, unsigned int baud){
  switch(dev){
  ---
    case 2:
      m_reg_write(U2DLL, (((LPC24xx_Fcclk/16)/baud) & 0xFF));
      m_reg_write(U2DLM, (((LPC24xx_Fcclk/16)/baud) >> 8));
    break;
  ---
  }
}
```

The other functions used: *intToStr*, *strToInt*, *chToUpper*, *help* is code reuse from some of the other device drivers provided with the kernel: SFR, ADC and PWM.

Inserting the UART module in the kernel To automatically insert the module in the kernel during startup, a few files shall be made. The first thing to do is to create the different uart files in dev and specify that the module shall be inserted to the kernel after startup. The different files created is specified in the rc file. In the kernel folder structure the rc file used is placed in: */home/emb/uClinux-dist/vendors/EmbeddedArtists/LPC2478OEM_Board*

```
---
/bin/mknod /dev/uart0 c 239 0
/bin/mknod /dev/uart1 c 239 1
/bin/mknod /dev/uart2 c 239 2
/bin/mknod /dev/uart3 c 239 3
---
if [ -f /drivers/adc.ko ]; then
  insmod /drivers/adc.ko
fi
---
```

In the *Makefile* in the same path a line shall be inserted to add the module to the *romfs image* which is extracted during boot time. Also in the top, the value 5 shall be exchanged with 6, to indicate that there is an extra char device.

```
---
DEVICES = console,c,6,1
---
$(ROMFSINST) -S drivers/2.6.x/uart/uart.ko /drivers/uart.ko & \
---
```

The last thing to do is to add uart to the makefile in the drivers/2.6.x folder, to be sure that the kernel remakes the UART module when compiled if any changed were done.

Adding the UART user space application When compiling a user space application through the kernel, there is no need to specify which gcc compiler to use or which flags to set in the different local Makefiles (this is all set in the big kernel Makefile). Therefore a makefile was found in the uClinux pdf book[13] in order to make small changes to the program, add it to the NFS folder and run it on the target, without the need for compiling the whole kernel and restarting the target each time. The makefile used:

```
CFLAGS= -Wall -W
LDFLAGS= -Wl,-elf2flt
CC= /usr/local/bin/arm-elf-gcc
RM=rm -f

PROG=uart
SRC= uart.c

OBJ=$(SRC:%.c=%.o)

$(PROG): $(OBJ)
  $(CC) $(CFLAGS) -o $(PROG) $(OBJ) $(LDFLAGS)

.PHONY: clean all dep

clean:
  $(RM) $(PROG) $(OBJ) *~ *.gdb .depend *.elf2flt *.elf
```

After finishing the application, it is added to the kernel by following the steps on Klaus wiki[14].

5.4.4 Verification

The module is only verified with the UART2 hardware. The verification of the module is done by the UART user space application. At first the rx and tx pins where short-circuit, to make a loopback. After performing a write to the file by running the UART user space application, the file was read and the values compared to verify that the data was not corrupt. Afterward a test has been made similar to the one performed on the bare-metal driver, where the Embedded Artist board and an MBED communicates with each other through their own power line modules, this test turned out successfully we where able to communicate between the ARM7 and the MBED through the PLC modules.

5.4.5 Conclusion

The UART hardware can be accessed through the device driver, however, the efficiency of the module is not that good. Therefore the driver still need some improvements which shall be further added in order to get a better

[13] Getting Started With uClinux Development A, page 72

[14] `http://klaus.ede.hih.au.dk/index.php/How-to_add_a_user_space_application_to_uClinux`

performance.

- Add interrupts to the receive part, to avoid reading the file once in a while to see if someone wrote anything.
- Add file write function, so a whole file can be compressed and sent to another module (wind turbine, photovoltaic, CAES or battery).
- Add more flexibility to initializing the module (break control, interrupt, ...)
- Add more error handling and better feedback description if an invalid command is received.

5.5 Interrupt register - Theis

The interrupt register is made to tune performance by limiting the number of read statements the ARM7 is doing, in order to read from the Spartan 6. The interrupt register is not by itself fulfilling any requirements, but it is tuning the following: *NF-1.5*[15] and *B-3*[16]

ID	Requirement	Description	Comments
NF-1.5	HW Interface	1 start button for the hub. 10 buttons to each start a module.	It is only reading if the buttons state changes
B-3	Errors	Humidity and Temperature sensor will be placed inside the system housing	The Spartan 6 interrupts if one of the following is happening
B-3.1	Humidity	If the humidity is above the maximum level 70%, the system shuts down	ARM7 only read on interrupt
B-3.2	Temperature High	If the temperature is higher than 55 degrees, the system shuts down	ARM7 only read on interrupt
B-3.3	Temperature Low	If the temperature is below 0 degrees, the system shuts down	ARM7 only read on interrupt

5.5.1 Analysis

The purpose of the interrupt register is to limiting the number of readings the ARM7 is making from the Spartan 6. There is one interrupt pin routed from the AMR7 to the Spartan 6, but there is two blocks in the Spartan 6 that makes data for the ARM7. These blocks are the switch input block and the analog to digital converter[17] block. One way to use the interrupt is to allow the switch block to interrupt when a user interact with the system, and then read the ADC with a specific time interval. But to avoid to many unnecessary readings of the ADC, the interrupt register is the only block that is allow to interrupt the ARM7, the interrupt register gets interrupt signals from the ADC and the switch blocks, the register is arranging the interrupt, and when the ARM7 is interrupted, it reads data from the register to find out which block it should read.

[15] Requirement is found in table 3.2 in EPRO 3 project energy-hub
[16] Requirement is found in table 3.4 in EPRO 3 project energy-hub
[17] ADC

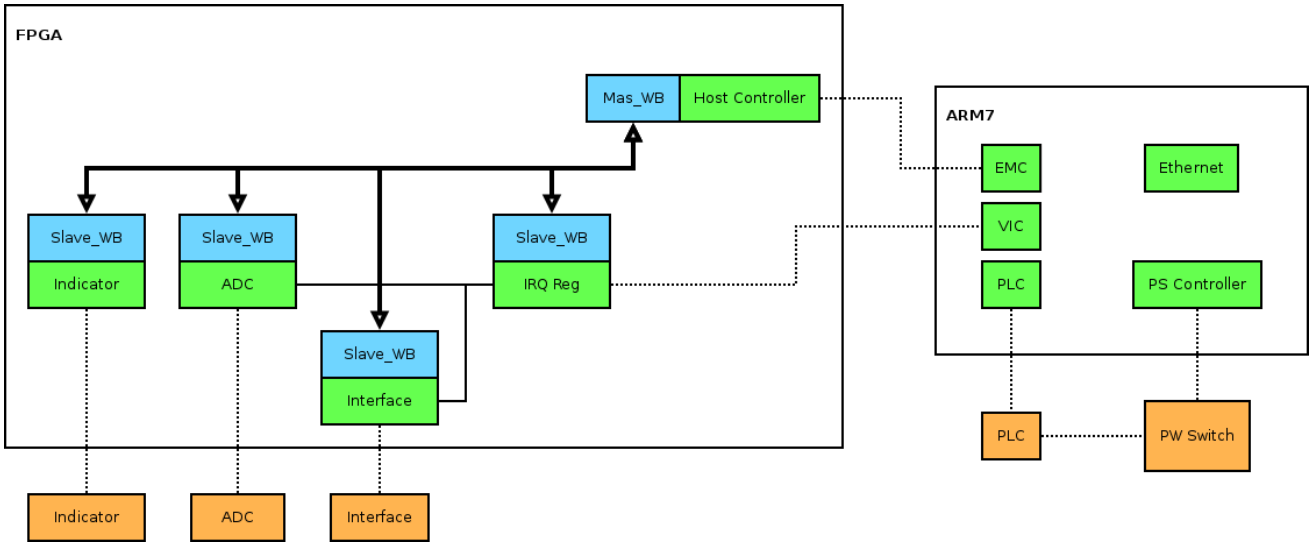


Figure 5.14: Module design with IRQ register

When a block sends an interrupt to the IRQ[18] register, it locates the interrupt and put the address to the block on to a FIFO[19], and sends an interrupt to the ARM7, then the AMR7 read the FIFO to get the address on the interrupting block, in that way the ARM7 knows where to read next.

5.5.2 Design

The purpose of the interrupt register is to make interrupt functionality on several block with only one interrupt pin to the ARM7. The interrupt register shall be optimized to wishbone use. The interrupt register without wishbone has the inputs, a vector of size N[20] and N AddrRange[21] bit wide vectors for the interrupt data. The output is the IRQ pin to the ARM7 and a data vector equal to the wishbone data width. This is shown in the figure below.

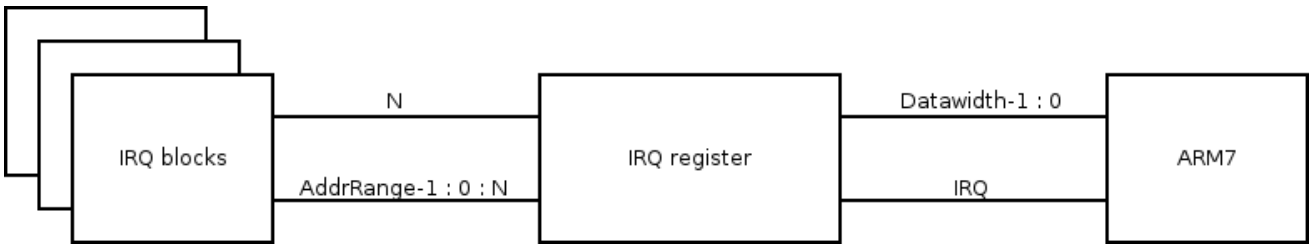


Figure 5.15: IRQ register without wishbone

When a block needs to make an interrupt, it puts the wishbone address onto the input array to the IRQ register, and then sets the interrupt output high to tell the IRQ register that there is a valid address that needs to be read. The IRQ register save the address and interrupt the ARM7. When an interrupt occurs on the ARM7, it read the IRQ register to get the address that it shall read next, in this way it takes two wishbone read cycles for

[18] Interrupt Request
[19] First In First Out
[20] N = number of blocks that is allow to interrupt
[21] Wishbone address range

the ARM7 to get the right data, but the ARM7 is only reading if the Spartan 6 have valid data to read, instead of make reading with specific interval. The IRQ register save the interrupt in a FIFO in case interrupt is made faster than the ARM7 can read.

Interrupt output The interrupt output is set low as long a the FIFO is not empty, to tell the ARM7 to keep on reading till the FIFO is empty and sets the IRQ high again.

Data output The data output is equal to the wishbone data width, in this case it is 16 bit, and the address is 7 bit. The seven lowest data bits is used for the address to the block that sends the interrupt, the rest of the bits can be used for extra data from the IRQ register.

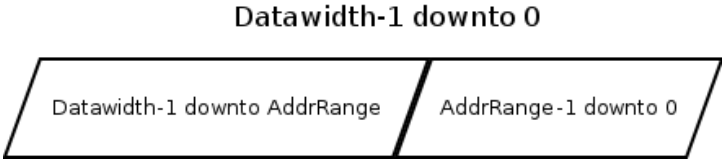


Figure 5.16: IRQ register data output

5.5.3 Implementation

A block diagram for the interrupt register is shown below. It consist of three block for handling the interrupt, a input block which handled the input and write data to the FIFO. The FIFO act like memory for the interrupt data, and sends a interrupt to the ARM7. The output block takes input from the wishbone and handle the data to the wishbone, form the FIFO.

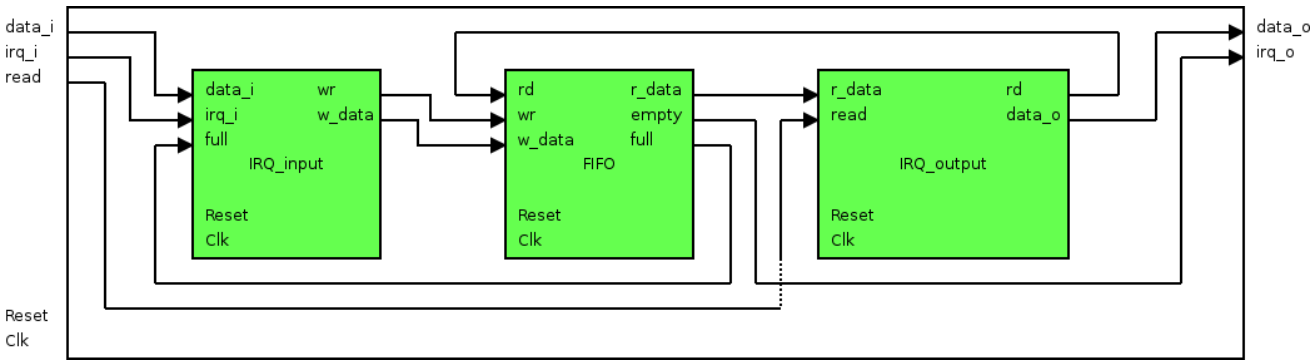


Figure 5.17: Block diagram for the IRQ register

FIFO - First In First Out The FIFO buffer is where the interrupt data is stored before they are read. The input block write data into it, and the output block read data out of it. The FIFO is necessary if the Spartan6 make interrupt faster than the ARM7 can read them. The figure below shows how the First In First Out principle is working.

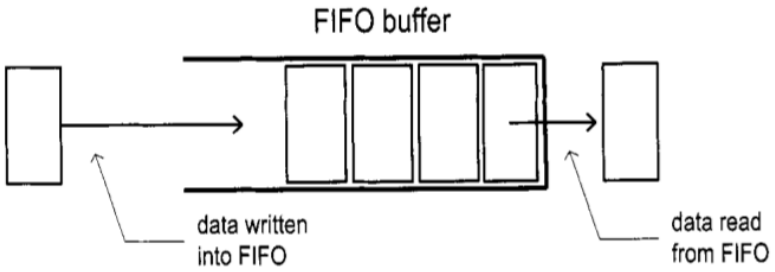


Figure 5.18: Conceptual diagram of a FIFO buffer

The FIFO buffer code is from the book *FPGA Prototyping by VHDL Examples - Listing 4.20*. It is a standard FIFO buffer, with a data vector input, an input for read and one for write, plus clock and reset. As output there is a data vector equal to the input vector, further more there is an empty and a full signal to indicate if the buffer is empty or full. A block diagram for the FIFO buffer is shown below.

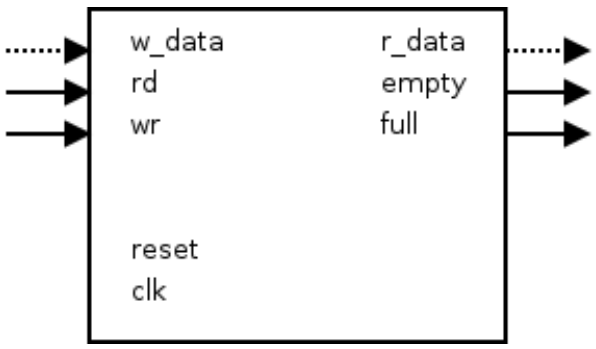


Figure 5.19: State diagram for input

The FIFO buffer is used to hold the interrupt data from the other blocks in order to present it for the ARM7 in the right order. A digram of how it works is shown below.

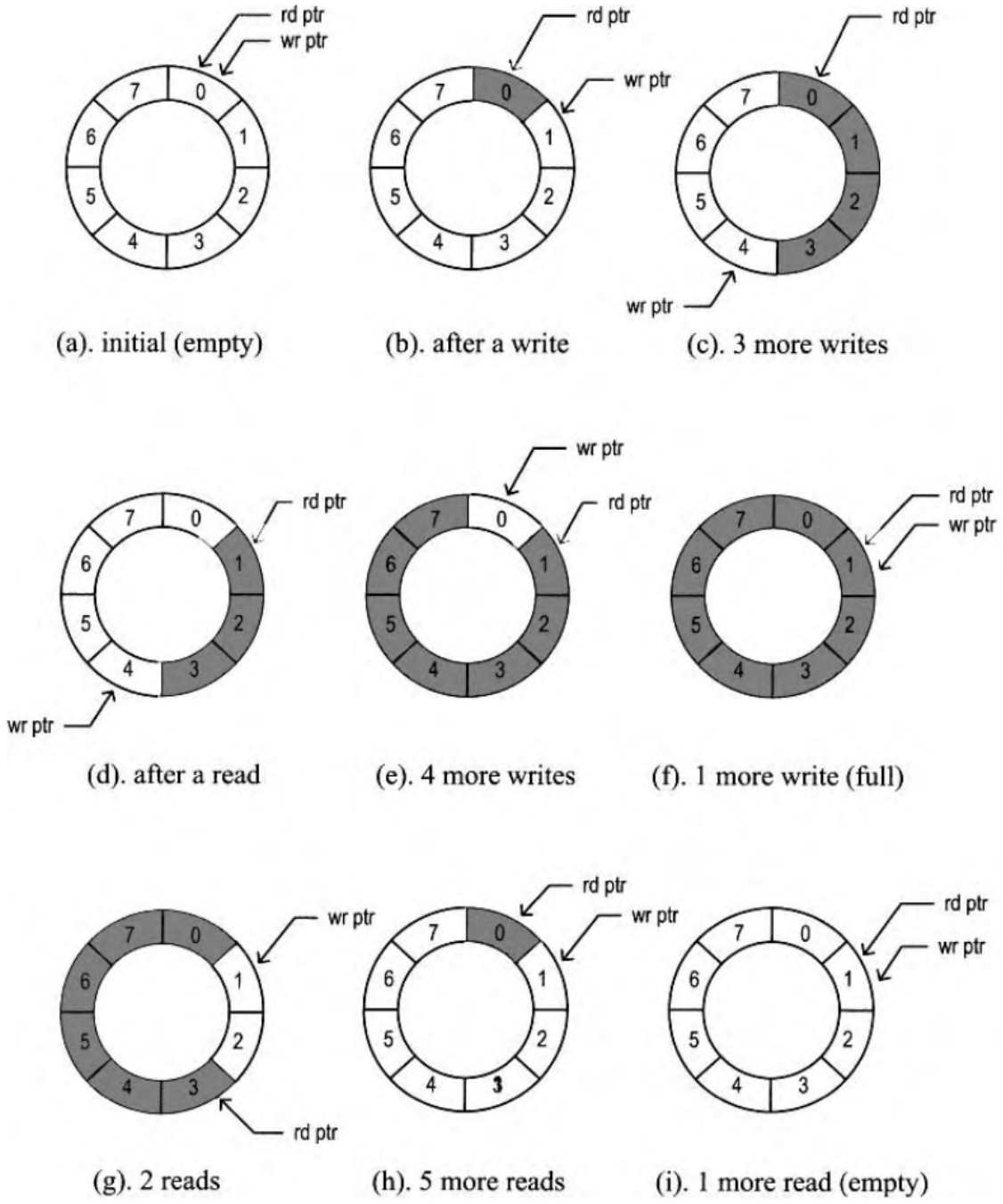


Figure 5.20: Ring buffer diagram

A ring buffer writes data in at the write pointer and count it up by one. When a read state happens it reads from the read pointer and count it up by one. this is also way a full and empty signal i necessary, in order not to overwrite data or read ahead of the write pointer.

IRQ input The input block gets data and interrupt input from every block with interrupt permissions in the system. The purpose of the block is to handle which block is interrupting and which data that have to be written to the FIFO in order to be handle by the ARM7 in the right way. Below the state diagram of the input block is shown. The purpose of a state digram is to make the coding easier.
When in IDLE the FIFO write is set low, the interrupt inputs is check, if non of them is high the interrupt counter is set low, to enable new interrupts. If a interrupt is high the state is change to IRQx, where the data output to the FIFO is set to the data from the interrupting source, after that is checks if it is the first time this data is written to the FIFO, if that is the case, the data is written to the FIFO and the returns to IDLE.

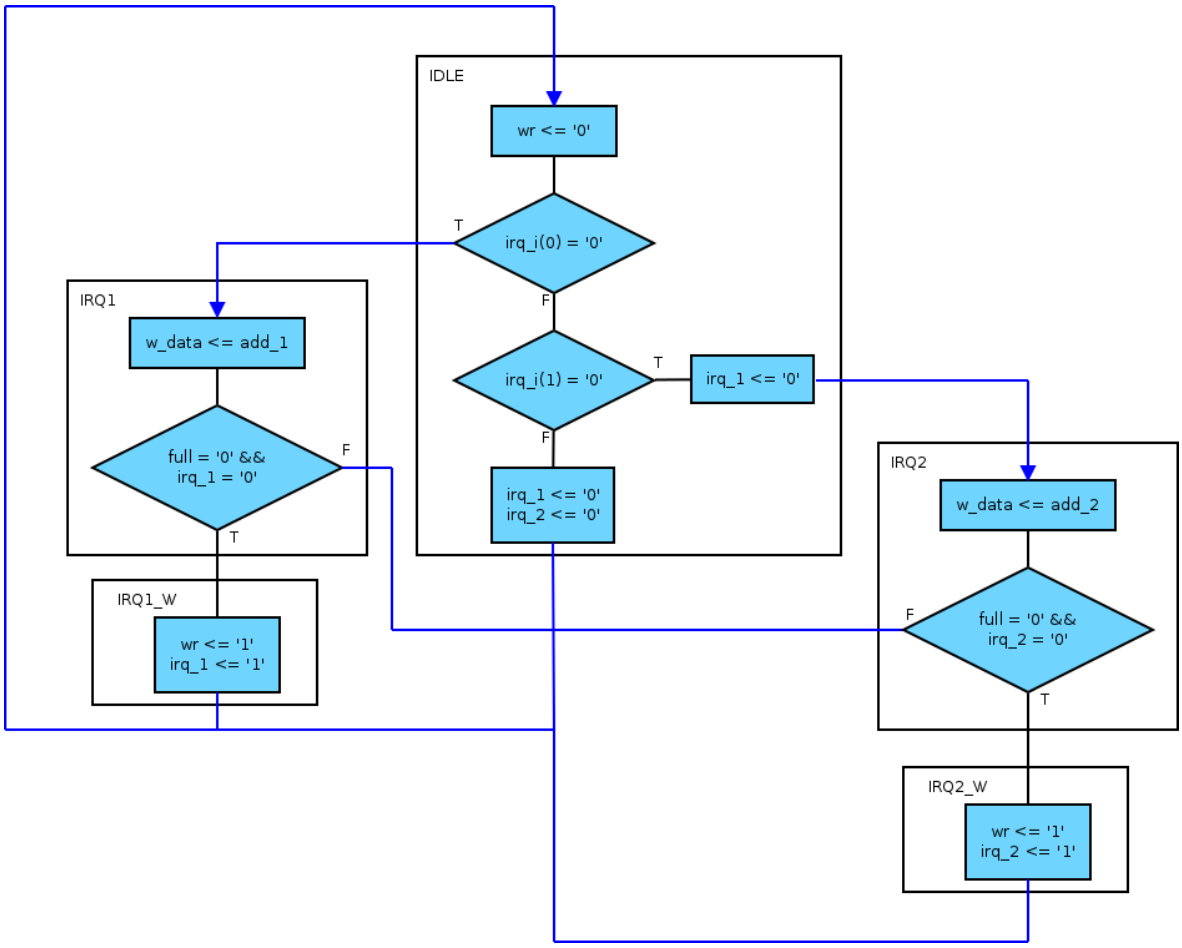


Figure 5.21: State diagram for input

In the code below the states are defined, this is easily done from the state diagram, the signals that can have the different states a value is also defined here.

```
...
  --Types
  type state_type is  (
                IDLE,
                IRQ1,
                IRQ1_W,
                IRQ2,
                IRQ2_W
                );
  --Signals
  signal state_reg    : state_type;
```

```
  signal state_next   : state_type;
  signal irq_1        : std_logic   :=  '0';
  signal irq_2        : std_logic   :=  '0';
...
```

This code is where the states are set, if reset is active the state is set to IDLE else the state is set to state_next.

```
...
  -- state register
  process(Clk, Reset)
  begin
    if  (reset = '1') then
      state_reg   <= IDLE;
    elsif (Clk'event and Clk='1') then
      state_reg   <= state_next;
    end if;
  end process;
...
```

This is the part of the code where the state digram is implemented. A case statement is made with the state value as input. First the IDLE state is defined, here wr is set low, then it check if there is a interrupt, if that is the case then it sets the state to IRQx which sets the data and then check if it is first time the data is written else it just sets the state to IDLE, if it is first time the state is set to IRQx_W where it write the data into the FIFO.

```
...
  -- next state/output logic
  process(state_reg, irq_i, add_1, add_2, full, irq_1, irq_2)
  begin
    state_next    <= state_reg;
    --wr          <= '0';
    case state_reg is
    -- Idle state -------------------------------------------
      when IDLE =>
        wr    <= '0';
        if(irq_i(0) = '1') then
          state_next    <= IRQ1;
        elsif(irq_i(1) = '1') then
          irq_1         <= '0';
          state_next    <= IRQ2;
        else
          irq_1   <= '0';
          irq_2   <= '0';
        end if;
    -- IRQ1 state -------------------------------------------
      when IRQ1 =>
        w_data        <= add_1;
        state_next    <= IRQ1_W;

      when IRQ1_W =>

        state_next    <= IDLE;
        if((not full and not irq_1) = '1') then
          wr      <= '1';
          irq_1   <= '1';
        end if;
    -- IRQ2 state -------------------------------------------
      when IRQ2 =>
        w_data        <= add_2;
        state_next    <= IRQ2_W;

      when IRQ2_W =>

        state_next    <= IDLE;
        if((not full and not irq_2) = '1') then
          wr      <= '1';
          irq_2   <= '1';
        end if;
    end case;
```

```
45   end process;
46 ...
```

IRQ output The output block is reading the data out of the FIFO and present them for the wishbone. It gets a read input from the wishbone, when the ARM7 wants to read data from the register. The main purpose of the block is to make sure that the wishbone read the write data, and it is only reading once from the FIFO.
A state diagram for the output block is shown below. In the IDLE state rd is set low, then et check if the wishbone wants to read, else it reset the read counter, if the wishbone wants to read, the IDLE state checks if it is the first time by checking the read counter, if it is the first time the rd is set high to get the next data from the FIFO, and the read counter is set to one.

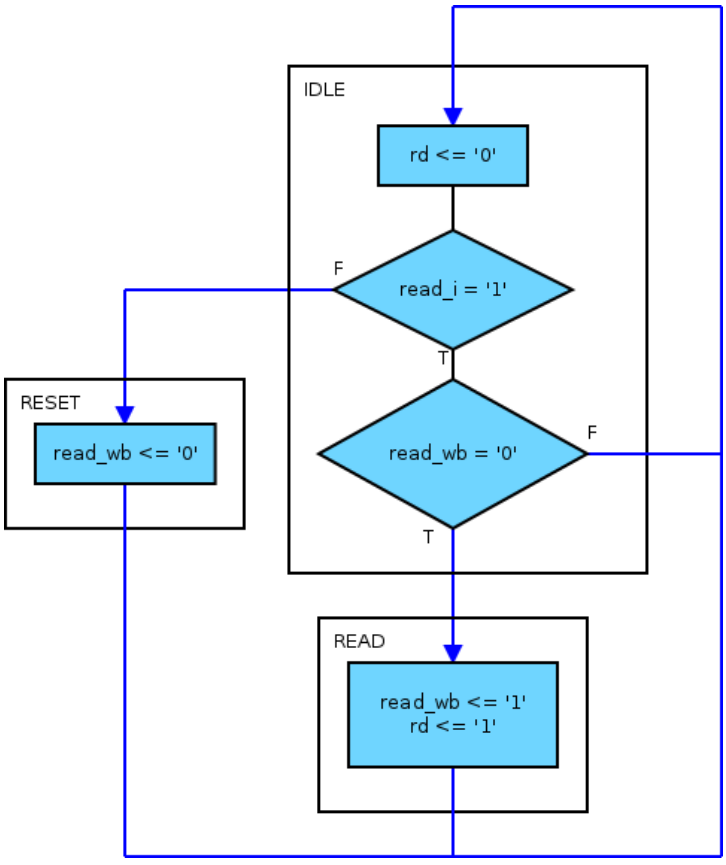


Figure 5.22: State diagram for output

Here the states is defined plus the read counter.

```
1  ...
2    --Types
3    type state_type is  (
4                        IDLE,
5                        READ_0,
6                        RSET
7                        );
8    --Signals
9    signal state_reg    : state_type;
10   signal state_next   : state_type;
```

```
  signal read_wb      : std_logic;
...
```

The lower bits in the data_o is set to the data from the FIFO, the other bits is set to zero. In the process the next state is loaded.

```
...
  data_o(Datawidth-1 downto AddrRange)  <= (others => '0');
  data_o(AddrRange-1 downto 0)          <= r_data;

  -- state register
  process(Clk, Reset)
  begin
    if  (reset = '1') then
      state_reg   <= IDLE;
    elsif (Clk'event and Clk='1') then
      state_reg   <= state_next;
    end if;
  end process;
...
```

Below the state actions is defined following the state diagram. The rd is set low in IDLE and the wishbone read is checked. In READ the read counter is set to one and rd is set high to read data out of the FIFO.

```
...
  -- next state/output logic
  process(state_reg, read_i, r_data, read_wb)
  begin
    state_next  <= state_reg;
    rd          <= '0';
    read_wb     <= read_wb;
    case state_reg is
    -- Idle state ------------------------------------------------------
      when IDLE =>
        rd    <= '0';
        if  (read_i = '1') then
          if  (read_wb = '0') then
            state_next    <= READ_0;
          else
            state_next    <= IDLE;
          end if;
        else
          state_next    <= RSET;
        end if;
    -- READ_0 state ----------------------------------------------------
      when READ_0 =>
        state_next    <= IDLE;
        read_wb       <= '1';
        rd            <= '1';
    -- RSET state ------------------------------------------------------
      when RSET =>
        state_next  <= IDLE;
        read_wb     <= '0';
    end case;
  end process;
...
```

Wishbone interface In order to use the interrupt register in the system, is shall be implemented as a wishbone slave, with a wishbone interface, this is done in the code below. The error and retry bit i set to zero, the acknowledge bit i set then the strobe input and the cycle input is high. In the process a synchronous reset is made. If the reset is inactive, it checks if the strobe and cycle input is high and the write enable is low, the meaning is to check if the wishbone wants to read or write. This module is a read only so it is not reacting on

a write statement. If the wishbone wants to read, the data output is set and a read to the output block is set high, when the wishbone determinates the read cycle the data output is set to zero and the read statement for the output block is set low again.

```
...
-- ================================================
-- WishBone logic
-- ================================================
  --  Concurrent assignments
  --  Wishbone cycle acknowledge
  err_o <= '0'; --error signal
  rty_o <= '0'; --retry signal
  ack_o <= stb_i and cyc_i;  --! asynchronous cycle termination is OK here.
  --Wishbone read
  data_output : process(clk_i)
  begin
    if(clk_i'event and clk_i = '1') then
      if(rst_i = '1') then
        dat_o <= (others => '0');
      else
        if((cyc_i and stb_i and not we_i) = '1') then
          case adr_i is
            when WBS_REG1 =>
              dat_o   <= data_o;
              read_i  <= '1';
            when others =>
          end case;
        else
        dat_o   <= (others => '0');
        read_i  <= '0';
        end if;
      end if;
    end if;
  end process;
...
```

5.5.4 Verification

The verification of the IRQ register is made in a VHDL test bench, because there is currently no blocks with interrupt ability implemented. The verification test bench is shown below. The test is made by make four interrupts, this can be seen on the *IRQ_in* signal which has four pulses on the two signals. The address on *add_1* and *add_2* is written into the FIFO. The test bench also shows that the *IRQ_out* change after the first interrupt is assigned. After this four wishbone reads are made, by setting the cycle and the strobe input high, this give a acknowledge bit, and the *Data_out* is change to the first element in the FIFO, and the next read get the next data and further on. After the last wishbone read the FIFO is empty and the *IRQ_out* is set high again to indicate there is no interrupt.

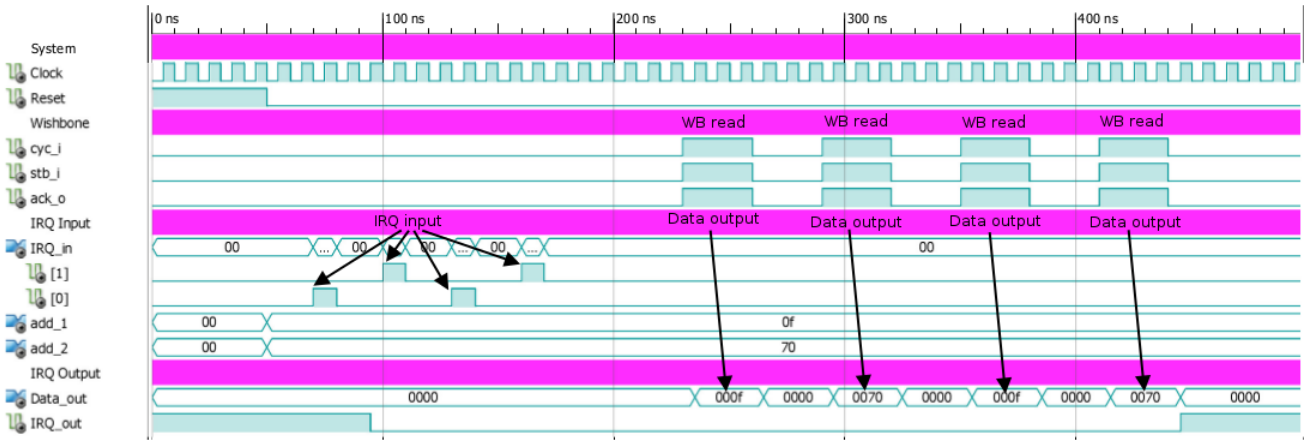


Figure 5.23: Test bench for the interrupt register

5.5.5 Conclusion

The test bench verify that the interrupt register is working with virtual signals in simulation. The switch block is to be change to have interrupt ability, and then it shall be implemented with the interrupt register and tested properly with communication and interrupt to the ARM7

5.6 Deployment

UART Device Driver The first version of the Device Driver is up and running (v0.1), which has implemented read and write functionality (non interruptible). Also different write calls have been implemented in order to initialize a channel.

Power Switch The power switch system is under development, 4 of this modules will be developed and a test bench build using fast prototyping tools. In time box 6 the complete development of this main system will be finish and a prototype will be present to the client.

Interrupt register The interrupt register works in a test bench with wishbone interface. The implementation in the Spartan 6 is the next step and, interrupt ability is to be added to the switch block.

6 Time box 6

6.1 Time box planning

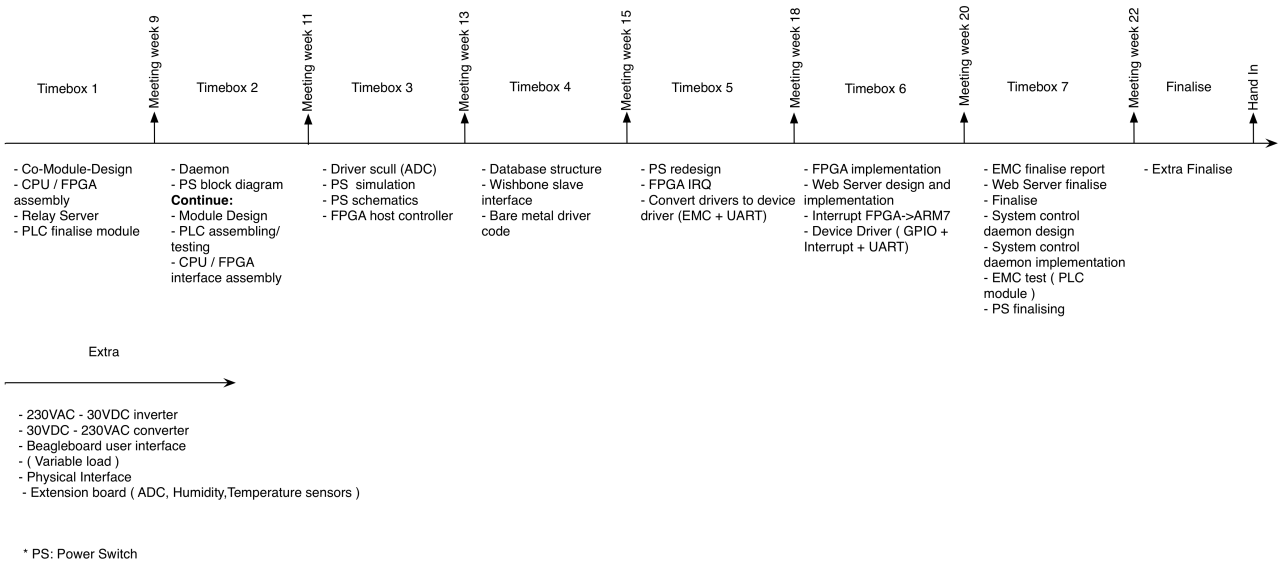


Figure 6.1: Updated time-box

6.1.1 Work to be done in this time box

- Switch interrupt
  - debouncer
  - Interrupt
- Web Server
  - Web Services - Communication
  - File structure
- UART Device driver
  - Improvement on the driver made in last timebox

Description:

Switch interrupt In order to send data to the interrupt register, an interrupt output for the switch block has to be implemented, and the switches has to be debounced, to secure that the interrupt data is send only once.

Web Server The web server is the communication between the system and the world. The communication between the web server and energy hub is implemented and the file structure is set for the collaboration of other teams/modules.

UART Device driver Implementation of input/output control and interrupt handling in the UART device driver made in time box 5.

6.1.2 Time planning

	Switch interrupt	Jesus thing	UART improvements
Estimation	12	15	5
Actual	20	22	17
Developer	Theis	Paulo	Dennis

Table 6.1: Estimation and actual time used on the project

6.2 Project scaling

Unfortunately there is still a lot of work tasks to be done before the fully project is realized. With only 2 weeks before delivering the report it has been decided to down scale the project, to have some blocks fully up and running. The picture below shows the parts that will hopefully be implemented in the handed in version.

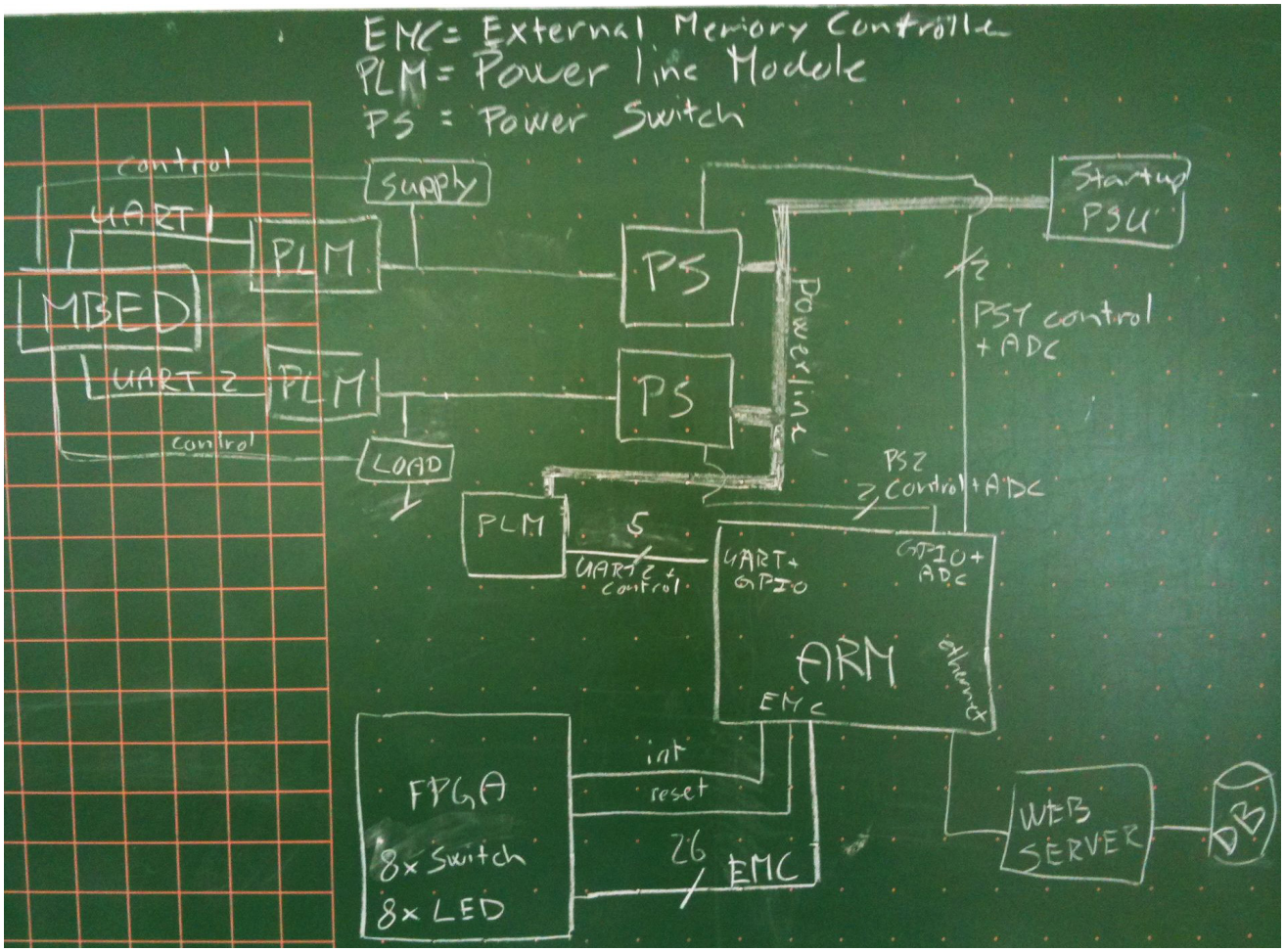


Figure 6.2: Project Scaling diagram

The scaled project will then contain:

- ARM to FPGA communication through External Memory Controller. The ARM gets an interrupt from the FPGA whenever a switch changes status. The ARM then sets the LEDS on the ARM board according to the switches status.
- UART communication to Power Line Module to send and transmit data between the modules and the ARM board.
- MBED setup to work as two modules (communicating with the ARM board through two power line modules).
- Dummy protocol to send data from MBED to ARM board.
- Send measurements from modules to the web server.
- Control modules status through the web interface.
- Save current sensors measurements to database and plot data in the web interface.
- System control background application running on ARM board.

The setup will then be able to check which state it should be running in, according to the switches set on the FPGA board. The ARM board will be able to control the Power Line modules (switching direction and turning off modules). When communication is established between the mBed and the ARM through Power Line, the data received from the mBed (Module) is convert and sent to the web server with a URL request to the web service savedata.php, the data is then saved into the database. In the web interface the user is able to start or stop a module and see the current production or consumption of the system.

6.3 Switch interrupt - Theis

This part is, together with the interrupt register from earlier timebox a way to tune performance in reading and writing between the Spartan 6 and the LPC2478.

6.3.1 Analysis

The interrupt block in the Spartan 6 is the switch block. The switch block needs to be redesign in order to get an interrupt output to the interrupt register, the switches also needs to be debounced, to prevent it from sending the same interrupt data more than once. Two output is added to the switch block, one single bit for interrupt indication and a 7 bit vector for the data to the interrupt register. Inside the block a finite state machine is made for denouncing the switches. Below the redesigned switch block is shown.

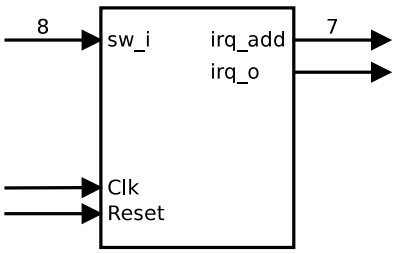


Figure 6.3: Updated switch block

6.3.2 Design

Because of switch bounce, the block needs to take care of this. Below a picture of switch bouncing is shown. The problem is every time the signal goes high the switch block will send data to the interrupt register, to prevent this the block compare a delayed input signal to the present signal, and first when the signal is stable, the system will react on the input.

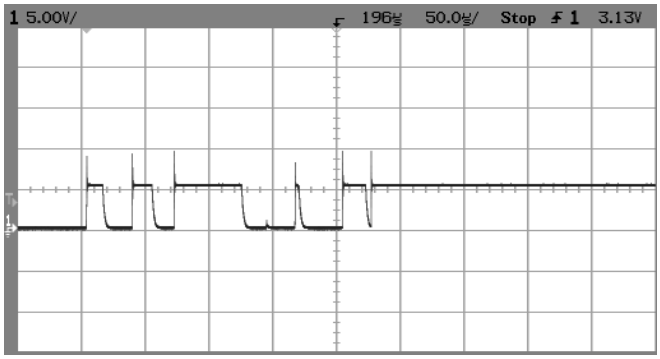


Figure 6.4: Switch bouncing

To debounce the switch a state machine for the switch block is made. This diagram is shown below. The start box set the start output signal to the input signal. The interrupt signal is set high in the OUT0 state, and low again in the IDLE state after the "q2" delay. This is done to have the interrupt signal high enough time for the interrupt register to save the data. The "q1" delay is used to compare the input signal with the output signal in order to capture changes on the 8 switches.

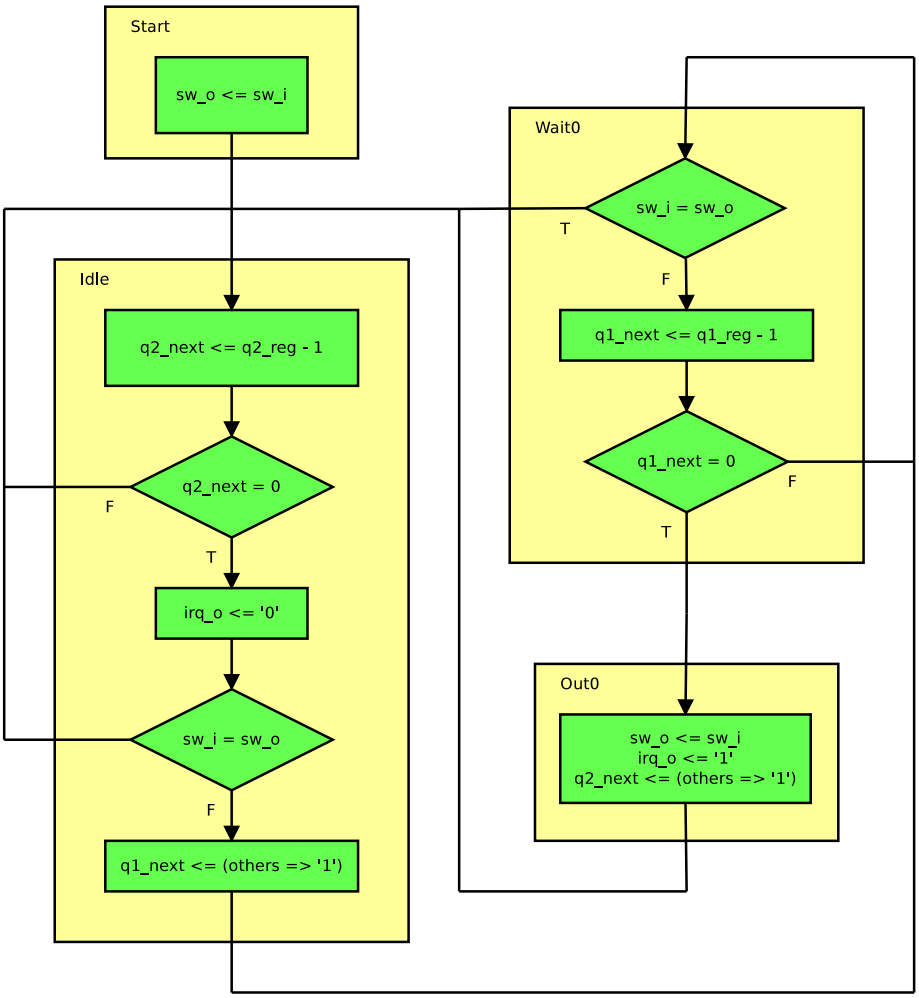


Figure 6.5: Switch block state machine

6.3.3 Implementation

The code for the IDLE state is shown below, here the "q2" delay is used in the start, then the interrupt pin is set to zero, then the input and output is compared, if they are not equal the next state is WAIT0 and the "q1" delay is set.

```
...
when IDLE =>
  q2_next <= q2_reg-1;
  if  (q2_next = 0) then
    irq_o <= '0';
    if  (sw_i = sw_o) then
      state_next <= IDLE;
    else
      q1_next   <= (others => '1');
      state_next  <= WAIT0;
    end if;
  else
    state_next <= IDLE;
  end if;
...
```

In the WAIT0 state the input and output is compared repeatedly to check if the input is stable. If the input is stable long enough time the OUT0 state is entered.

```
...
when WAIT0 =>
  if  (sw_i = sw_o) then
    state_next <= IDLE;
  else
    q1_next <= q1_reg-1;
    if  (q1_next = 0) then
      state_next <= OUT0;
    else
      state_next <= WAIT0;
    end if;
  end if;
...
```

In the OUT0 state the switch output is set to switch input signal, and an interrupt signal is set high, the "q2" delay is set and it returns to the IDLE state.

```
...
when OUT0 =>
  sw_o     <= sw_i;
  irq_o    <= '1';
  q2_next   <= (others => '1');
  state_next  <= IDLE;
...
```

6.3.4 Verification

The code is tested on a test bench in isim. The test verify that the block first set the switch output after the input has been stable for a while. And when the output is set the interrupt signal is set high for some time and then set low again.

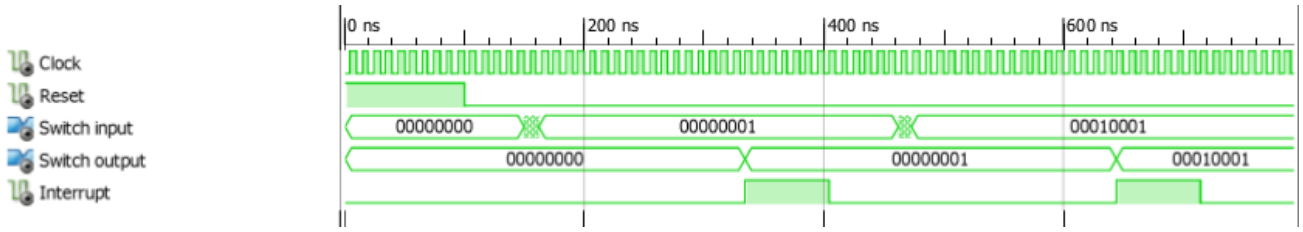


Figure 6.6: Switch block test bench

6.3.5 Conclusion

The test shows that the interrupt function is working properly, the design is also tested on the Spartan 6 with the LCP2478, and it is working as expected.

6.4 Web Server - Paulo

In time box 4 a web server was implemented at the ip address 10.1.18.223, this is a virtual machine assign as development environment for the uClinux distribution. The server is running Apache 2, PHP version 5.1.6 and

MySQL server 5.0.95. In this time box a web services system is developed for the communication between the Embedded device and the web server and the other way around. The web page made in Project 3 is incorporated with server side scripts (PHP) for a fully functional web interface.

An user with read only permissions was created:

User: eval

Pass: ede10eval

For evaluation reasons a web application is developed with the name SeeIt, this is a PHP web application that allows the user to see the file structure of the server and the file content, this can be seen at http://10.1.18.223/Seeit/.

The development of the energy hub web interface can be followed at:

http://e10.ede.hih.au.dk/index.php/Web_Server_Structure

The database structure for this project was change to a common database to all modules, the new structure can be seen at:

http://e10.ede.hih.au.dk/index.php/Common_Database

With the credentials above, the user is able to login into the phpMyAdmin tool in the AU-Herning network at the address: http://10.1.18.223/phpMyAdmin where the database structure is implemented.

6.4.1 Analysis

For a fully functional web interface the communication have to present in both direction, since some teams need to send commands from them web page to the modules. A file structure was created in the server, where each team have is own folder where all the needed scripts, images and layout styles can be implemented without changing the common layout and requirements approved in project 3.

In this time box the follow scripts are created:

- index.php
- savedata.php
- sendcmd.php
- saveip.php
- db_connect.php
- db_globals.php

Web interface file Structure The file structure is still in development, as such and for the correct version the structure is updated at the address:

http://e10.ede.hih.au.dk/index.php/Web_Server_Structure A short description for each script can be seen bellow:

- index.php - First page of the web interface.

- savedata.php - Webservice that save data retrieved from the module to the database.
- sendcmd.php - Webservice to send commands to the desired module in the system.
- saveip.php - Webservice necessary to save the ip address of the energy hub, this will keep the system up and running even if a change on the network is made.
- db_connect.php - Handle the connection to the MySQL database.
- db_globals.php - Includes all the global variables with the credentials for the database.

Communication
The server have to save the data retrieved from the system and send commands to the modules connected to the energy hub. Two main scripts are created: *savedata.php* this script capture the values send by the HTTP request has a POST method saving them to the database and *sendcmd.php* enables the user to send commands to a precise module in the system or even the energy hub.
Measurements to be saved in the database:

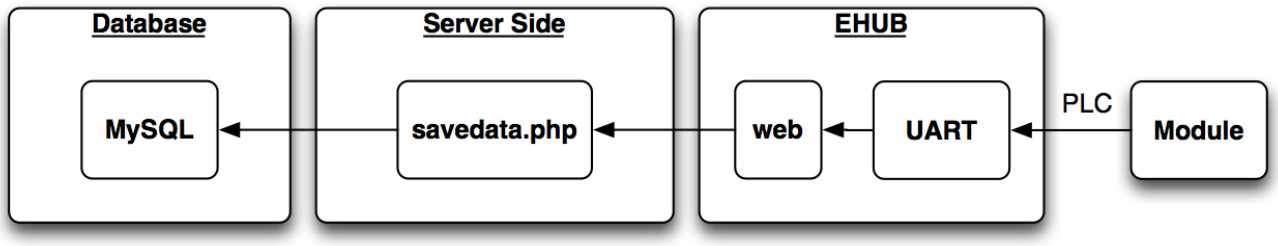


Figure 6.7: Save measurements retrieved by the modules

To retrieve the measurements from the modules, an application running at the energy hub translate the data retrieved from the module through PLC to a URL request at the web server. In the server side the web server will collect the data and save it in the database.

Flow of the communication from the user until the final destination in this case the module or the energy hub.

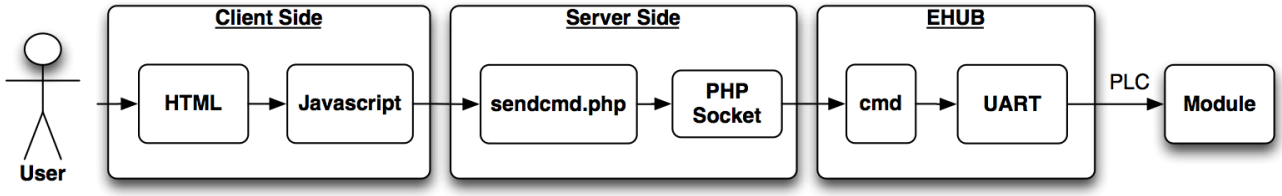


Figure 6.8: Send a command to a module

This script doesn't give a feedback to the user, the commands send are not verified by the web server or the energy hub. The commands are handle by each module. With this system the flexibility of the system is ensured, since new modules can be added with different functionalities from the already known. An application running in background at the energy hub OS, ensure that the command is translated to UART so it can be send through the power line communication to the modules.

IP address is send from the Embedded Device:

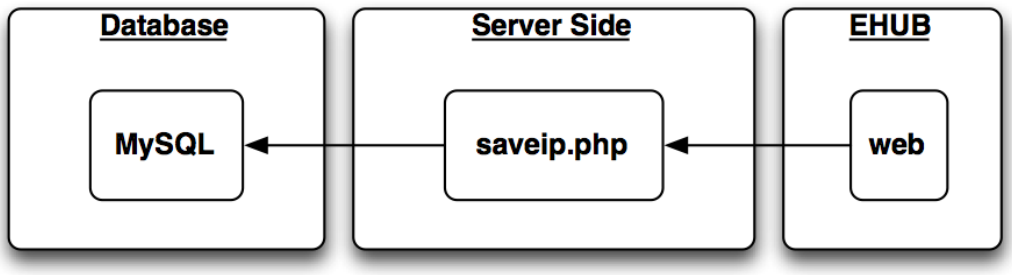


Figure 6.9: Save the IP address of the Embedded device

A background application running (daemon), ensure that after a reboot, the IP address assigned by DHCP to the energy hub is saved in the database so commands can be send through the web interface. This add flexibility to the system, since a change in the internal network could stop the normal work of the system.

Structure Generals

A folder is created for each team, it will contain the necessary scripts to show data for the users, send commands to the energy hub, make their own layout, etc.

This is a necessary structure for this project, since the requirements are different for all the teams. For example the energy hub page needs to show completely different data from the wind turbine or other modules.

In real production software a general layout and data should be set for all the modules pages, being possible to make small adjustments to satisfy the final user/client. This way the final software would have a improved user experience since the layout is equal in all the modules.

6.4.2 Design

The main structure for the correct work of the system is created, in the next time box the energy hub page is built to allow the user to stop and start modules, get the current efficiency as green energy system, etc.

6.4.3 Implementation

index.php

The *index.php* is the main page of the system, this will redirect the users to the modules or login page. The efficiency of the system can be seen on the first page as the amount of lamps able to power, the money saved and the mount of less CO2 emissions, this is not available in this version, will be part of the hub web page development in next time box.

savedata.php

A background application running in the energy hub converts the measurement sent through PLC (UART) to a HTTP request **savedata.php?sensor_id=ID Sensor&value=Sensor Measurement&hub_port=Hub Port**.

```php
<?php

  require_once("db_connect.php");

  $date = getdate();

  $today = $date['year'].'-'.$date['mon'].'-'.$date['mday'].' '.$date['hours'].':'.$date['minutes'].':'.$date
      ['seconds'];

  if(isset($_GET['sensor_id']) && isset($_GET['value']) && isset($_GET['hub_port'])){
    $sql= "INSERT INTO `iEnergy`.`MEASUREMENTS` (".
        "`ID_MEASURE` ,".
        "`ID_SENSOR` ,".
        "`DATE_TIME` ,".
        "`HUB_PORT` ,".
        "`VALUE`)".
        "VALUES (NULL , '".$_GET['sensor_id']."', '".$today."', '".$_GET['hub_port']."', '".$_GET['value']."
            ')";

    $con->query($sql); // Run the query in the MySQL server

    // No feedback needed since the energy hub will not expect an answer.

  } else {
    echo 'Incorrect parameters';
  }

?>
```

This script saves the measurement retrieved from a sensor to the database. At first it established the connection to the MySQL server so SQL requests can be made. A PHP function returns an array with the current date and time, this is formatted into YYYY-M-D H:M:S, after it can be saved to the MEASUREMENTS table. The script collects all the parameters send through the URL encoding GET, a SQL code is generated and a request made to the MySQL server, the data is added to the MEASUREMENTS table.

sendcmd.php

The web interface is able to send commands to the energy hub and the modules using the **sendcmd.php?id_module=<module id>&cmd=<command to be send>**. The id_module tells the hub which module the command should be send. No verification is made of the send command by the web server or the energy hub unless the command is specifically send to the hub.

```php

  // SQL request for the device page.
  require_once("includes/db_connect.php");

  if(isset($_GET['cmd']) && isset($_GET['id_module'])){

    $device_port = 5555;

    //echo 'Module Id: '.$_GET['id_module']."<BR>Command: ".$_GET['cmd']."<BR>";

    $sql= "SELECT IP ".
        "FROM `DEVICE` ".
        "ORDER BY ID_DEVICE DESC ".
        "LIMIT 1";

    $res = $con->query($sql); // Run the query in the MySQL server

    $row = $res->fetch_row();

    $device_ip = $row[0];

    if ($socket=socket_create(AF_INET, SOCK_STREAM, SOL_TCP)){
      echo "Socket created <br>";
    }
```

```
25
26      if (socket_connect($socket,$device_ip,$device_port)){
27        echo "Connection stablished<br>";
28      } else {
29        exit (socket_strerror(socket_last_error()));
30      }
31
32      $str = $_GET['cmd'].';'.$_GET['id_module'];
33
34      socket_write($socket,$str);
35
36      socket_close($socket);
37
38    } else {
39      echo 'Command or Module Id not set';
40    }
```

At first the script will get the ip address of the energy hub, running a SQL code that retrieves the last IP address added to the table DEVICE. With the energy hub ip and a predefined port, a connection is created using a TCP socket for the communication. The command and the module id are send and the socket is closed.

saveip.php

saveip.php script is called by the background application running on the embedded device, it saves the IP address given by DHCP, this is used for further communication between the web server and the energy hub.

```
1   require_once("includes/db_connect.php");
2
3   if(isset($_GET['ip'])){
4     $sql= "INSERT INTO `iEnergy`.`DEVICE` (".
5         "`ID_DEVICE` ,".
6           "`IP`)".
7           "VALUES (NULL , '".$_GET['ip']."')";
8
9     $con->query($sql); // Run the query in the MySQL server
10
11    // No feedback needed since the energy hub will not expect an answer.
12
13  } else {
14    echo 'IP not set';
15  }
```

The ip address is send by the GET method (saveip.php?ip=127.0.0.1), this is how the data in encoded into a URL, being collected in the variable $_GET['ip']. A $sql variable string is created containing the SQL code to be run at the MySQL server.

db_connect.php

For the communication to the database a driver is used in PHP that provides an interface to the MySQL server. The PHP mysqli extension (MySQL improved) is used in this project, this is recommend for MySQL servers version 4.1.3 or later. This extension provides several benefits as a objective-oriented interface, support for multiple statements, embedded server support and more can be found in the MySQL documentation.

```
1   require_once("db_globals.php");
2
3   $con = new mysqli(DB_HOST,DB_USER,DB_PASS,DB_NAME); // Creates new mysql connection
4
5   if($con->connect_error){
```

```
6     echo "Failed to connect to MySQL: (" . $con->connect_errno . " ) ". $con->connect_error;
7   }
8   else { echo "Connection established"; }
```

In this script an object is instantiated with a connection to the MySQL server.

$con = new mysqli<parameters >)

The parameters are included from the db_globals.php, setting the server host, user, password and database to be used.

db_globals.php

Using a script to define the parameters for the MySQL connection, All the scripts that need to use the global parameters for the connection to the MySQL server, should include db_globals.php as shown in the db_connect.php above.

```
1   // MySQL condifuration
2   DEFINE ('DB_USER','root');
3   DEFINE ('DB_PASS','root');
4   DEFINE ('DB_HOST','localhost');
5   DEFINE ('DB_NAME','iEnergy');
```

The global parameters the connection to the MySQL server are define in this script.

- DB_USER - Database user name with read, write and execute permissions.
- DB_PASS - Password for the user
- DB_HOST - Hostname for the MySQL server, if running at the same host as the PHP server, localhost or 127.0.0.1 should be used.
- DB_NAME - Database name to connect to.

6.4.4 Verification

The verification is made using the tools described in the beginning of this section:

Verification	Description	Acceptance
1	Save new IP address: saveip.php?ip=127.0.0.1	OK
2	Send command: sendcmd.php?unique_id=0&cmd=led 1 on	TimeBox7
3	File structure created	OK

6.4.5 Conclusion

The common web services and file structure in the web server was created for all the modules/teams.
The implementation of the savedata.php will required further analysis for a working system, since the energy hub needs to have a table of all the sensors connected to each module. When a module retrieves a value the hub needs to know which sensor the measurement belongs to. This was not taken in consideration in earlier analysis of the

system and might affect the protocol developed and system dynamics.
In the next time box a scaled down prototype is developed and the communication dynamics between web server and modules can be tested.

6.5 UART Device driver improvements - Dennis

As described in section 5.4.5, some improvements should be implemented in order to boost performance on the UART devices. The parts which will be further implemented in the UART device driver is:

- Interrupt controlled read.
- More flexibel initialization.
- Better feedback/help description to the user.

Furthermore it was pointed out (by Klaus Kolle) that the implemented way of setting up the device broke general politic for device drivers. To avoid this, an IOCTL (Input Output Control) function shall be implemented to take care of all setup and to leave the write function to sending characters.

6.5.1 Design

The functionalities implemented in the IOCTL call are:

- Help command. Write out the different choices of IOCTL calls.
- Default setup call. Sets up the defined UART to 8n1, 9600 baud rate.
- Baud rate. Set a baud rate in the range 2400 to 230400.
- Word length. Define the word length between 5 and 8.
- Stop bits. Number of stop bits, 1 or 2.
- Parity bits. Define parity bit setting: none, odd, even, forced 1 stick, forced 0 stick.
- Fifo. Enable or disable fifo.
- Fifo trigger. Number of characters in the buffer before an interrupt: 1,4,8,14

6.5.2 Implementation

Common header file for the UART device driver and the UART user space application. The magic number 'k' is used to by the IOCTL macros _IO and _IOR to create an IOCTL number, which is then decoded and verified in the kernel module.

```
1  #ifndef UARTIOCTL_H
2  #define UARTIOCTL_H
3
4  #include <linux/ioctl.h>
5
6  #define UART_IOC_MAGIC  'K'
7
8  #define IOCTL_HELP    _IO(UART_IOC_MAGIC,  1)
9  #define IOCTL_DEFAULT _IO(UART_IOC_MAGIC,  2)
10 #define IOCTL_BAUD    _IOR(UART_IOC_MAGIC, 3, int)
11 #define IOCTL_WORDLEN   _IOR(UART_IOC_MAGIC, 4, int)
```

```
#define IOCTL_STOPBIT     _IOR(UART_IOC_MAGIC, 5, int)
#define IOCTL_PARBIT    _IOR(UART_IOC_MAGIC, 6, int)
#define IOCTL_FIFO      _IOR(UART_IOC_MAGIC, 7, int)
#define IOCTL_FIFO_TRIG   _IOR(UART_IOC_MAGIC, 8, int)

#define UART_IOC_MAXNR 8

#endif
```

Using IOCTL to setup UART registers. Note that the *fwrite* call cannot be used with IOTCL, as it needs an file descriptor instead of a file pointer. Therefore the libraries *fcntl.h* and *unistd.h* are included in order to use the functions *open, close, read, write* and *ioctl* as they make use of the file descriptor.

```
  if(strncmp("io", argv[1], 2) == 0){
    if ((fd = open(UART,O_RDWR)) < 0){
                  printf("Cannot open file.\n");
      exit(-1);
          }

    ret_val = ioctl(fd, IOCTL_DEFAULT);
    if(ret_val < 0){
      printf("ioctl_get_msg failed:%d\n", ret_val);
      exit(-1);
    }
    ret_val = ioctl(fd, IOCTL_BAUD, 19200);
    if(ret_val < 0){
      printf("ioctl_get_msg failed:%d\n", ret_val);
      exit(-1);
    }

    close(fd);
  }
```

Loop reading used for testing purpose of the power line communication.

```
  ---
  else if(strncmp("readloop", argv[1], 8) == 0){
    if ((fd = open(UART,O_RDONLY)) < 0){    //Open the file
                  printf("Cannot open file.\n");
      exit(-1);
          }
    while(1){
      if(read(fd,rb,10) != 0)
        printf("%s\n",rb);
    }
    printf("\n");
    close(fd);
  }
```

IOCTL added to file operator

```
static struct file_operations uart_fops = {
  .owner    = THIS_MODULE,
  .read     = uart_read,
  .write    = uart_write,
  .open     = uart_open,
  .ioctl    = uart_ioctl,
  .release  = uart_close,
};
```

IOCTL implementation. At first the function verifies the IOCTL number sent and checks if the transferred command is valid. The rest of the function is a switch/case which takes the cmd parameter sent and performs an action according to the command (help, default setup, baud rate etc.)

```
/*****************************************************************************
 * IOCTL call
 *****************************************************************************/
static int uart_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
  unsigned int num = 0;
        int err = 0, ret = 0;
  int div, mult,bflag = 0;
  unsigned int baud = 0;

        /* don't even decode wrong cmds: better returning  ENOTTY than EFAULT */
        if (_IOC_TYPE(cmd) != UART_IOC_MAGIC) return -ENOTTY;
        if (_IOC_NR(cmd) > UART_IOC_MAXNR) return -ENOTTY;

        /*
         * the type is a bitmask, and VERIFY_WRITE catches R/W
         * transfers. Note that the type is user-oriented, while
         * verify_area is kernel-oriented, so the concept of "read" and
         * "write" is reversed
         */
        if (_IOC_DIR(cmd) & _IOC_READ) err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
        else if (_IOC_DIR(cmd) & _IOC_WRITE) err =  !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd
            ));
        if (err) return -EFAULT;
  // Get minor number
  num = MINOR(inode->i_rdev);
  DPRINT("\nuart%d_ioctl, ioctl_cmd: %u\n",num,cmd);
  // Check if minor number is okay
  if(num >= NUM_UART_DEVICES){
    return -ENODEV;
  }

  switch (cmd) {
  /***************************** HELP ********************************************/
  case IOCTL_HELP:
      DPRINT("\nHELP\n");
      help(); // Print different IOCTL call options to user
    break;
  /***************************** DEFAULT *****************************************/
  case IOCTL_DEFAULT:
      m_reg_write(ulcr[num], 0x80); // Enable write to divisor latch
      m_reg_write(udll[num], (unsigned char)((LPC24xx_Fpclk/(16*9600)) & 0xFF));
      m_reg_write(udlm[num], (unsigned char)((LPC24xx_Fpclk/(16*9600)) >> 8));
      m_reg_write(ulcr[num], 0x03); // 8N1 setup and disable write to divisor latch
      m_reg_write(ufcr[num], 0x7);  // Enable FIFO's
    break;
  /***************************** BAUD ********************************************/
  case IOCTL_BAUD:
      if(arg < 2400 || arg > 230400){ // Verify argument
    printk("\nInvalid parameter");
    return -EFAULT;
      }
      bflag = 0;
  /* From datasheet:
     Mult values: 1-15, Div: 0-14. Mult shall be bigger than div.
     DLL shall be above 2 if DLM is zero.

  */
     // Loop through different div and mult values in order to get a integer value to
     // the divisor latch reg (DLM+DLL).
      for(div = 0; div <= 15; div++){
    for(mult = div+1; mult <= 15; mult++){
        baud = ((16*arg)+((16*arg*div)/mult));
        if(LPC24xx_Fpclk%baud == 0){
      if((div != 0) && ((LPC24xx_Fpclk/baud) >= 3)){
          bflag = 1;
          break;
      }
        }
    }
    if(bflag==1) break;
      }
      DPRINT("\ndiv: %d, mult: %d, baud: %d",div,mult,baud);
      if(div == 15){  // If no value was found, return error
    printk("\nCannot calculate BAUD!");
    return -EFAULT;
      }
      m_reg_bfs(ulcr[num], 0x80); // Enable write to divisor latch
```

```
77        m_reg_write(ufdr[num], ((div<<0) | (mult<<4))); // Write div and mult to fraction dividor reg
78        m_reg_write(udll[num], (unsigned char)((LPC24xx_Fpclk/baud) & 0xFF)); // Input DLL val (8 lowest bits)
79        m_reg_write(udlm[num], (unsigned char)((LPC24xx_Fpclk/baud) >> 8));   // Input DLM val (8 highest bits)
80        m_reg_bfc(ulcr[num], 0x80); // Disable write to divisor latch
81
82      break;
83    /***************************** Wordlen ********************************************/
84    case IOCTL_WORDLEN:
85        if(arg < 5 || arg > 8){ // Verify that argument is valid
86      printk("\nInvalid parameter");
87      return -EFAULT;
88        }
89        m_reg_bfc(ulcr[num], 0x3);  // Clear bits holding world lenght
90        m_reg_bfs(ulcr[num], ((arg-5)<<0)); // Inset value for world lenght
91      break;
92    /***************************** STOP bits ******************************************/
93    case IOCTL_STOPBIT:
94        if(arg != 1 || arg != 2){ // Verify that argument is valid
95      printk("\nInvalid parameter");
96      return -EFAULT;
97        }
98        m_reg_bfc(ulcr[num], (1<<2)); // Set stop bit to 1
99        if(arg == 2){
100         m_reg_bfs(ulcr[num], (1<<2)); // Set stop bit to 2
101       }
102     break;
103   /***************************** Parity Bit *****************************************/
104   case IOCTL_PARBIT:
105       if(arg < 0 || arg > 4){ // Verify that argument is valid
106     printk("\nInvalid parameter");
107     return -EFAULT;
108       }
109       m_reg_bfc(ulcr[num], (1<<3)); // Disable parity bits. if 0 was written
110       if(arg > 0){
111     m_reg_bfs(ulcr[num], (1<<3)); // Enable parity.
112     m_reg_bfs(ulcr[num], ((arg-1)<<4)); // Set parity (odd, even, forced 0 or 1
113       }
114     break;
115   /***************************** FIFO Enable ****************************************/
116   case IOCTL_FIFO:
117       if(arg != 0 || arg != 1){ // Verify that argument is valid
118     printk("\nInvalid parameter");
119     return -EFAULT;
120       }
121       m_reg_bfc(ufcr[num], 0x7);  // Disable fifo
122       if(arg==1){
123     m_reg_bfs(ufcr[num], 0x7);  // Enable fifo
124       }
125     break;
126   /***************************** FIFO Enable ****************************************/
127   case IOCTL_FIFO_TRIG:
128       if(arg != 1 || arg != 4 || arg != 8 || arg != 14){  // Verify that argyment is valid
129     printk("\nInvalid parameter");
130     return -EFAULT;
131       }
132       m_reg_bfc(ufcr[num], 0xC0); // Clear bits holding trigger level
133       m_reg_bfs(ufcr[num], (int)((arg/4)<<6));  // 1/4=0, 4/4=1, 8/4=2, 14/4=3
134     break;
135   /***************************** Wrong **********************************************/
136   default:
137       help(); // If wrong parameter sent. Print options.
138       return -ENOTTY;
139   }
140
141         return ret;
142 }
```

Interrupt implementation. The interrupt is requested in the open call. If there is no data to read, the module is sent to sleep and awaken again when the buffer is not empty anymore. In close the interrupt is freed again.

```
1 //Interrupt sources for UART0,1,2,3
2 #define UART0_IRQ 6
3 #define UART1_IRQ 7
4 #define UART2_IRQ 28
```

```
#define UART3_IRQ 29
//Array of interrupt flags
static int flag[NUM_UART_DEVICES];

//Prototypes of interrupt functions. Int\_uart is an array of function pointers.
static irqreturn_t interrupt_uart0(int irq, void *dev_id);
static irqreturn_t interrupt_uart1(int irq, void *dev_id);
static irqreturn_t interrupt_uart2(int irq, void *dev_id);
static irqreturn_t interrupt_uart3(int irq, void *dev_id);
typedef irqreturn_t (*INTERRUPT_UART)(int irq, void *dev_id);
const INTERRUPT_UART int_uart[NUM_UART_DEVICES] = {interrupt_uart0, interrupt_uart1, interrupt_uart2,
    interrupt_uart3};

static int uart_open(struct inode* inode, struct file* file){
---
  flag[num] = 0;
  m_reg_write(uier[num], 0x1);  //Enable interrupt on rx buf not empty
  m_reg_bfs(VICSoftIntClear, (1<<uirq[num])); // Clear interrupts from uart source
  m_reg_bfs(VICIntEnable, (1<<uirq[num]));    // Enable uart interrupt

  ret = request_irq(uirq[num], int_uart[num], SA_INTERRUPT,"UART interrupt", NULL); //NULL = Pointer based on
      IRC handler

  if(ret){
    printk("IRQ %d is not free. RET: %d\n", UART2_IRQ,ret);
    return ret;
  }
  return 0;
}

static int uart_close(struct inode* inode, struct file* file){
  ---
  free_irq(uirq[num], NULL);  // Free interrupt
  return 0;
}
```

UART read call.

```
static ssize_t uart_read(struct file *p_file, char *p_buf, size_t count, loff_t *p_pos){
  ---
  if(!(m_reg_read(ulsr[num]) & 0x01)){    //If the read buffer is empty, go to sleep.
    DPRINT("\nBUF EMPT, SLEEP\n");
    wait_event_interruptible(my_queue, (flag[num] != 0)); // Put the function to sleep
    flag[num] = 0;
  }
  else{
    cread = m_reg_read(urbr[2]);
  }
  *p_buf = cread;   // Write value to read buffer.

  return count;
}
```

UART interrupt function. The four different interrupt functions are all similar, except for the registers that is handled.

```
/*****************************************************************************
 * Interrupt recieve routine UART2
 *****************************************************************************/
static irqreturn_t interrupt_uart2(int irq, void *dev_id){

  cread = m_reg_read(urbr[2]);    // Read the buffer

  DPRINT("\nREAD VAL: %c\n", cread);

  m_reg_bfs(VICSoftIntClear, (1<<uirq[2]));   // Clear int flag in vic
  flag[2] = 1;
  wake_up_interruptible(&my_queue);   // Wake up from interrupt
  return IRQ_HANDLED;
}
```

The help function implemented gives the following input if it is called:

```
# ./uart help
Available commands:
  IOCTL_HELP : show different help commands
  IOCTL_DEFAULT : 8N1, 9600 baud
  IOCTL_BAUD : send argument between 2400 and 230400
  IOCTL_WORDLEN : send argument 5-8 wordlenght
  IOCTL_STOPBIT : send argument, 1 or 2 stop bits
  IOCTL_PARBIT : send arg, 0, 1(odd), 2(even), 3(Forced 1 stick), 4(Forced 0 stick
  IOCTL_FIFO : send arg 0 (off), 1 (on)
  IOCTL_FIFO_TRIG : send arg number to trig, 1, 4, 8 or 14 characters
```

6.5.3 Verification

According to the requirement F-1.2 and the test for it, the communication between two modules have been tested over power line.

ID	Requirement	Test Description	Grade/Comment
F-1.2	Communication - System	Connect a module to the hub, by connecting the module to the hubs power line (plugs on the back of the hub). If the module respond to a ping signal send from the hub, the two modules are communicating through power line. The response of the ping can be seen by using an oscilloscope and simply analyzing the packages on the power line according to the protocol.	PASSED

A small test setup have been made with a MBED device as one of the modules and an ARM board running uClinux as the other module. Data have been sent in both direction (from the MBED to the ARM and from the ARM to the MBED) through each of their Power Line modules, with use of the UART user space application written. The default settings of the Power Line module is 8N1 with a baud rate of 19200 which has been used for the test.
The test has passed. No invalid or missing data have been observed throughout the test.

6.5.4 Conclusion

The UART module is working with implemented IOCTL and interrupt handler. Further improvements before the fully functional energy system can use the UART driver is implementation of file write.

6.6 Deployment

Switch interrupt The switch block is the last part in the Spartan 6. The VHDL coding for the Spartan 6 is finished, the last part is to test it all together and verify that everything works.

Web Server The web server is up and running with the basic structure for the web interface development. The web server is able to send commands to the energy hub and it will route the commands to the desired module. For a complete system the applications running in background will be developed so all the communication can be handle.

UART Device driver improvements The UART device driver is working with implemented interrupt handling, input output control and clean write function. Klaus Kolle and Morten Opprud have been by and verified the test setup.

7 Time box 7

7.1 Time box planning

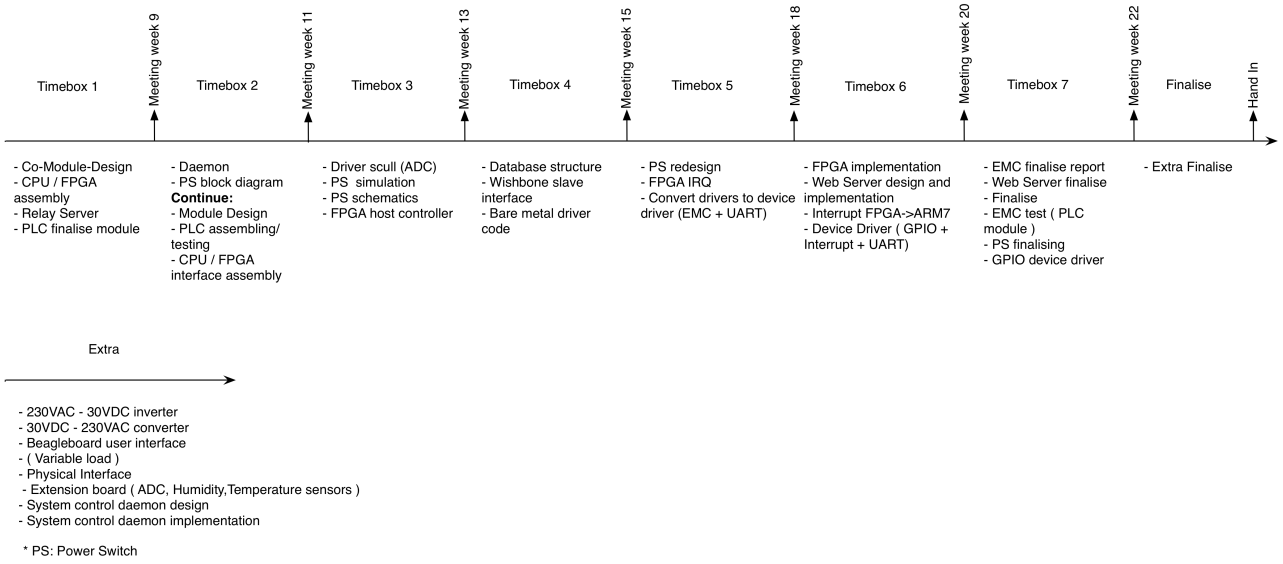


Figure 7.1: Updated time-box

7.1.1 Work to be done in this time box

- Requirements for debouncing
  - Stability of switches
  - Time limits
- Verification
  - External memory controller
  - Verification of the Spartan 6
- Power Switch system
  - Testing
- Web Interface
  - Database Corrections
  - Error Handling
  - User experience
- GPIO Device driver
  - Setup and control of GPIO pins used in the system

Description:

Requirements for debouncing This is the timing requirements for the switch block to avoid bouncing when switching state

Verification This is verification of the VHDL design including the EMC part in the LPC2478

Power Switch system Verification on the power switch system

Web Interface User web interface development.

GPIO device driver Driver to set and read the GPIO pins used in the energy HUB system.

7.1.2 Time planning

	Requirements for denouncing	Verification	Web Interface	Power Switch Verification	GPIO device driver
Estimation	3	7	35	3	10
Actual	3	15	40	5	14
Developer	Theis	Theis	Paulo	Paulo	Dennis

Table 7.1: Estimation and actual time used on the project

7.2 Requirements for debouncing - Theis

This section is requirement update for the Switch block from section 6.3.

7.2.1 Analysis

In order to figure out how long the switches is bouncing, some measurement has been made on the switches, while it is shifted. The result is shown below.

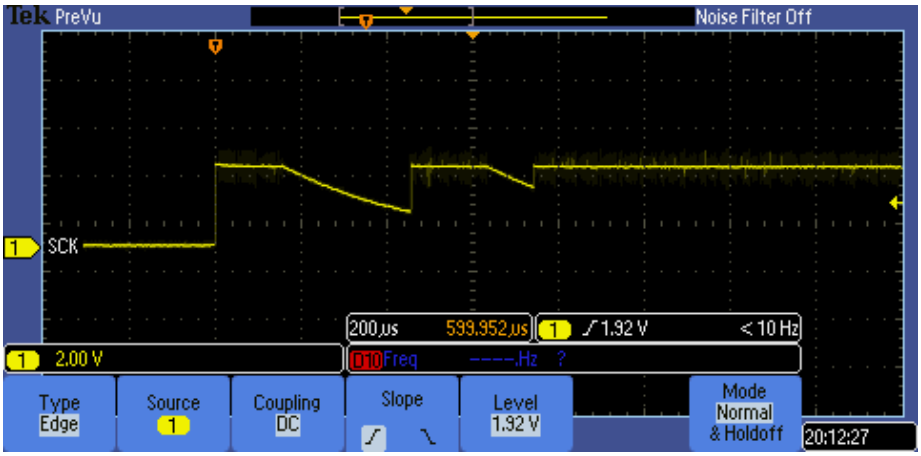


Figure 7.2: Signal from switch. Low to high

From the figure above it is clear that the switch is bouncing, and it is possible to see how long it takes the switch to be stable. From the measurements it takes around $750\mu s$.

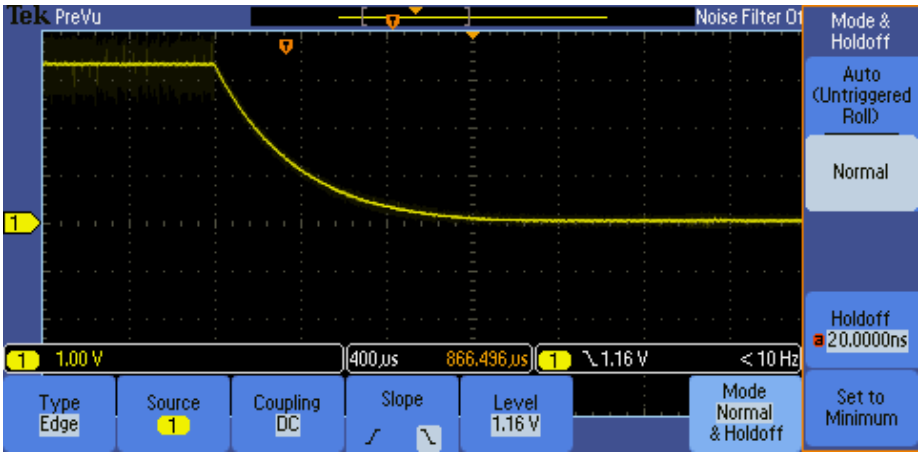


Figure 7.3: Signal from switch. High to low

The figure above shows the switch signal from high to low, this signal is now bouncing, and would not cause any problem because the slope is negative at all time, so when the signal hits the point where the Spartan 6 is switching, the signal would not trigger a switch back. But is takes around $1200\mu s$ for the switch to stabilise.

7.2.2 Design

In order to remove any debouncing on the switches, a delay is needed. In this system the time has to be the double plus one bit. From the measurements the longest time is $1200\mu s$, the double of that is $2400\mu s$. Calculations for the delay is shown below.

$$t = 1200\mu s \cdot 2 \tag{7.1}$$
$$f = 100MHz \tag{7.2}$$
$$t = 2^n \cdot \frac{1}{f} \tag{7.3}$$
$$n = \frac{\ln(f \cdot t)}{\ln(2)} \tag{7.4}$$
$$n = \frac{\ln(100MHz \cdot 2400\mu s)}{\ln(2)} \tag{7.5}$$
$$n = 17.87 \tag{7.6}$$

The bit length for the double delay is 18 and plus 1 bit, a 19 bit vector has to be used in the delay. The delay for 19 bit vector is.

$$t = 2^{19} \cdot \frac{1}{100MHz} \tag{7.7}$$
$$t = 5.243ms \tag{7.8}$$
$$\tag{7.9}$$

This is the time delay that is used to debounce the switches.

7.2.3 Conclusion

This time delay has been tested in timebox 6 and it is working with the interrupt register in the Spartan 6.

7.3 Verification - Theis

From the former timeboxes the blocks in the Spartan 6 has been implemented to make a complete system, which is communicating with the external memory controller in the LPC2478.

7.3.1 External memory controller

The EMC interface has been analysed with timings and signal assignment in order to make the Spartan 6 read and write data on the right time. On the figure below a block diagram for the EMC in the LPC2478 is shown. The data and pictures is taken from the *LPC24xx user manual*[22] and the *electrical datasheet*[23] for the LPC2478

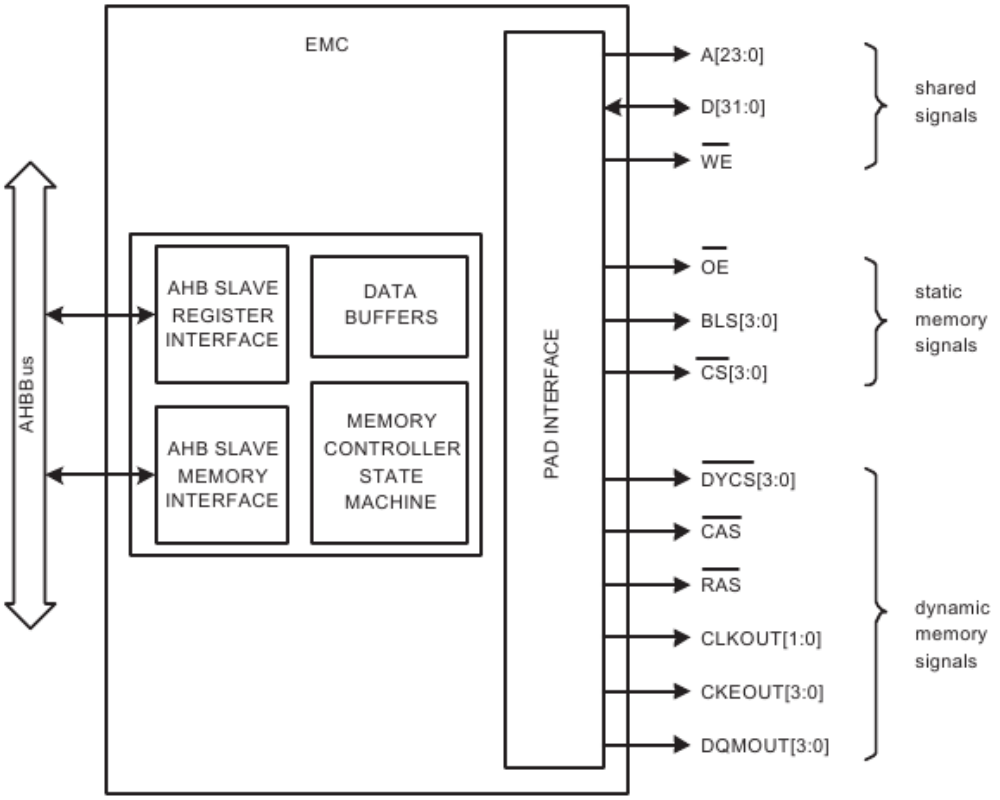


Figure 7.4: EMC block

The EMC is used in static mode without the *BLS*[24] signal, and for memory selection it is only using the *CS2*[25] for the Spartan 6, this is the interface that has to be implemented in the Spartan 6 to control the wishbone

[22] http://www.nxp.com/documents/user_manual/UM10237.pdf
[23] http://www.nxp.com/documents/data_sheet/LPC2478.pdf
[24] Byte lane selects
[25] Chip Select 2

master. The block that takes care of this assignment is the host controller from timebox 3. The timings and signal assignment is shown on the next two figures.

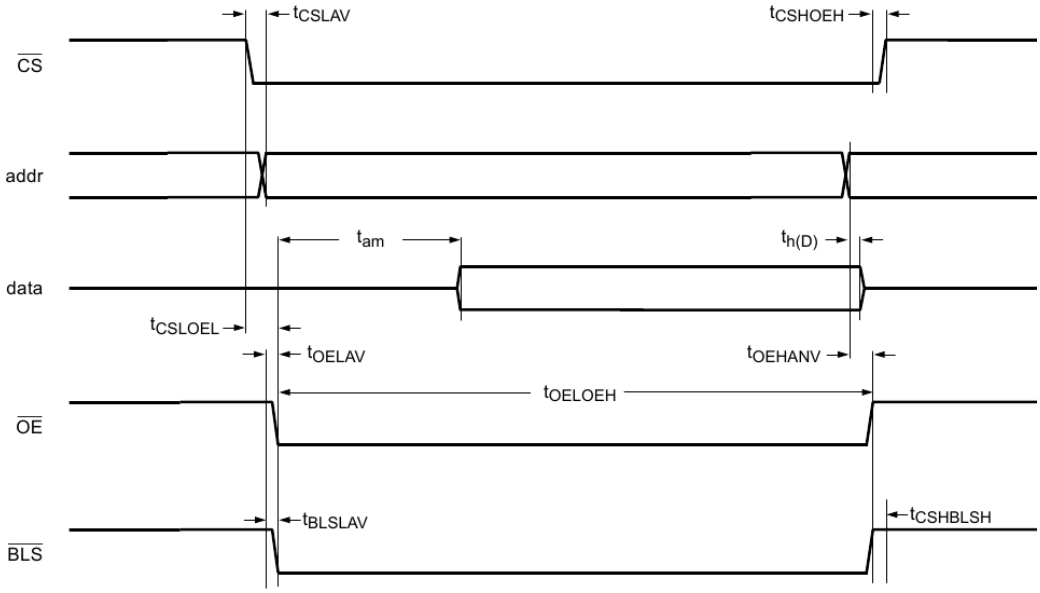


Figure 7.5: EMC Read timing

The important thing when the EMC reads data from the Spartan 6, is the t_{am}, which is where the EMC reads the data from the Spartan 6, therefore the Spartan 6 has to have valid data ready latest at that time, for the EMC to read correct data. From the electrical datasheet the timing for t_{am} is taken.

$$t_{am}min = (WAITRD - WAITOEN + 1) \cdot T_{cy} - 12.70ns \tag{7.10}$$
$$t_{am}typ = (WAITRD - WAITOEN + 1) \cdot T_{cy} - 9.57ns \tag{7.11}$$
$$t_{am}max = (WAITRD - WAITOEN + 1) \cdot T_{cy} - 8.11ns \tag{7.12}$$

The *WAITRD* and *WAITOEN* is integer values that is set in the setup of the EMC in the LPC2478, *WAITRD* has influence on the timing of t_{am} and *OELOEH* increasing this value increase the timing. When *WAITOEN* is increased *CSLOEL* is increased to and t_{am} and *OELOEH* is decreased. The T_{cy} is the time of the clock period.

$$CCLK = 72MHz \tag{7.13}$$
$$T_{cy} = \frac{1}{CCLK} \tag{7.14}$$
$$T_{cy} = 13.889ns \tag{7.15}$$

The exact timing of t_{am} is calculated late with the Bus functional module used for verification of the Spartan 6 design.

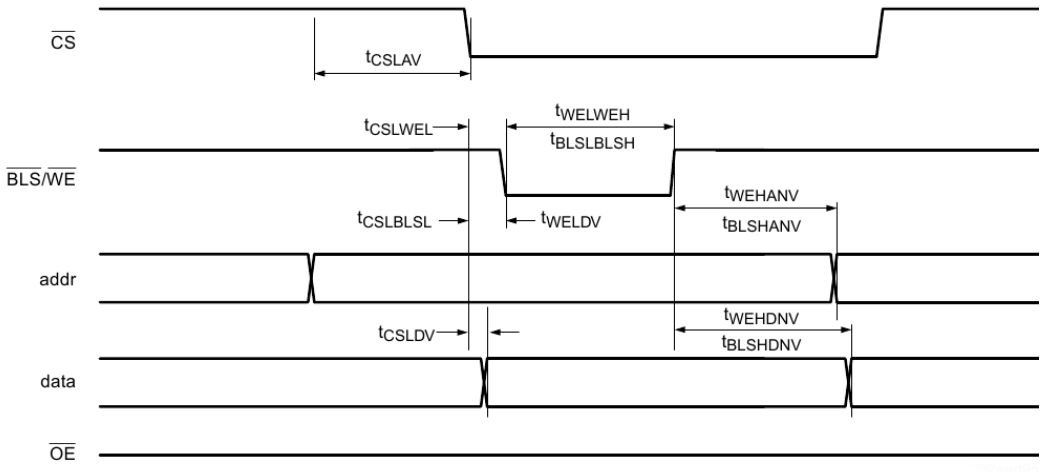


Figure 7.6: EMC Write timing

The write diagram shows that the data is valid at the same time as *WE*[26] is set low, as long as the Spartan 6 is reacting on *CS2* and the *WE* invalid data is not a problem.

7.3.2 Bus functional module

The bus functional module is a module written in VHDL for test purpose. A BFM is able to simulate different interfaces, depending on the code. In this project the BFM is used to test the total design of the Spartan 6 by simulating the EMC interface from the LPC2478. A rough code sketch was handed out from Morten Opprud Jakobsen, timing tweaks needed to be updated in order to make a correct simulation.

```
sim
├── arm_emc_package.vhd
├── log
│   └── arm7_bfm_log.txt
├── stim
│   └── arm7_bfm_stim.dat
├── tb_ARM_BFM.vhd
└── txt_util.vhd
```

The file structure of the BFM is shown above, in *arm_emc_package.vhd* all the timings and constants is contained. This file also holds the functions for a read and write cycle, with signal assignment in the correct order and time from the timings. *txt_util.vhd* holds function for converting text strings. The *tb_ARM_BFM.vhd* is the test bench file, this file takes a data input from the data file. The data file tells the BFM what to do, if it shall read, write, wait or log something, the logs is written to the log file. The BFM is used on the final Spartan 6 design with the following data file.

```
#WAIT 10
#LOG Write 0x00 to LEDs
#WR 0000000000010000 0000000000000000
#LOG Change switches
#SW 01000000
#WAIT 30
#LOG Read IRQ reg
#RD 0000000000000000
```

[26] Write Enable

```
#LOG Read switch reg
#RD 0000000000100000
#LOG Write 0x00 to LEDs
#WR 0000000000010000 0000000001000000
#WAIT 10
#END
```

Listing 1.1: BFM data file

The test bench signals is shown below. In the first figure, the BFM write to the LEDs to turn off, in the figure it is shown that the LEDs is changed to all zeros. Next the switches positions is changed, the arrow shows that the interrupt to the LPC2478 is activated after the switches is changed.

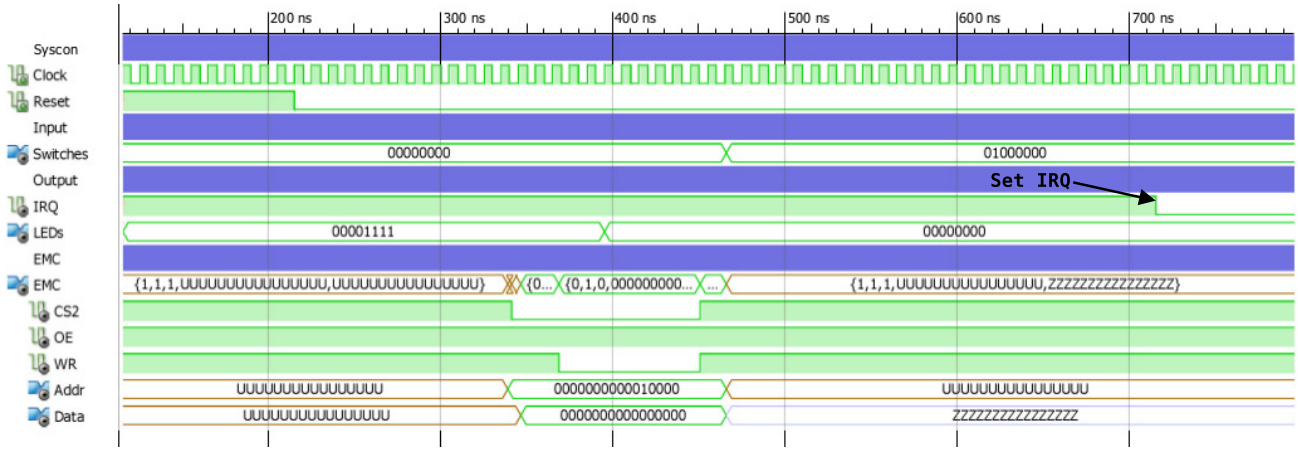


Figure 7.7: BFM write

The figure below is the continued BFM test bench. First the BFM reads the interrupt register, the first arrow shows when the data is valid on the Spartan 6, the blue maker indicate when the EMC is reading data from the Spartan 6. The second arrow shows that the interrupt is deactivated as the interrupt register is read. Next the BFM read the switch register to get the switch position, the arrow again shows when the data is valid, and the blue marker show when the data is read. Last the LEDs is set to the new switch position by another write to the LEDs from the BFM.

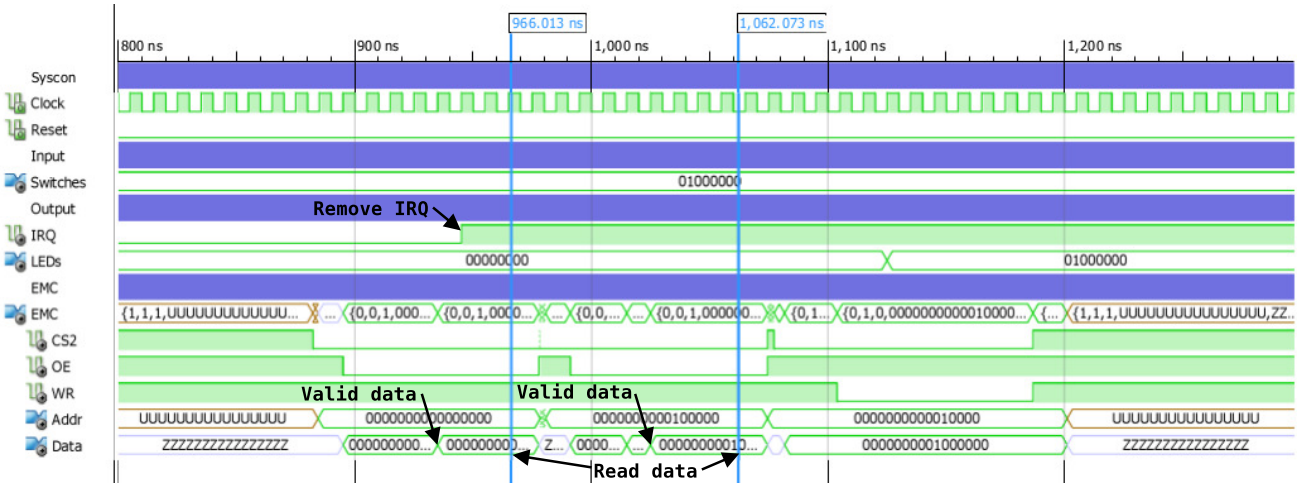


Figure 7.8: BFM read IRQ

7.3.3 Digital clock manager

In the code handed out by Morten Opprud Jakobsen, there is a DCM[27]. This module is used for making a jitter-free clock, and have the possibility to change the clock speed in order to match the clock for the LCP2478. The DCM in Spartan 6 tolerates two types of jitter, *cycle-to-cycle jitter* and *period jitter*. Cycle-to-cycle jitter is the change in clock period from cycle to cycle, the period jitter is the change over millions of clock cycles. The code for the DCM is shown below.

```
...
  clk_o <= clk_r;

-- DCM instantiation for the system clock.
  DCM_Sys : DCM_SP
  generic map (
    CLKFX_DIVIDE        => 2,                   -- Can be any integer from 1 to 32
    CLKFX_MULTIPLY      => 2)                   -- Can be any integer from 2 to 32
  port map  (
        CLK0        => clk_s,                   -- 0 degree DCM CLK ouptput
        CLKFX       => clk_r,                   -- DCM CLK synthesis out (M/D)
        CLKFB       => clk_s,                   -- DCM clock feedback
        CLKIN       => clk_i,                   -- Clock input (from IBUFG, BUFG or DCM)
        RST         => '0'                      -- DCM asynchronous reset input
        );
...
```

Listing 1.2: DCM_SP

The DCM in the Spartan 6 has many different features, one of the features that is used is removing jitter, this is done in the code above. The DCM takes the standard clock input, and the zero phase shifted clock is routed into the clock feedback, for the DCM to compensate for jitter. The DCM is also used to scale the clock to match the clock in the LPC2478, this is done by multiplying and dividing the clock with integer values.

7.3.4 Spartan 6

The final design of the Spartan 6 in this project has been verified and the figure below shows the top layer block, and the inputs and outputs on the Spartan 6.

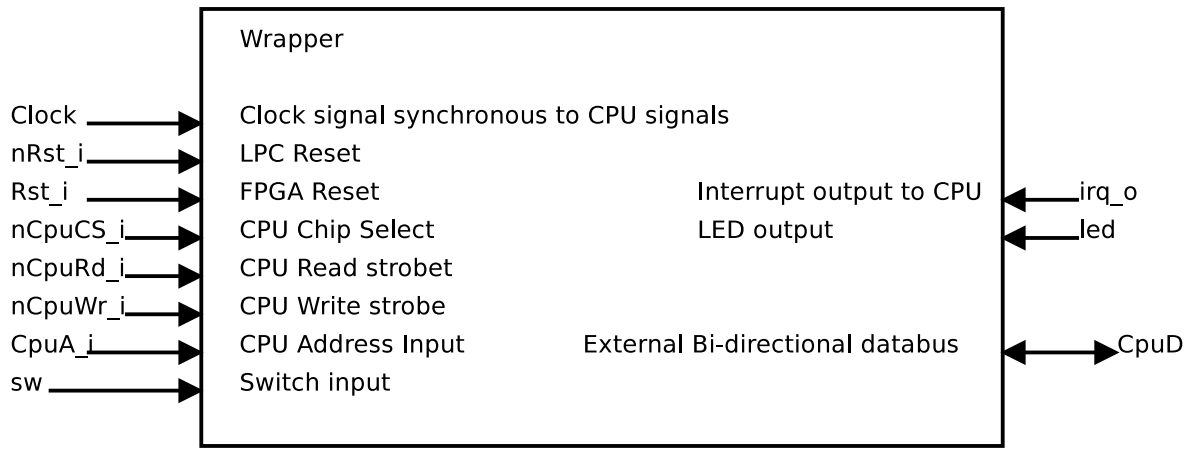


Figure 7.9: Top level in Spartan 6

The Spartan 6 acts as a memory block seen from the EMC in the LPC2478, the memory map for the Spartan 6 is shown below, with slave addresses, data direction and data length.

[27] Digital Clock Manager

Name	DIR	S select	S address	Data	EMC address	Note
IRQ Register	Read	000	0000	7 bit - 6 down to 0	0x82000000	Hold data from the interrupting slave
LED	Write	001	0000	8 bit - 7 down to 0	0x82000020	Set states of LEDs
Switch	Read	010	0000	8 bit - 7 down to 0	0x82000040	Hold state of switches

Table 7.2: Memory map of Spartan 6
S = Slave, DIR = Direction

7.3.5 Conclusion

The BFM verifies the design. Afterward the design has been tested on the hardware and it is working properly. It is not possible to measure the clock speed with the equipment at the university, so it is not possible to verify that the code fails without the DCM because of jitter on the clock.

7.4 Power Switch System- Paulo Fontes

The power switch system is the main functionality of the energy hub, it have to be able to switch the energy flow direction and turn off the port if needed. A prototype PCB was designed in time box 5, in this time box the verifications are made and the PCB is improved according to the results.

Verifications:

ID	Description	Verification
1	Switch the voltage as input or output	Using an mBed to drive the input pins from the power switch system high or low.
2	Over voltage and under voltage are kept between 30V ±10%	Connect two 30V power supplies in series and decrease the voltage below 27V, and increase again to 30V. Increase the voltage above 33V, and decrease to 30V again.
3	Current between maximum 30A	Not tested
4	Current Sensor giving feedback voltage	The current sensor feedback pin is connected to the analogue input in the mBed, the value is shown by the serial connection.

Verification 3 is not performed since this board is a prototype, the header pins soldered and the tracks are not consistent to such current and this test could damage the components in the board. For this requirement to be fulfilled the tracks on the PCB should be increase and connectors that could support high currents have to be used.

7.4.1 Verification

The verifications are done following the numeration in ID.

ID	Description	Verification
1	Switch the voltage as input or output	Using an mBed to drive the input pins from the power switch system high or low.

Verification ID: 1 The mBed is used to simulate the control system, it will set the input pins from the power switch module high and low according to the commands given to the mBed by serial connection.

Test bench set up: An variable power supply (0-60V) with max. 1A is used as supply to the system, the mBed will simulate the control system of the power switch system and the voltmeters will show the energy flow direction.

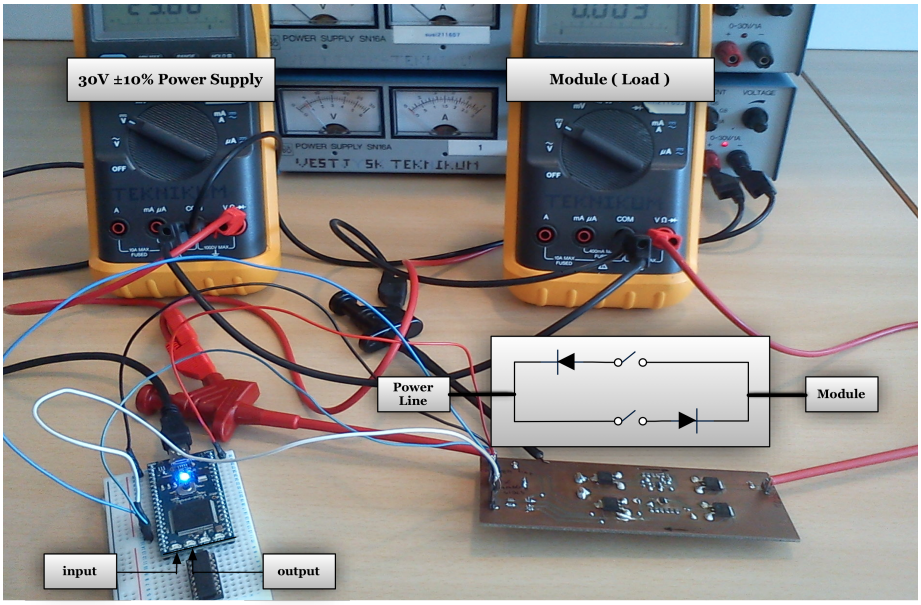


Figure 7.10: Test bench set up

The code implemented in the mBed uses 3 pins, a regulated 5V is supplied to the current sensor and the USB serial connection is used for the communication. Pins 13 and 14 are set as digital outputs while pin 15 is set as analogue input for the current sensor measure.

Case scenario 1: Setting the output pin as high, will drive the SHTD pin high, closing the circuit. The energy will flow from the power line (source) to the module (load). Using a voltmeter the input voltage and output voltage is measure and a drop of 10mV. The results are as expected.

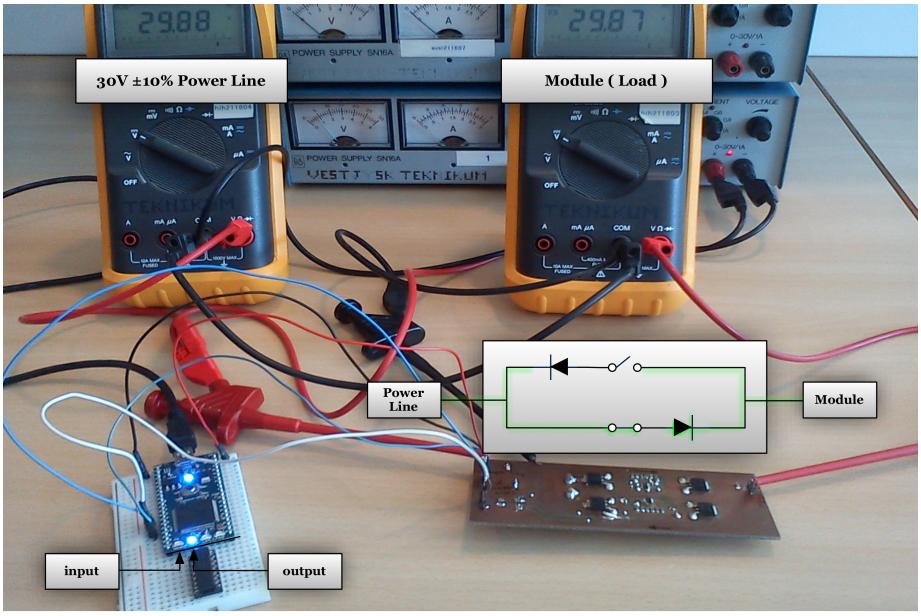


Figure 7.11: Case scenario 1: Energy flow from power line to the module

Case scenario 2: Changing, the input pin to high and the output pin low, energy will not flow in any direction since the circuit will be open from source to load and two MOSFETs back to back will work as a diode. The results are as expected.

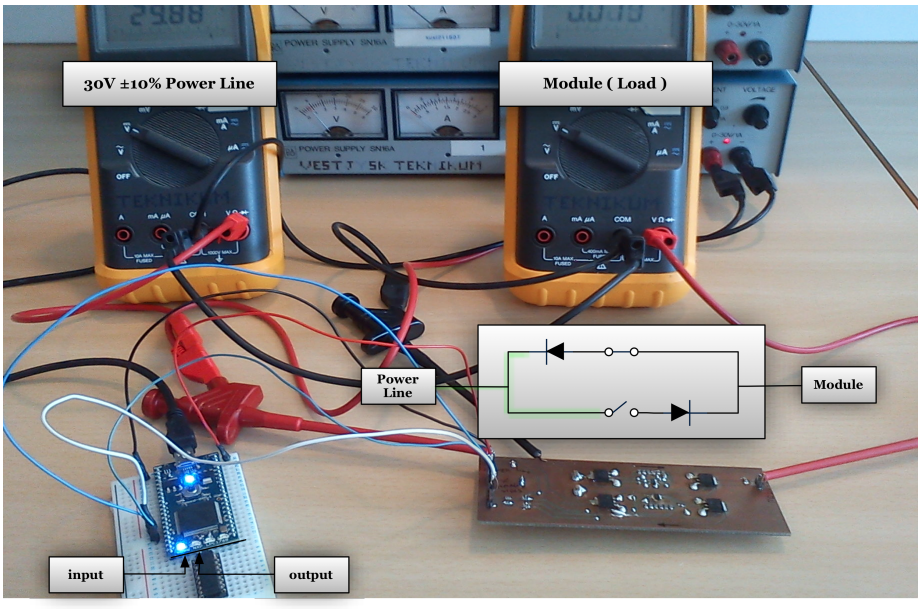


Figure 7.12: Case scenario 2: Open circuit energy is blocked by the back to back MOSFETs working as diode.

Both ports are tested and the system works as expected. The tests show that the system have a low voltage drop across the MOSFETs but still working as a diode. The direction of the energy flow can be changed without problems. The verification is successful.

Verification ID: 2 In this verification both ends of the systems are tested for over and under voltage. The power supply is connected and the voltage in increased until 35V, after its estabelized again at 30V and decreased to 25V. The power switch system should shut down the port if over 33V and under 25V.

ID	Description	Verification
2	Over voltage and under voltage are kept between 30V ±10%	Connecting two 30V power supplies in series and decrease the voltage below 27V and increase again to 30V until stabilise. Increase the voltage above 33V, and decrease to 30V again.

The test bench used is the same as in the earlier verification.

Case scenario 1: The power supply is connected to the power line, the voltage is increased to 34V.

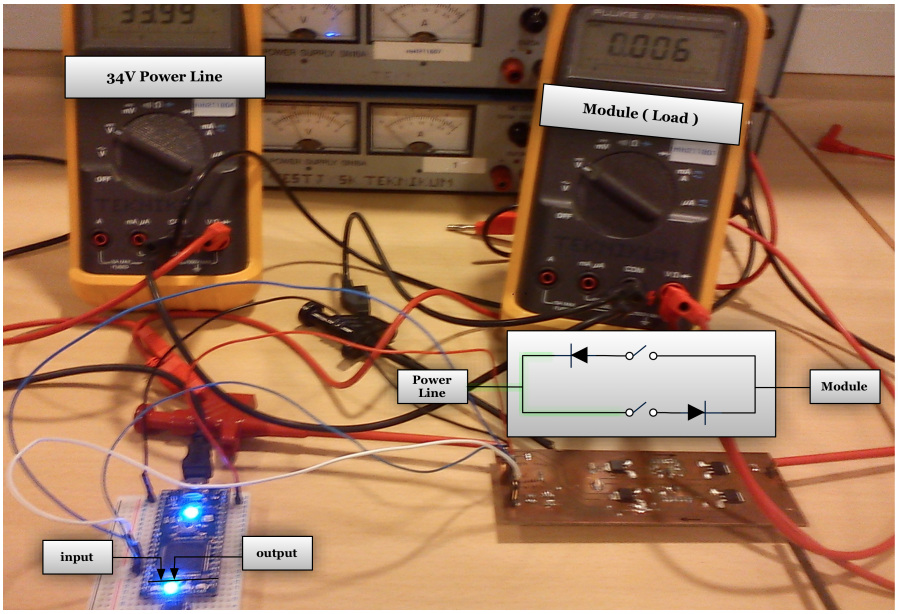


Figure 7.13: Case scenario 1: Voltage above 33V will shut down the port.

Case scenario 2: The power supply is connected to the power line, the voltage is decrease to 26V.

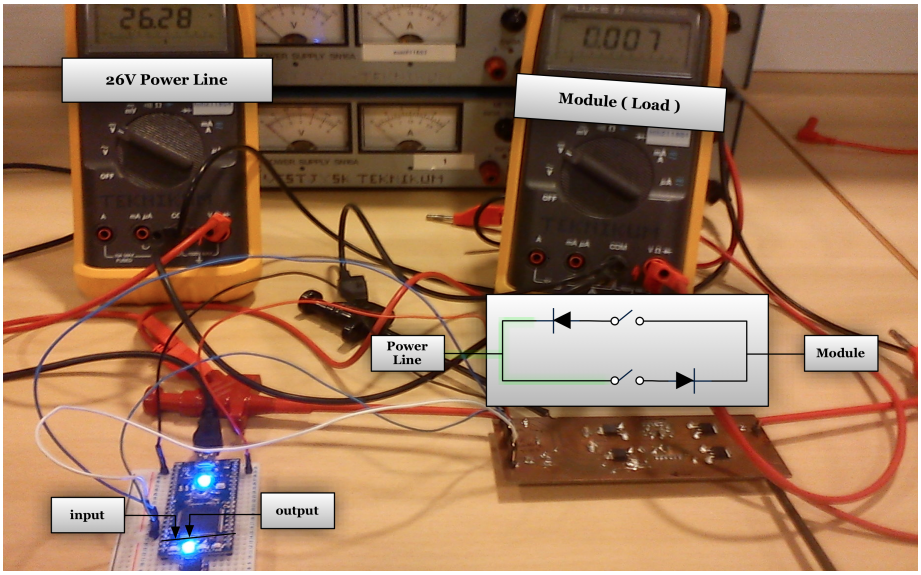


Figure 7.14: Case scenario 2: Voltage below 27V will shut down the port.

The results were the same in both ports, this verification shows that the system have a low and high voltage protection as requested in common requirements. The under voltage protection is unnecessary and might even give malfunctions to the system, since the system might be running only on the power provided by a battery module at the time the voltage goes under 27V, this will shut down the port and the system will be disconnected from any power source.

Verification ID: 4 The current sensor implemented in the power switch system is used to retrieve data regarding the direction of the flow and for security reasons turn of the port in case the current is too high. For this test a variable load is added to the test bench and set to 30ω. The current sensor implemented gives a feedback of 40mV/A.

4	Current Sensor giving feedback voltage	The current sensor feedback pin is connected to the analogue input in the mBed, the value is shown by the serial connection.

Case scenario 1: The power supply is connected to the power line and a variable load as module. The resistance is set to 30ω giving a difference of approximately 40mV in the feedback pin.

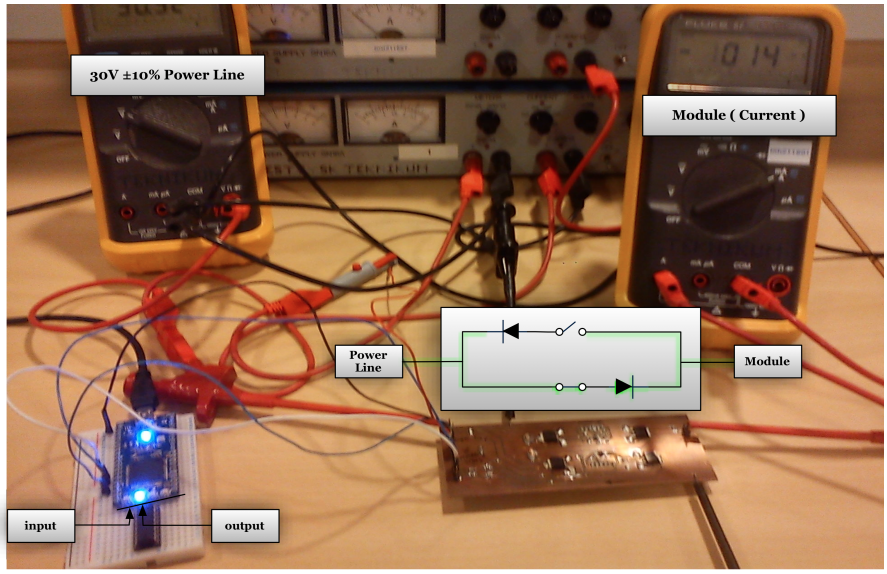


Figure 7.15: Case scenario 1: Current measures.

Due to the sensitivity of the current sensor, the accuracy can change according to the temperature and voltage supplied. Improvements regarding the accuracy have to be done.

7.5 Web Interface - Paulo Fontes

For the development and evaluative procedures, a LAMP(Linux Apache MySQL PHP) web server is set up, this process is described in time box 4. The web interface can be accessed on the university network at the address http://10.1.18.223/.

The web interface application includes a great amount of different scripting languages such as server side PHP and client side JavaScript, along with markup languages HTML and XML. The code for each will not be explained extensively, instead the most important functionalities will be filtered and explained.

An extra web application is developed in PHP only for evaluative propose, it allows the user to view the code of all the files on the web server file structure. By this method the last version of the code is always updated for visualization. The application is called SeeIt and can be accessed at the address: http://10.1.18.223/SeeIt/.

7.5.1 Analysis

Database For a well constructed database, it has to be consistent, flexible and efficient so no data is lost or repeated when saved. The data base structure has changed since time box 4 when it was developed. The new structure can be seen bellow.
TYPE(ID_TYPE, NAME);

ID_TYPE	Auto increment integer, a new id number is generated when a different type is needed to the system.
NAME	Describe the type name for example: Input, output, bidirectional, etc.

STATUS(ID_STATUS, NAME);

ID_STATUS	Auto increment integer, a new id number is generated when a different status is needed to the system.
NAME	Describe the status name for example: Running, stopped, warning, etc.

MODULES(ID_MODULE, ID_TYPE, NAME, HUB_PORT);

ID_MODULE	Module unique ID, this id as primary key ensures that no module with is repeated in the database.
ID_TYPE	This is a foreigner key for the table type, this way if some other type of module is needed it can be dynamical add.
NAME	The name of the module for example, Solar Panel, wind turbine, battery.
HUB_PORT	Actual connected port for this module

As a requirement, the database have to store the measurement from different sensors. This is stored in a the table MEASUREMENTS.
MEASUREMENTS(ID_MEASURE, ID_SENSOR, DATE_TIME, HUB_PORT, VALUE)

ID_MEASURE	Measurement id, auto increment field.
ID_SENSOR	This is a foreigner key for the table sensor, this associated the value retrieved to the correspondent sensor.
DATE_TIME	Date and time of the measurement is saved so it can be plotted or in case of a lower efficiency a detailed history can analysed.
HUB_PORT	Actual connected port for this module
VALUE	Measurement retrieved by the energy hub.

Logs are a simplified method of keeping track about what is happening in the system, this is one of the most fundamental functionalities of the system.
LOGS(ID_LOG, ID_MODULE, DATE_TIME, ID_STATUS, ID_USER, ID_ERROR);

ID_LOG	Auto increment integer, a new id number is generated every time a measurement is add.
ID_MODULE	This is a foreigner key for the table modules, this allow the system to know from which module correspond the measurement .
DATE_TIME	Date and time of the log is saved, in case of malfunction it will help with a time line.
ID_STATUS	The new status of the module that was changed.
ID_USER	Foreigner key for the table users, this will allow the system to know which user made the change.
ID_ERROR	Foreigner key for the table error, if any error occurs it will be stored.

For error handling situations a table ERRORS is created this way the administrator can have more control and act in case some mall function of the system.

ERRORS(ID_ERROR, NAME);

ID_ERROR	Auto increment integer, a new id number is generated when a different unit is needed to the system.
NAME	Predefined error description.

Measurements are add to the database constantly, which in a non-stop system database size could be a problem, to avoid this situation a MERGES data set is created. This will store an average of values between a time span for each sensor, allowing the customer either to erase the values from the log or export them out from the database.

MERGERS(ID_MERGE, DATE_FROM, DATE_TO, ID_SENSOR, VALUE);

ID_MERGE	Auto increment integer, a new id number is generated every time an wrap is needed.
ID_SENSOR	This is a foreigner key for the table sensors, this allow the system to know from which sensor correspond the values .
DATE_FROM	Date and time of the time span start.
DATE_TO	Date and time of the time span end.
VALUE	Average measurements value.

SENSOR(ID_SENSOR, ID_MODULE, ID_UNITS);

ID_SENSOR	Auto increment integer, a new id number is generated when a different sensor is needed to the system.
ID_MODULE	This is a foreigner key for the table sensors, this allow the system to know from which module correspond the sensor, being a primary key with the id_sensor, this way each sensor correspond to only one module .
ID_UNITS	This is a foreigner key for the table units, this allow the system to know which units the sensor is measuring.

UNITS(ID_UNIT, NAME);

ID_UNIT	Auto increment integer, a new id number is generated when a different unit is needed to the system.
NAME	Describe the unit name for example: A, V, deg ,m/s,etc. (Ampere, Volt, Degrees, Velocity)

To increase security, an USERS table is added associated with different PRIVILEGES for different users. A user root is created leaving the system able to manage multiple users with different privileges if needed.

USERS(ID_USER, NAME, PASS, EMAIL, ID_PRIV);

ID_USER	Auto increment integer, a new id number is generated when a different privilege is needed to the system.
NAME	Username credential.
PASS	User password (Not encrypted in this version).
EMAIL	Email to which warnings are send.
ID_PRIV	Foreigner key for the table privileges.

PRIVILEGES(ID_PRIV, NAME);

ID_UNIT	Auto increment integer, a new id number is generated when a different privilege is needed to the system.
NAME	Describe the user privileges.

For the communication process between the web interface and the energy hub, the ip address of the device need to be always accessible this way a table DEVICES is created in which the last known IP address of the device is stored.

DEVICES(ID_DEVICE, IP);

ID_DEVICE	Auto increment integer, a new id number is generated when a different ip added.
IP	IP address of the device.

At this point, in an abstract way, the database can initialise a new module or identify if the module where connected before, change the status for each module and add new measurements for each sensor.

Data Model Data model is a high-level overview of the database structure. At this stage data sets are translated to a logic structure with the relationship between tables.

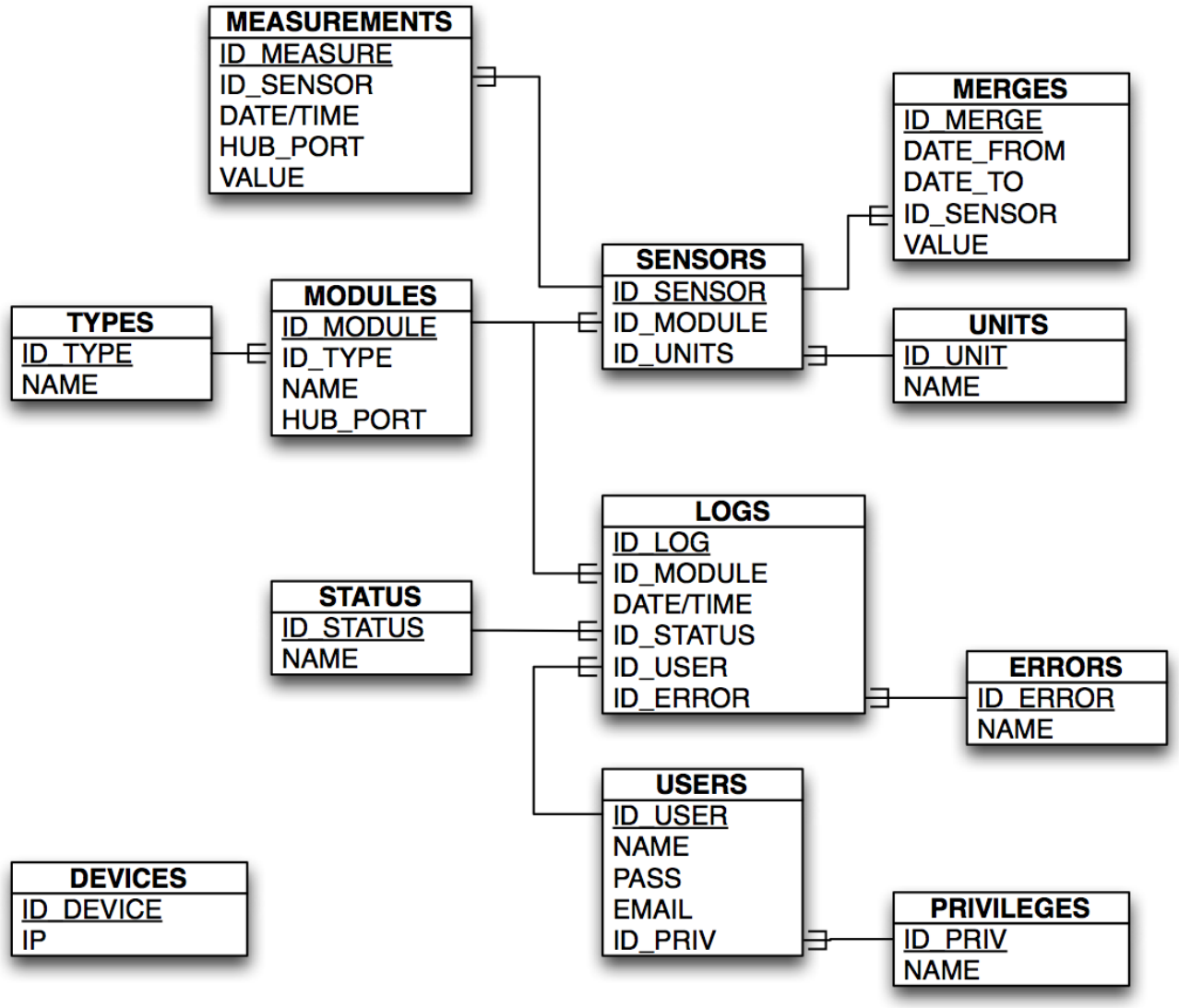


Figure 7.16: Database relational overview (Data Model)

getData.php This script is described in this section since its main functionality is to retrieve values from the database using SQL queries and construct a XML file with the data retrieved. Only the Ampere measurements are requested and saved into this database since this is prototype of the system the necessary protocol and structure for the sensors in the modules was not defined.
Script parameters:

- unique_id - this is the only parameter expected by this script, since it will retrieve all the measurements from an unique module/sensor, the URL encoding method for this parameter have to be POST.

Several requests to the database are done by this script, each SQL query and is described bellow:
Current power production of the modules:

```
// Current Power Production and Current Power Consuption
  $sql = "SELECT  VALUE FROM `MEASUREMENTS` WHERE `ID_SENSOR`=".$_POST['unique_id']." ORDER BY ID_MEASURE
      DESC LIMIT 1";
  $res = $con->query($sql);
  $row = $res->fetch_row();
```

```
5   if($row[0]>0){
6     $c_pw_pro = ($row[0]*$voltage)."W ";
7     $c_pw_con = "0W ";
8   } else if ($row[0]<0){
9     $c_pw_con = ($row[0]*$voltage*-1)."W ";
10    $c_pw_pro = "0W ";
11  } else {
12    $c_pw_pro = "0W ";
13    $c_pw_con = "0W ";
14  }
```

The SQL query, uses the keyword **SELECT** to show the value of the column **VALUE**, **FROM** the table **MEASUREMENTS**, **WHERE** the field **ID_SENSOR** match with the **unique_id** send by parameter, the results are **ORDER BY** the column **ID_MEASURE**, from the last inserted value to the first(**DESC**), with a **LIMIT** of **1** result.
The result of this query is the last data entry in the table. If the value is negative it will be multiplied by the voltage and -1, this is the current power consumption. If positive it will be multiplied by the voltage and assigned to the current power production.
Total Power Production:

```
1   $sql = 'SELECT SUM(VALUE)
2       FROM `MEASUREMENTS`
3       WHERE `ID_SENSOR`='.$_POST['unique_id'].'
4       AND DATE_TIME>(SELECT DATE_TIME FROM LOGS WHERE ID_MODULE='.$_POST['unique_id'].' ORDER BY DATE_TIME
            DESC LIMIT 1)';
5   $res = $con->query($sql);
6   $row = $res->fetch_row();
7   $tt_pw_pro = ($row[0]*$voltage)."W ";
```

The SQL query, uses the keyword **SELECT** to retrieve the addition (**SUM**) of the values from the column **VALUE**, **FROM** the table **MEASUREMENTS**, **WHERE** the field **ID_SENSOR** match with the **unique_id** send by parameter, AND the columns DATE_TIME is greater than the result from the **SELECT** value of the field **DATE_TIME FROM** table LOGS **WHERE** the **ID_SENSOR** match with the **unique_id** send by parameter, this result is **ORDER BY** the column **DATE_TIME**, from the last inserted value to the first(**DESC**), with a **LIMIT** of **1** result.
This query returns the addition of all data in column VALUE from the table measurements after the date and time of the last log entry. This way the total power production seen in the user interface corresponds to the uptime of the module/sensor.
Average power production:

```
1   // Average Power Production
2   $sql = 'SELECT AVG(VALUE)
3       FROM `MEASUREMENTS`
4       WHERE `ID_SENSOR`='.$_POST['unique_id'].'
5       AND DATE_TIME>(SELECT DATE_TIME FROM LOGS WHERE ID_MODULE='.$_POST['unique_id'].' ORDER BY DATE_TIME
            DESC LIMIT 1)';
6   $res = $con->query($sql);
7   $row = $res->fetch_row();
8   $avg_pw_pro = ($row[0]*30)."W ";
```

This query is similar to the total power production, instead of the addition of the data in the column VALUES the average is made. This will give the average of values since the last data entry in the table logs.
Module status:

```
1   // Status
2   if($_SESSION['error_device']==1){
3     $stat = "Connection Failed";
4   } else {
5
6   $sql = 'SELECT `STATUS`.NAME, `STATUS`.ID_STATUS
7       FROM STATUS
```

```
8       INNER JOIN LOGS ON `STATUS`.ID_STATUS = `LOGS`.ID_STATUS
9       WHERE `ID_MODULE`='.$_POST['unique_id'].'
10      ORDER BY `LOGS`.ID_LOG DESC
11      LIMIT 1';
12
13  $res = $con->query($sql);
14  $row = $res->fetch_row();
15
16  if($row[0]==""){
17    $stat = "Disconnected";
18    $id_stat = 2;
19  } else {
20    $stat = ($row[0]);
21    $id_stat = $row[1];
22  }
23
24  if($row[1]!=3){
25      $uptime = " --- ";
26      $c_pw_pro = " --- ";
27      $c_pw_con = " --- ";
28      $tt_pw_pro = " --- ";
29      $avg_pw_pro = " --- ";
30  } ...
```

The SQL query, uses the keyword **SELECT** to show the values of the columns **NAME** and **ID_STATUS**, **FROM** table **STATUS**, when joined (**INNER JOIN ON**) with table **LOGS** where the **STATUS.ID_STATUS** match with **LOGS.ID_STATUS**, and **WHERE** the field **ID_MODULE** match with the **unique_id** send by parameter, the results are **ORDER BY** the column **ID_LOG**, from the last inserted value to the first(**DESC**), with a **LIMIT** of **1** result.
This query returns the status name and id for the desired module, the returned values are assigned to the variable stat and id_stat for later use in this script. In case the status id is different then 3, the module will not be running, so no values are shown to the user.
Create the XML result:

```
1   echo '<DATA>'.
2        '<C_PW_PRO>'.$c_pw_pro.'</C_PW_PRO>'.
3        '<C_PW_CON>'.$c_pw_con.'</C_PW_CON>'.
4        '<TT_PW_PRO>'.$tt_pw_pro.'</TT_PW_PRO>'.
5        '<AVG_PW_PRO>'.$avg_pw_pro.'</AVG_PW_PRO>'.
6        '<UPTIME>'.$uptime.'</UPTIME>'.
7        '<STAT id_stat="'.$id_stat.'">'.$stat.'</STAT>'.
8        '<M1 id="'.$m_id[1].'" name="'.$m_name[1].'" status="'.$m_status[1].'" id_status="'.$m_id_status[1].'
             "></M1>'.
9        '<M2 id="'.$m_id[2].'" name="'.$m_name[2].'" status="'.$m_status[2].'" id_status="'.$m_id_status[2].'
             "></M2>'.
10       '<PRIV>'.$priv.'</PRIV>'.
11       '<DEV_ERROR>'.$_SESSION['error_device'].'</DEV_ERROR>'.
12      '</DATA>';
```

XML as HTML is a markup language, using elements defined by tags. JavaScript is able to translate XML files with built-in objects, this allows the return of a large amount of data at once, reducing the number of scripting needed and requests to the web server.
The construction of the XML file can be seen above, the main element is DATA, having several 'child nodes'. This nodes (like HTML) can have several attributes, for example the element M1 and M2 contain attributes that defines the modules connected to the port 1 and 2 of the energy hub.

File Structure The file structure defines how files are grouped on a system, in a web server architecture the files are stored in the root directory defined by the web server (Apache) configuration. The file structure necessary for this web interface has change since last time box due to a meeting with other teams. The above structure was approved from the meeting.

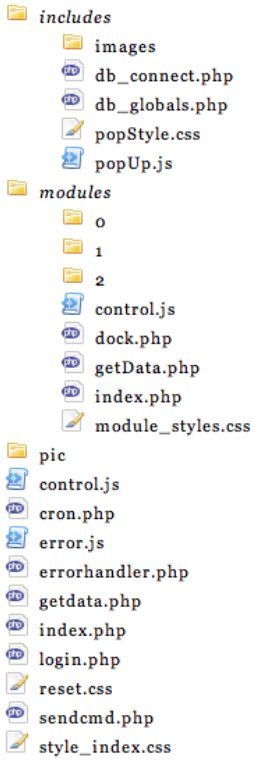


Figure 7.17: Proposed File structure

A short description for each script listed above can be seen in this list:

- includes/images/ - Contains images related with the system functionalities.
- includes/db_connect.php - Handle the connection to the MySQL database.
- includes/db_globals.php - Includes all the global variables with the credentials for the database.
- includes/popStyle.css - Popup login style definition.
- includes/popUp.js - Client side script to handle the user login.
- modules/[0-2] - Modules pages, the numbers are associated with the modules ids, being 0 the hub, 1 the battery and 2 the photovoltaic panels.
- modules/control.js - Client side script, makes background requests to a PHP script, this script is part of the user experience optimization.
- modules/dock.php - PHP script contains the navigation system for the web interface.
- modules/getdata.php - PHP script called in background by the client side script running recursively control.js, this script retrieves the system status and data form database in real time, is part of the user experience optimization.
- modules/index.php - This page includes each module page structure.
- modules/module_styles.css - Graphical layout definition.

- control.js - Client side script, makes background requests to a PHP script and handle the connections of new modules, this script is part of the user experience optimization.
- cron.php - Cron jobs are running by the server in a predefined time, in this case the cron.php at the web server root will point to the cron jobs inside each module folder.
- error.js - Client side script, make background requests to a PHP script part of the error handling system.
- errorhandler.php - PHP script called in background by the client side script running recursively, is part of the error handling system.
- getdata.php - PHP script called in background by the client side script running recursively control.js, this script retrieves the modules connected to the system in real time. Is part of the user experience optimization.
- index.php - First page of the web interface.
- login.php - This PHP script is called in background by the pop up login client side script.
- reset.css - Layout definition of the web interface.
- sendcmd.php - Webservice to send commands to the desired module/enegy hub.
- style_index.css - Layout definition of the web interface.

Due to the size of such system and the few time for the development, the project was scaled down so a high fidelity prototype where the system to be can be implemented and the main functionalities of the system tested.

Error Handling The web server have to be able to handle errors such the lost of communication with the energy hub or database connection problems. For a reliable system, a script running on the client side using JavaScript have to be implemented, this way the web interface can recursively test the communication to the energy hub and database.

Is necessary to define the web interface response for both situations database and energy hub connection lost. If the connection is lost to the device the user is able to see the data stored in the database, so the system is not blocked, but a warning is shown to the user/administrator, do the correspondent debug can be done. In case the connection to the database is lost, the user interface is blocked with an warning, the energy hub will continue working, since it doesn't affected the correct work of the system to route the energy, in this situation all the data retrieved by the modules will be lost.

Since a script is running recursively at the client side, when the communications are re-establish to the energy hub or database the user will be able to use the system normally, this is a great improvement in the user experience and functionalities as an web application.

7.5.2 Implementation

The development of the web interface is made in a local machine using MAMP (MAC Apache MySQL PHP), and transfer later on to a web server on the university network.

Communication The communication between the several components of the system is crucial, data is retrieved at a high rate from the modules, so an effective communication between the energy hub and the web server is implemented. Among the communication with the energy hub, the web interface needs a constant connection with the MySQL server, so data can added and retrieved from the database.

savedata.php A background application running in the energy hub converts the measurement sent through PLC (UART) to a HTTP request **savedata.php?sensor_id=ID Sensor&value=Sensor Measurement&hub_port=Hub Port**.
Script parameters:

- sensor_id - Id of the sensor to where save data.
- value - the vlue to be saved
- hub_port - the connected port of the module
- op - status, change status of a module. ¡unique_id¿¡new status¿, this is added to the log table.

```
<?php
  function changeStatus($con){
    $date = getdate();
    $today = $date['year'].'-'.$date['mon'].'-'.$date['mday'].' '.$date['hours'].':'.$date['minutes'].':'.
        $date['seconds'];
    $sql = 'INSERT INTO `LOGS` (
        `ID_LOG` ,
        `ID_MODULE` ,
        `DATE_TIME` ,
        `ID_STATUS` ,
        `ID_USER` ,
        `ID_ERROR`
        )
        VALUES (
        NULL , \''.$_GET['id_module'].'\', \''.$today.'\', \''.$_GET['id_status'].'\', \''.$_SESSION['id_user
            '].'\', \'1\'
        )';
    if($con->query($sql)){
      echo "Success";
    }
    $sql = 'UPDATE `MODULES` SET `HUB_PORT` = '.$_GET['hub_port'].' WHERE `MODULES`.`UNIQUE_ID` ='.$_GET['
        id_module'];

    if($con->query($sql)){
      echo "Success";
    }
  }

  if(isset($_GET['op'])){

    switch ($_GET['op']){
      case "status": changeStatus($con); break;
    }

  } else {
    $date = getdate();

    $today = $date['year'].'-'.$date['mon'].'-'.$date['mday'].' '.$date['hours'].':'.$date['minutes'].':'.
        $date['seconds'];

    if(isset($_GET['sensor_id']) && isset($_GET['value']) && isset($_GET['hub_port'])){
      $sql= "INSERT INTO `MEASUREMENTS` (".
          "`ID_MEASURE` ,".
            "`ID_SENSOR` ,".
            "`DATE_TIME` ,".
            "`HUB_PORT` ,".
            "`VALUE`) ".
            "VALUES (NULL , '".$_GET['sensor_id']."', '".$today."', '".$_GET['hub_port']."', '".$_GET['value'
                ]."')";

      $con->query($sql); // Run the query in the MySQL server
      echo $sql;
    } else {
      echo 'Invalide parameters';
    }
  }

?>
```

This script saves the measurement retrieved from a sensor to the database. At first it established the connection to the MySQL server so SQL requests can be made. A PHP function returns an array with the current date and time, this is formatted into YYYY-M-D H:M:S, after it can be saved to the MEASUREMENTS table. The script collects all the parameters send through the URL encoding GET, a SQL code is generated and a request made to the MySQL server, the data is added to the MEASUREMENTS table.

In case of status changes, a parameter **op** is encode with the method GET and if it value is 'status' the function changeStatus() will be called. This function will update the HUB_PORT on the table MODULES and insert a new log to the table LOGS. The request for this functionality is **savedata.php?op=status&id_module=MODULE ID&id_status=STATUS TO CHANDE TO&hub_port=HUB_PORT**.

sendcmd.php The web interface is able to send commands to the energy hub and the modules using the **sendcmd.php?id_module=<module id>&cmd=<command to be send>**. The id_module tells the hub which module the command should be send. No verification is made of the send command by the web server or the energy hub unless the command is specifically send to the hub.

```
1
2  <?php
3
4    // SQL request for the device page.
5    require_once("includes/db_connect.php");
6
7    if(isset($_GET['cmd']) && isset($_GET['id_module'])){
8
9      $device_port = 5555;
10
11     $sql= "SELECT IP ".
12         "FROM `DEVICE` ".
13         "ORDER BY ID_DEVICE DESC ".
14         "LIMIT 1";
15
16     $res = $con->query($sql); // Run the query in the MySQL server
17
18     $row = $res->fetch_row();
19
20     $device_ip = $row[0];
21
22     if (!$socket=socket_create(AF_INET, SOCK_STREAM, SOL_TCP)){
23       exit (socket_strerror(socket_last_error()));
24     }
25
26     if (!socket_connect($socket,$device_ip,$device_port)){
27       exit (socket_strerror(socket_last_error()));
28     }
29
30     $str = $_GET['cmd'].';'.$_GET['id_module'].';'.$_GET['dir'];
31
32     socket_write($socket,$str);
33
34     $msg='';
35     $c='';
36
37     while(socket_recv($socket, $c, 256,0)){
38         if($c != null) {
39           $msg .= $c;
40       }
41     }
42
43     echo $msg;
44
45     socket_close($socket);
46
47   } else {
48     echo 'Command or Module Id not set';
49   }
50
51 ?>
```

At first the script will get the ip address of the energy hub, a SQL query retrieves the last IP address added to the table DEVICES. With the energy hub ip and a predefined port, a connection is created using a TCP socket for the communication between the energy hub and the web server. The parameters are collected from the encode URL and translated to a recognized format in the energy hub. The data is send to the energy hub and a answer of success or not is returned.

This script is called in background by a client side script, this will handle how to warning the user in case a problem occurs.

saveip.php This script is called by the background application running on the energy hub, it saves the IP address given by DHCP, this is used for further communication between the web server and the energy hub.

```
require_once("includes/db_connect.php");

if(isset($_GET['ip'])){
  $sql= "INSERT INTO `DEVICE` (".
      "`ID_DEVICE` ,".
        "`IP`)".
        "VALUES (NULL , '".$_GET['ip']."')";

  $con->query($sql); // Run the query in the MySQL server

  // No feedback needed since the energy hub will not expect an answer.

} else {
  echo 'IP not set';
}
```

The ip address is send by the GET method (saveip.php?ip=127.0.0.1), this is how the data in encoded into a URL, being collected in the variable $_GET['ip']. A $sql variable string is created containing the SQL code to be run at the MySQL server.

db_connect.php For the communication to the database a driver is used in PHP that provides an interface to the MySQL server. The PHP mysqli extension (MySQL improved) is used in this project, this is recommend for MySQL servers version 4.1.3 or later. This extension provides several benefits as a objective-oriented interface, support for multiple statements, embedded server support and more can be found in the MySQL documentation.

```
require_once("db_globals.php");

$con = new mysqli(DB_HOST,DB_USER,DB_PASS,DB_NAME); // Creates new mysql connection

if($con->connect_error){
  echo "Failed to connect to MySQL: (" . $con->connect_errno . " ) ". $con->connect_error;
}
else { echo "Connection established"; }
```

In this script an object is instantiated with a connection to the MySQL server.

$con = new mysqli<parameters >)

The parameters are included from the db_globals.php, setting the server host, user, password and database to be used.

db_globals.php Using a script to define the parameters for the MySQL connection, All the scripts that need to use the global parameters for the connection to the MySQL server, should include db_globals.php as shown in the db_connect.php above.

```
// MySQL condifuration
DEFINE ('DB_USER','root');
DEFINE ('DB_PASS','pass');
```

```
4   DEFINE ('DB_HOST','localhost');
5   DEFINE ('DB_NAME','iEnergy');
```

The global parameters the connection to the MySQL server are define in this script.

- DB_USER - Database user name with read, write and execute permissions.
- DB_PASS - Password for the user
- DB_HOST - Hostname for the MySQL server, if running at the same host as the PHP server, localhost or 127.0.0.1 should be used.
- DB_NAME - Database name to connect to.

Error Handling The implementation of an error handling will keep the web interface more reliable, improve user experience and alert the administrators to a error situation. A error handling PHP script is called in background, this script will test the communication between the web server and the energy hub, and the communication to the MySQL server. As described in the analysis in this report, the user interface is blocked giving the message warning when the communication with the MySQL server is lost and a warning is given to the user when no communication with energy hub is found.

errorhandler.php The error handler PHP script is included in each web page before any other script, this script is included when a page is loaded and is recursively called in background to test the connections. The script gives a XML response, when called as background so the client side JavaScript can handle the real time warnings to the user. This method can be called as active error handle, since it acts in real time, usually error handling is implemented when is need to do some action, for example retrieve data from database. With this system the user is immediately blocked from doing any action until the situation is solved.

Since this script is included before everything else, the HTML, JavaScript and CSS layout is included in the file. Bellow several code blocks were extracted for and easier explanation, the full code can be seen using the application SeeIt and in the appendix of this report.

At first and most important the database connection have to be tested, as such a connection to the database is attempted. If the connection is successful the database error session variable will be 0. In case of connection failure the error variable will be changed to one and kept until the connection is re established, changing the session error variable to 2. This will tell the client side script that the situation was solved and in the next test the variable is changed to 0 again. This can be seen in the code bellow.

```
1
2   if ($_SESSION['error_db']>1) {
3     $_SESSION['error_db']=0;
4   }
5
6   $_SESSION['error_device']=0;
7
8   // Test DB connection
9   require_once("includes/db_globals.php");
10
11  $con = mysql_connect(DB_HOST,DB_USER,DB_PASS,DB_NAME);
12
13  if(!$con){
14    $_SESSION['error_db']=1;
15    $error = "Failed to connect to database: ". mysql_error();
16  } else {
17    if ($_SESSION['error_db']==1){
18      $_SESSION['error_db']=2;
19    } else {
20      $_SESSION['error_db']=0;
21    }
22  }
```

If no error regarding the database connection is acknowledge, the ip address for the energy hub is retrieved and the connection with the energy hub can be established.

```
if($_SESSION['error_db']==0){

    $device_port = 5555;

    $con = new mysqli(DB_HOST,DB_USER,DB_PASS,DB_NAME);

    $sql= "SELECT IP ".
        "FROM `DEVICE` ".
        "ORDER BY ID_DEVICE DESC ".
        "LIMIT 1";

    $res = $con->query($sql);

    $row = $res->fetch_row();

    $device_ip = $row[0];
```

The SQL statement **SELECT IP FROM 'DEVICE' ORDER BY ID_DEVICE DESC LIMIT 1** retrieves the last ip address insert in the table. The keywords ORDER BY ID_DEVICE DESC will show the last data added and the keyword LIMIT 1, limits the number of values return to only one. The value is then fetched from the result and assigned to the variable device_ip for later use in the connection.

For the communication to the energy hub a socket have to be created and a connection established. In case of one of this steps don't work (Socket creation and Connection), a error session variable is set to 1, the client side application will catch this error and alert the user for such situation. The socket is closed after testing to leave the connection path open for new tests or commands to be send.

```
    if (!$socket=socket_create(AF_INET, SOCK_STREAM, SOL_TCP)){
      $_SESSION['error_device']=1;
    } else {
      $_SESSION['error_device']=0;
    }

    if (!socket_connect($socket,$device_ip,$device_port)){
      $_SESSION['error_device']=1;
    } else {
      $_SESSION['error_device']=0;
    }

    socket_write($socket,"a");

    socket_close($socket);

  }
```

The error handle client side (JavaScript) and server side (PHP) are developed at the same time, since the cooperation between both is essential for the proper operation of the error handling system.

This block of code is used only by the client side script, it make a request to the errorhandle.php with the encoded URL variable op=d, this block will return an XML response. The client side script then interpret the answer and reacts according to the errors.

```
  if(isset($_GET['op']) && $_GET['op']=='d'){

    echo '<ERROR>'.
          '<DB>'.$_SESSION['error_db'].'</DB>'.
        '<DEVICE>'.$_SESSION['error_device'].'</DEVICE>'.
        '</ERROR>';

```

```
8   } else {
```

The else statement in this code indicates that, if the script was not called by the error.js, then it will include the necessary HTML, CSS and JavaScript to the file where it was included.

An example of this situation is found at the index.php, the first lines of this script starts the session and includes errorhandle.php.

```
1 <?php
2   session_start();
3
4   require_once("errorhandler.php");
5
6   require_once("includes/db_connect.php");
7 ?>
```

error.js This script catches and handle errors in the client side, it uses background requests to the PHP script errorhadle.php. By the XML response it acts according the situation. JavaScript is a dynamic language with a lot of potential, and this is used in detail in this script.Meaningful blocks will be extracted from the code and explained here.

Three functions are part of this script, being two of them for animation to improve the user experience, they are not documented in this section. The remaining function is the mains focus of the error handling system, is called makeTests().

This functions initialize the variables req for the URL request, db_error and dev_error that will be assigned with the values returned by the request. At first the browser is tested to define which HTTP request object to use, the main difference is between IE (version 5 and 6) form Microsoft and the rest ot the browsers, since IE (version 5 and 6) uses an ActiveX object to deal with background requests.

```
1   var req;
2   var db_error;
3   var dev_error;
4
5   if(window.XMLHttpRequest){
6     req = new XMLHttpRequest();
7   } else {
8     req = ActiveXObject("Microsoft.XMLHTTP");
9   }
```

The open and send methods from the XMLHttpRequest object, are used to send a request to the server. The 'open' method specifies the type of encoding used in the URL (GET or POST) and if the request is synchronous or not. The 'send' method sends the request to the server, parameters are needed in case of POST encoding method.

```
1   req.open("GET","/errorhandler.php?op=d",false);
2   req.overrideMimeType('text/xml');
3   req.send();
4
5   var result = req.responseXML;
```

The result variable is initialized and the returned XML data is assigned to it, this data is then extracted and used according to the situation. An example of retrieved XML data can be seen bellow:

```
1     <ERROR>
2       <DB>0</DB>
```

```
3       <DEVICE>0</DEVICE>
4     </ERROR>
```

Graphical representation of an error in the connection to the database and communication to the energy hub device.

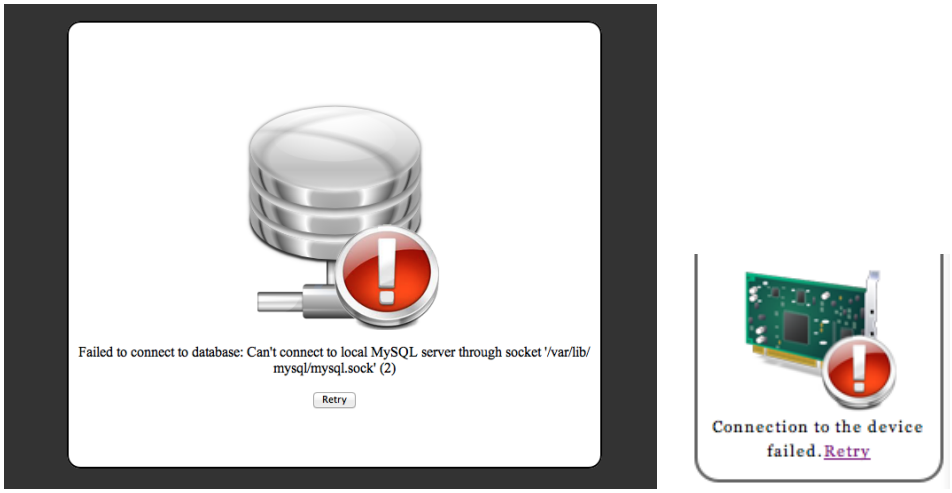


Figure 7.18: Database Error and Device communication Error

7.5.3 Conclusion

In the analysis phase of this project, it was necessary to understand how can the database an the web interface be verified. As such a scaled down web interface with only two modules and the energy hub is implemented.

With this more realistic approach to the system, it was possible to verify problems regarding the common requirements for the communication between the modules and the energy hub. The database is able to handle several sensors connect to one module, but the energy hub have no knowledge of the sensors connected to each module. For a completely working system, deeper communication and protocol analysis have to be done.

A working high-fidelity prototype is created, ids are assigned to the modules being id:0 to the energy hub, id:1 to the battery and id:2 for the photovoltaic panels. Each module have just one sensor that measures the current (ampere), the id of the sensor is the same as the module id for a working prototype.

The end system can handle all functionalities expected for this project, the energy hub is able to send commands to the energy hub and data can be added or retrieved from the database. The error handling and the dynamic update of contents on the client side, improved the user experience for the control and navigation in the system.

The web interface is up and running at the address http://10.1.18.223 inside the university network.

7.6 GPIO Device driver - Dennis

In order to fulfill the requirements shown in table 7.3[28], an driver is needed in order to control the GPIO pins on the micro controller.

[28] All requirements can be found in the Appendix EEPRO3

ID	Requirement	Description	Comments
B-2.1	Over production	Overproduced energy is wasted in a dummy load connected to the hub.	Energy routing is controlled by GPIO's
B-2.2	Over production	If two or more producers are connected and one can be without, it is stopped.	Energy routing is controlled by GPIO's
B-2.3	Under production	If there is no overproduction (dummy load is turned off), the grid is con- nected to the power line in case the producers cannot produce enough energy.	Energy routing is controlled by GPIO's

Table 7.3: Requirements that concerns GPIO

7.6.1 Analysis

The kernel is running in an embedded system environment with only known parameters connected to it. Therefore a specific GPIO driver is written to decrease development time compared to if a more generic driver was made.

In order to verify a part

GPIO nr.	Pin nr.	Name	Direction
GPIO0	0.12	PLC_nReset	Output
GPIO1	0.13	PLC_wake	Output
GPIO2	0.18	PLC_nSleep	Output
GPIO3	2.10	FPGA_int	Input Interrupt
GPIO4	1.6	PS_main	Output
GPIO5	1.23	PS1_in	Output
GPIO6	1.25	PS2_in	Output
GPIO7	1.19	PS1_out	Output
GPIO8	1.21	PS2_out	Output

Table 7.4: GPIO connection table

Table 7.4 shows that there will be one device driver with nine valid minor number calls in it. Each of the GPIO devices identifies a GPIO port on the micro controller. The name is an indication of what the GPIO pin is connected to.

PLC_nReset, *PLC_wake* and *PLC_nSleep* controls the Power Line Module, by resetting it, wakening it or put it to sleep when there is no need for it.

FPGA_int indicates if the FPGA has new data for the micro controller to read, this is indicated by an interrupt.

PS_main Turns on or off the supply from the grid.

PS1_in and *PS2_in* sets each of their power switches as an input port whereas *PS1_out* and *PS2_out* sets each of their power switch as an output device.

7.6.2 Design

When inserting the module the pins needed are configured and given default values. The driver is made so that the pins can be accessed singly and thereby be given a logical value '0' or '1', or a logical value can be read from a single pin.

In order to verify if the GPIO device driver works as excepted, the *echo* command and the *cat* commands have been used.
Example of reading GPIO0: *cat /dev/gpio0*

Example of setting GPIO0 to 1: *echo 1 /dev/gpio0*

In order to control the power switches an user space application has been written to easy the control of these. The program reads the two first arguments of argv. argv[1] defines the power switch to control, where 0 controls GPIO4 which is the power switch controlling the main. 1 and 2 each controls a power switch connected to a module (wind turbine, battery or similar). argv[2] defines the action of the power switch, where valid arguments are: *in*, *out* and *off*.

- in, opens for energy flow from a module to the power line.
- out, opens for energy flow from the power line to a module.
- off, turns off energy flow in both directions.

```
/*
 ============================================================================
 Name        : port.c
 Author      : E10-Team3 dENNES
 Description : EA-LPC2478 - Control of the power switches connected
 ============================================================================
 */
#define GPIO "/dev/gpio"

/*
        PORT <num> <status>

        Status: in, out, off
        num: 0, 1, 2

        in      out
----------------------------
        4       x               PORT0
        5       7               PORT1
        6       8               PORT2
----------------------------
*/


int main(int argc, char *argv[]) {
          int fd, ret, num;
          char ibuff[10], obuff[10];
          char val;
  char *cmd;
  char dev[3][2]

  num = *argv[1]-'0';
  cmd = argv[2];

          if(num >= 0 && num <= 2){   // Check if a valid port is called
            if(strncmp("in", cmd, 2) == 0){ // Set module as input
                    printf("\nINPUT port%d",num);
                          dev[num][0] = '1';
                          dev[num][1] = '0';
                  }
                  else if(strncmp("out", cmd, 3) == 0){ // Set module as output
                        printf("\nOUTPUT");
                          dev[num][0] = '0';
                          dev[num][1] = '1';
                  }
                  else if(strncmp("off", cmd, 3) == 0){ // Turn off module in both directions
                          dev[num][0] = '0';
                          dev[num][1] = '0';
                  }
                  else{             // Invalid command sent
                          printf("\nSomething wrong");
                          exit(-1);
                  }

                  sprintf(ibuff,"%s%d",GPIO,num+4); // Compress write string
                  sprintf(obuff,"%s%d",GPIO,num+6); // ex. /dev/gpio4 if PORT0

                  if((fd = open(ibuff,O_WRONLY)) < 0){  // Open GPIO device driver
                          printf("Cannot open file in.\n");
```

```
60                              exit(-1);
61                      }
62                      ret = write(fd, &dev[num][0], 1);
63                      if(ret < 0){
64                              printf("Write failed in: %d\n", ret);
65                              exit(-1);
66                      }
67                      close(fd);
68
69                      if(num != 0){           // If port 1 or 2, set second pin on module
70                              if((fd = open(obuff,O_WRONLY)) < 0){
71                                      printf("Cannot open file out.\n");
72                                      exit(-1);
73                              }
74                              ret = write(fd, &dev[num][1], 1);
75                              if(ret < 0){
76                                      printf("Write failed out: %d\n", ret);
77                                      exit(-1);
78                              }
79                      close(fd);
80                      }
81              }
82              return 0;
83      }
```

Listing 1.3: User space application to control the Power switches

7.6.3 Implementation

The gpio header file contains prototypes of the functions used, defines and arrays holding values for the different registers to call in order to avoid *if/else* or *switch/case* in the different functions.

```
1  #define GPIO_MAJOR 245
2  #define NUM_GPIO_DEVICES 9
3
4  #define GPIO_IRQ 14
5  ---
6  static u32 fdir[] ={
7    FIO0DIR,
8    FIO0DIR,
9    FIO0DIR,
10   FIO2DIR,
11   FIO1DIR,
12   FIO1DIR,
13   FIO1DIR,
14   FIO1DIR,
15   FIO1DIR,
16 };
17 ---
```

Listing 1.4: GPIO header file

The fops (file operations structure) structure with the different device driver call possibilities. This device driver can be opened, closed, read and write.

```
1  static struct file_operations gpio_fops = {
2    .owner    = THIS_MODULE,
3    .read     = gpio_read,
4    .write    = gpio_write,
5    .open     = gpio_open,
6    .release  = gpio_close,
7  };
```

Listing 1.5: GPIO fops structure

Initialization of the GPIO pins is called from the init function. The *init* and *exit* functions are similar to the once implemented in the ADC and the UART device drivers except for the *gpioInit* function call.

```
1  static void gpioInit(void){
2    int i=0;
3    m_reg_bfs(SCS, 0x1);  // Enable fast I/O on port 0 and 1
4    for(i=0 ; i<NUM_GPIO_DEVICES ; i++){
```

```
    m_reg_bfc(psel[i], enable_pinsel[i]);
    if(i != 3){
      m_reg_bfs(fdir[i], gpio_pins[i]);
      m_reg_bfs(pin_default_reg[i], gpio_pins[i]);
    }
  }
}
```

Listing 1.6: GPIO init function

When the file is opened, the interrupt is configured if it is GPIO3 that is called.

```
static int gpio_open(struct inode* inode, struct file* file){
---
  if(num == 3){
    // setup interrupt for pin input
    DPRINT("\nEnable int for input");
    flag = 0;
    m_reg_bfs(EXTMODE, 1);    // INT0 Edge sensitive
    m_reg_bfc(EXTPOLAR, 1);   // falling edge
    m_reg_bfs(EXTINT, 1);     // clear pending INT0 interrupts

    ret = request_irq(GPIO_IRQ, interrupt_gpio, SA_INTERRUPT,"GPIO P2.10 interrupt", NULL);
          //SA_SHIRQ  = Shared interrupt
          //SA_INTERRUPT  = Fast interrupt

    if(ret){
      printk("IRQ%d is not free. RET: %d\n", GPIO_IRQ, ret);
      return ret;
    }
  }
  return 0;
}
```

Listing 1.7: GPIO open function

The interrupt is freed in the close function if operating in the GPIO3 file.

```
static int gpio_close(struct inode* inode, struct file* file){
---
  if(num == 3){
    free_irq(GPIO_IRQ, NULL);
  }

  return 0;
}
```

Listing 1.8: GPIO close function

When read the file returns the character '1' or '0' according to which value the output has. If GPIO3 it read '0' is returned if the pin is low, otherwise the driver is sent to sleep until the pin is '0'. GPIO3 is connected to a pin on the FPGA which is used to symbolize that data is ready to be read from it. The pin goes high again after the external memory address *0x82000000* has been read.

```
static ssize_t gpio_read(struct file *p_file, char *p_buf, size_t count, loff_t *p_pos){
---
  if(num == 3){
    if((m_reg_read(pin_read[num]) & gpio_pins[num]) > 0){
      DPRINT("\nNo interrupt from FPGA, SLEEP!\n");
      m_reg_bfs(PINSEL4, (1<<20));        // Set p2.10 to EINT0
      wait_event_interruptible(my_queue, (flag != 0));  // Put the function to sleep
      flag = 0;
    }
    value[0] = '0';
  }
  else{
    DPRINT("\nREAD ELSE");
    if((m_reg_read(pin_read[num]) & gpio_pins[num]) > 0){
      value[0] = '1';
    }
    else{
      value[0] = '0';
    }
  }
```

```
21
22   if(copy_to_user(p_buf, value, count)){  // Copy read value to user space
23     DPRINT("\nFailed: copy_to_user");
24     return -EFAULT;
25   }
26   return 1;
27 }
```

Listing 1.9: GPIO read function

The value written to the device driver shall be a logical '0' or '1' which is then set on the appropriate GPIO pin.

```
1  static ssize_t gpio_write(struct file *filp, const char *bufp, size_t count, loff_t *p_pos){
2  ---
3    if(copy_from_user(value, bufp, count)){ // Copy value from user space to kernel space
4      return -EFAULT;
5    }
6    if(value[0] == '0'){
7      //FIO_CLEAR
8      DPRINT("\nCLEAR PIN");
9      m_reg_bfs(pin_clear[num], gpio_pins[num]);
10   }
11   else if(value[0] == '1'){
12     //FIO_SET
13     DPRINT("\nSET PIN");
14     m_reg_bfs(pin_set[num], gpio_pins[num]);
15   }
16   else{
17     DPRINT("\nNot a valid character");
18     return -EFAULT;
19   }
20
21   return count;
22 }
```

Listing 1.10: GPIO write function

If the driver is sent to sleep (in the read function) it jumps to the *interrupt_gpio* function when awakened as result of the pin having a falling edge.

```
1  static irqreturn_t interrupt_gpio(int irq, void *dev_id){
2
3  //  cread = m_reg_read(urbr[0]);    // Read the buffer
4
5    DPRINT("\nREAD int\n");
6
7    m_reg_bfs(EXTINT, 1);   // clear pending INT0 interrupts
8    flag = 1;
9    wake_up_interruptible(&my_queue);
10   return IRQ_HANDLED;
11 }
```

Listing 1.11: GPIO interrupt function for GPIO3

7.6.4 Verification

In order to test requirement B-2.1, B-2.2 and B-2.3 the whole system needs to be assembled. Table 7.5 shows the tests described in the Launch phase.

ID	Requirement	Test Description	Grade/Comment
B-2.1	Energy Control - Over-Production	A dummy load and a producing module is connected to the system. As no consumers is connected, the system is over- producing. The current flow to the dummy load is measured. If the current flowing to the dummy load is the same as the one coming from the producer (maximum -10%), the test is valid.	
B-2.2	Energy Control - Under-Production	A dummy load and a producing module is connected to the system. As no consumers is connected, the system is over- producing. A variable consuming module is now connected. The resistance in the consuming module is decreased (increas- ing load). The current flow to the dummy load shall now fall. When no current is flowing to the dummy load, it shall be ob- served that the grid is connected. This is done by increasing the load of the consuming module and measuring the current flow from the grid.	

Table 7.5: Verification of energy routing requirements.

As these tests cannot be performed, due to only a part of the system is finish, a minor test has been made in order to verify the functionality of the written device driver.

With a multimeter the different pins used as outputs are measured one by one and the output set by sending a value to the file by the *echo* command. Also the pins are read with the *cat* command after setting the pin in order to verify the read function.

The interrupt input pin is tested by reading the pin when the FPGA is not yet interrupting to verify if it goes to sleep. If the interrupt pin are read after an interrupt has been given from the FPGA, '0' is returned.

7.6.5 Conclusion

The small GPIO device driver test has been made with success. All 8 output pins can be set and cleared through the driver. Also the status of the 8 output pins and the one input pin can be read through the driver. If the FPGA has set the interrupt pin to '0', the driver returns '0' immediately without waiting for an falling edge.

7.7 Deployment

Web interface The final web interface is up and running at the address http://10.1.18.223 inside the university network.

GPIO Device driver The GPIO driver is working with implemented read and write to single GPIO pins and interrupt on GPIO3. Morten Opprud has been by and verified the functionality.

Chapter 2

Post Project

This is the final state of the working process. This part is about evaluating the final product and the whole project. The product acceptances is left out, because the product is not completely finish.

1 Product evaluation

Due to a limited development time the product was not completed, according to the requirements. The different requirements are listed below together with an status and a short description.
Status options:
X : requirement implemented and tested.
/ : Part of the requirement has been implemented and tested.
None, requirement not implemented.

ID	**Requirement**	**Description**	Status
F-1	Communication		
F-1.1	Web Server	The hub shall be able to put data on a web-server.	X
F-1.2	System	The hub shall be able to communicate with a connected module through power line communication.	X
F-1.2.1	Protocol	The Power Line Communication shall be done according to the common protocol (see appendix: Protocol)	
F-2	Routing		
F-2.1	Direction	When a module is connected the system shall automatically find out if it is an producer-, storage- or consumer module.	
F-2.2	Consumer	Only allowed to take energy from the system.	
F-2.3	Producers	Only allowed to give energy to the system.	
F-2.4	Storage	Must acts either as a consumer or a producer (can be changed dynamically).	X

Table 1.1: Functional: System shall do...

Through the Power Line module developed, the system is able to send data from one module to another.
With the user space application written (PORT), the system is able to control the Power Switches in a way that they can only be set as a consumer or producer.

ID	**Requirement**	**Description**	Status
NF-1	User Interface		
NF-1.1	Web Interface	Maximum 2 click to go where you want on the website.	X
NF-1.2	HW Interface	Able to connect 10 modules to the hub.	/
NF-1.3	HW Interface	A locker shall be unlocked to access the physical hw.	
NF-1.4	HW interface	An Emergency stop shall be visible on the hub and shall shut down the system if pushed.	
NF-1.5	HW Interface	1 start button for the hub. 10 buttons to each start a module.	/
NF-2	Electrical		
NF-2.1	Voltage	The input and output ports must work in the range 30V +/- 10%.	X
NF-2.2	Current	The maximum current on each port is defined to maximum 30 Amperes.	

Table 1.2: Non-Function: System shall be...

At the systems current state, it is only able to connect two modules. The switches on the FPGA board has been used as command buttons to control the modules where only 8 switches are available. The Power switch device has a build in level detection of 30V +/- 10 %, where is shuts down the energy routing if the voltage level is not accepted.

ID	Requirement	Description	Status
P-1	Hardware		
P-1.1	Housing	The housing have to be water-proof, to not harm the system.	

Table 1.3: Performance: How well does it have to be done...

The interface buttons and LEDs is used on the Spartan 6 board, but the housing and cabinet for the system has not been made. This mean that the requirement for water resistant has not been fulfilled.

ID	Requirement	Description	Status
B-1	Status		
B-1.1	Web Interface	Warnings are posted in the web interface.	X
B-1.2	Physical	1 diode showing when a module is off and one showing when it is on.	/
B-1.3	Report	An e-mail is sent to the defined user if an error occurs.	
B-2	Energy Control		
B-2.1	Over-production	Overproduced energy is wasted in a dummy load connected to the hub.	
B-2.2	Over-production	If two or more producers are connected and one can be without, it is stopped.	/
B-2.2	Under-production	If there is no overproduction (dummy load is turned off), the grid is connected to the power line in case the producers cannot produce enough energy.	/
B-3	Errors	Humidity and Temperature sensor will be placed inside the system housing.	
B-3.1	Humidity	If the humidity is above the maximum level 70%, the system shuts down.	
B-3.2	Temperature High	If the temperature is higher than 55 degrees, the system shuts down.	
B-3.2	Temperature Low	If the temperature is below 0 degrees, the system shuts down.	

Table 1.4: Behavioural: How the system shall react...

The requirements regarding the user interface application were fulfilled except warnings are not send to the administrator in case of error occurrence. This requirement was not fulfilled since a SMTP server is needed to accomplish the email sending functionality in PHP. In a working product a server could be implemented or even an existent server can be used to manage the send of warning to the administrator.

The requirements regarding stopping and starting modules are partly connected as the processor can control the different power switch modules connected.

1.1 Conclusion

The limited development time has affected the final product, the necessary requirements were not fulfilled in order for the product to be delivered to the costumer.

2 Project evaluation

The questions below were prepared as a checklist and answered together within the team.

1. *Did you choose the most suitable use of EUDP tools for the project? Which tools were used and which were satisfactory?*
   As part of a learning curve, all tools provided by the EUDP method were used, in order to gain broad knowledge on how to use the EUDP as a working process.

2. *Was the project finished on time?*
   According to limited development time beside the study and course time, the project was not finished on time. The missing knowledge about the EDUP phases has also affected the work done in the project.

3. *Did you use new (to you) methods?*
   All the four phases of the EUDP method was used for the first time. The realisation phase was handle the best. But the understanding of the different phases was not that good when they where used, but a grater understanding of the different phases and what to do in then has been gathered through this project. This method will be used with our new knowledge in the next project.

4. *What do you think needs to be changed in your next project compared to this project?*
   The launch phase has to be written more complete according to the EUDP philosophy.

5. *Are you satisfied with the project as a whole?*
   The project was satisfactory due to the knowledge acquired by the EUDP working process methods.

Chapter 3

Appendix

Team 3:
Interaction Design Report + videos
Web report
Pro3 report (Pre-project and launch phase)
Code
WEB
VHDL
Device driver
User space
All teams:
Protocol document - Communication protocol between the modules
PLC HW document - Hardware circuitry for the PLC modules
Common requirements - Common system requirements for all teams