Swakeup

PROJECT REPORT

within the lecture of
Programming Embedded Systems

at Uppsala University
in the Departement of Information Technology

Elmar van Rijnswou (Elmar.Vanrijnswou.9818@student.uu.se)
and Maximilian Stiefel (Maximilian.Stiefel.8233@student.uu.se)

Deadline: 2017-05-31 24:00

Processing Period:	January 2017 - May 2017
Supervisor:	Philipp Rümmer (philipp.ruemmer@it.uu.se)

Contents

List of Figures

List of Tables

1 Introduction

It is a well-known fact, that Sweden is a country with big variations of the daily hours of sun throughout the year. Far up north the sun does not set anymore in January. Even in Stockholm in January, the most dark month of the year, the sun rises at 8:47 am and sets at 2:55 pm (cf. [1]). According to *Sveriges Radio* "many Swedes suffer from the winter blues or seasonal affective disorder" (cf. [2]). In a strong winter every source of light is a source of happiness. This is why a wakeup light, which is based on a strong light source (at least 10 W RGB LED), is able to give one the optimal start into a dark winter day with an artificial sunrise as bright as a real sun shining through the window.

This report describes the Swakeup (from engl. "Swedish Wakeup Light"), a device communicating to the user not only through light. It does not simply wake one up, but also gives one information about social media, latest mails, calendar and weather. The user interface consits besides of a high-power LED of an OLED screen. Swakeup is also part of the IoT as it has the ability to communicate via IEEE 802.11. This of course enables a lot of possibilities e.g. connecting your phone to the wakeup light. A lot of effort has been put into the designing maxim, that everything should be as small as possible. The whole electronics fit on an base area of 5 cm x 4 cm. So the Swakeup fits smoothly on the bedside table. And honestly: What is the last thing people are doing before they go to sleep? Right! They look on your phone. That is why Swakeup comes with a USB charger for your e.g. phone as well. Another design maxim of this product is cheapness. Everybody should be able to buy one. As all engineering work is available online, it gives people (with the corresponding knowledge) the opportunity to build a wakeup light themselves, look up what this device is doing with their personal data or even contribute to the product.

2 Background and Analysis

Wake-up lights have been around for a while already, Phillips http://www.philips.se/ is one of the market leaders when it comes to wake-up lights. Their wake-up lights usually consist out of a one color lamp with a simple 7-segment display to display the time. Other higher priced variants of their assortment contain smart-phone connectivity. Most other smaller companies follow a same recipe. One of the exceptions are the so called smart light bulbs. Such as the"MagicLight Pro" https://www.magiclightbulbs.com/ and http://theuplight.com/ which, other than a normal RGB lightbulb, can also be used for waking you up. The problem with Phillips wake-up lights is that they are in the higher price range, where the cheaper lights don't contain a screen. Furthermore, none of the options have a screen that is capable of displaying more than just the time, which could be a great selling point with the use of social media nowadays. One could see onces activity right after waking up.

3 Design

3.1 System Architecture and Required Functionality

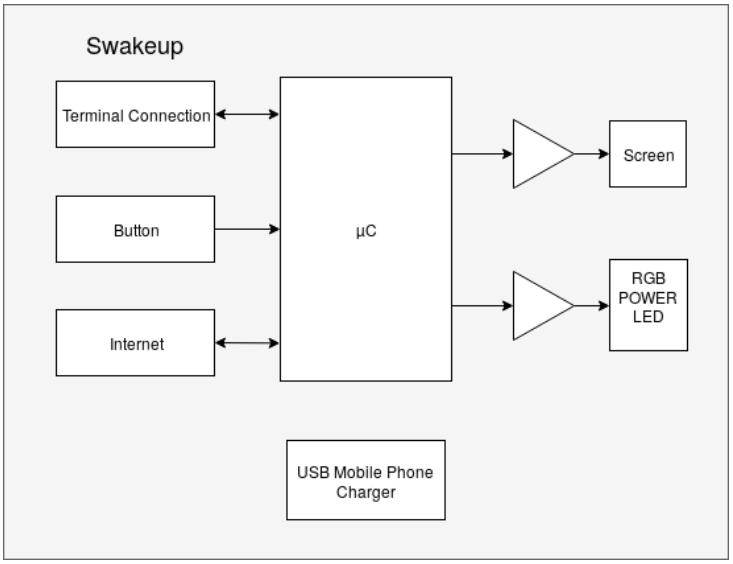


Figure 3.1: Simplified blockdiagram of Swakeup.

In fig. 3.1 a simplified system architecture from an abstract (e.g. customer) point of view is displayed. The heart of the system is a microcontrolller. This microcontroller has to procide the necessary peripherals and enough RAM and flash for the necessary funtionality, which has to be implemented in software.

A terminal connection is crucial to transport data to a computer. One needs this to do debugging and development. The idea is, that a USB cbale can simply be plugged in and then the developer can start working without additional complications.

A button is not only usefull for the programmer to interact with the code to set the system to a certain state, but it is also absolutely needed for deployment of the systems in the real world. Obviously, it should be extra convenient and simple for a sleepy user to snooze (e.g. short button pressing) or to turn off the alarm (e.g. long button pressing).

The idea of a IoT device is of course, that it is connected to the internet, that is why a component has to be included into the design to, which enables the system to receive a IP address. As the final product will be standing most likely on a bedside table of the customer a wired internet connection (e.g. Ethernet) does not make any sense. Instead a wireless connection is more convenient (e.g. IEEE 802.11).

USB charging circuitry i.e. a USB dedicated charging port has to be implemented to charge phones with a current of at leas 1.5 A.

As screen provides information to the user, which goes beyond the current date and time (e.g. weather, social media and email). It is reasonable, that this screen needs to be powered somehow. Also the software developer or the user should have controll over the screen brightness. This means, that some

time has to be invested in the contruction of a powerful software-steerable screen driver.

To light up a multi-color LED with a sufficient brightness to emulate a sunrise circuitry has to be implemented to drive a high current. This current has to be steerable in magnitude by the microcontroller to mix colors together according to the RGB color model.

3.2 Design Choices

Block	**Component(s)**	**Costs**	**Justification of the Choice**
Terminal Connection	SILICON LABS CP2102	24 SEK	Simple chip. Drivers for different operating systems are usually out-of-the-box available (plug and play). Available in QFN package.
Internet	ESP8266	25 SEK	ESP-E12 comes as ready to solder SMD module. No antenna design required. Easy connection to microcontroller via UART. Sming framework makes it easy to program chip in C++.
USB Mobile Phone Charger	TS30012 and TPS2514	40 SEK	TS30012 is a relatively cheap chip to break down 20V to 5V and to provide up to 2A output current. USB designated charging ports need signaling for different proprietary/open standards. TPS2514 is a quite nice chip to provide this signaling for different brands.
RGB LED and Driver	Custom Step-Down Converter	20 SEK per Stage	Fast, powerful LED driver built up with discrete components can be tailored to the requirements of the device and is therefore cheaper and more performant. Necessary feedback circuitry is also hard to find as COTS.
Screen and Driver	SEPS525F and Custom Opamp Amplifier	150 SEK and 30 SEK	The OLED displays 18-bit colors with a resolution of 160x128 pixels. Moreover the pciture is clearly visible from almost every angle. One channel of the LM324 (four channel operational amplifier) was unused a simple circuit made it a cheap and easy-to-use DAC screen driver.
Microcontroller	ATxmega128A4U	60 SEK	High-end 8-bit microcontroller with a lot of usefull perpherie such as a powerfull 12-bit, 2mbps ADC. Available in QFN package.

Table 3.1: Design choices: Why has which component been chosen?

4 Implementation

4.1 Hardware

4.2 Software Interfaces

As can be seen from the hardware, there are two systems that have to communicate with each other. Not only that, but its also needed to talk with a server back-end. The choice has been made to use Protobuf in order to save time. Protobuf, short for Protocol buffers, are a language-neutral, platform-neutral extensible mechanism for serializing structured data developed and maintained by google https://developers.google.com/protocol-buffers/. This translates message structures in language dependent types. In C the messages are translated into structs, where the messages are translated into Objects in Java.

4.3 Software Microcontroller

In order to facilitate all the functionality, code had to be written. Because the system is quite complex with many different functions, a good and clear structure has to be realized. Much of the development time has been invested in the writing and testing of underlying code which doesn't have a direct result for the finalized product. In the following sub-chapters this system will be described, and explained why certain choices and compromises have been made.
In fig. 4.3 below one approach of abstracting the code can be seen. Four layers are displayed with each dependencies on a lower layer. This has multiple advantages. Every block can now be swapped with different components while the system keeps running, if implemented well without breaches of layering.

For example, it's now possible to change microcontroller platform by only having to port the platform layer. Or if a different screen driver is used, only the software driver will have to be ported. Or of the application is meant to be changed, the application layer has to be adjusted without any changes on the tightly integrated platform code. Other advantages are the reduction of having to test the whole system time after time, as it is possible to rely on the underlying layer if that is tested accordingly.

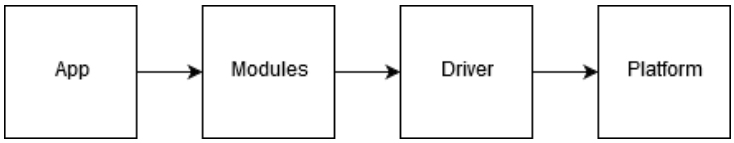


Figure 4.1: The layered approach.

To make this layered code structure a reality, a small operating system has been developed which allows event driven communication, and a modular approach for driver and sub-systems. This modular approach also makes power saving easier as a module will be uninitialized when it's not used anymore. Removing any possibility for the developer of the system to forget to disable the peripheral/module.

4.3.1 Operating System

To make the previously discussed layering possible with communication, a small operating system has been developed with a minimum subset of functions. This subset is divided in two parts, Events and Modules.

Module

The modular system is realized by a struct. Every module contains a name, a usage counter, initialize and deinitialize functions, and dependencies. A module can be initialized by the "module_init" function. This function will check for all the dependencies, whether they are already initialized. If this is not the case, it will be initialized including all the dependencies of the dependency. The "module_init" function will also call the initialize function, which can be used for a module to set up registers for example. Once a module is not needed anymore, it can be deinitialized with the "module_deinit" function. This function will decrease the usage counter for every dependency, and once a dependency is not used anymore it will also be deinitialized. This function also calls the deinitialize function of the given module.

```
#define MODULE_DEFINE(VAR, DESC, INIT, DEINIT, ...)     \
Module VAR = {                                          \
.init = INIT,                                   \
.deinit = DEINIT,                               \
.cnt = 0,                                       \
.name = DESC,                                   \
.deps = { __VA_ARGS__ }                         \
}
MODULE_DEFINE(CORE, "Central core", init, deinit, &TIME, &COMMAND, &ESP8266);
```

In the code above a simplified usage example can be seen of this modular system

Event

For communication between modules an event based system has been realized. The developed system does not have any kind of priority for events, and events can be ignored if the event buffer is full because of a too heavy event. Every event contains a counter for debugging purposes, a pointer to data that can be given along with the event, and a description. An event can be created by use of the following code:

```
#define EVENT_REGISTER(eventName, desc)\
Event eventName = \
{.eventId = __COUNTER__, .data = 0, .description = desc, .descLen = sizeof(desc) }
EVENT_REGISTER(EVENT_UART_DELIMITER, "Got UART delimiter");
```

Events alone do not serve any purpose without listeners. Thus it's possible to register listeners in the event system. With the "event_addListener" function a module can listen to a certain event and provide a function to be called upon the reception of such an event. This can be seen in the code below:

```
event_addListener(&EVENT_UART_DELIMITER, callback);
```

The listeners can be removed with the "event_removeListener" function. The adding listeners is usually done in the initialization function, as the modules require these events during the period that they are active. In order to fire an event to all the listeners the following example can be used:

```
event_fire(&EVENT_UART_DELIMITER, SYSTEM_ADDRESS_CAST (&delimiters[USART_ID][i]));
```

This line of code will fire a EVENT_UART_DELIMITER event, and adds some information to go with it.

In order for these events to be processed the "event_process" has to be called. This should be the only function called in the infinite while loop of the system. A way to save energy is to pass functions to the event system if the system is capable of a sleeping functionality. Now the system will sleep whenever there are no more events to process, or wake up when an interrupt occurs.

4.3.2 Deriving functionality

In fig.?? the functions of the system can be observed, these functions can be translated into modules and a hierarchy can be created. Along with drivers for the hardware, some software drivers have been written as well. These software drivers provide support for internal development such as logging text in a proper fashion, and creating a way of executing commands on the device. All the platform drivers are interfaces to the system's peripheral. On top of this layer are the actual drivers. This houses the initialization code for the hardware drivers, and gives a way to access the functionality of the hardware without needing to know the registers/commands. The layer above this provides a more generic function set, where no knowledge of the hardware is needed anymore. This can then be used by the last layer, the Application layer. The application layer houses different applications, and a general state of the device can be found inside this layer.

Figure 4.2: The system structure and hierarchy

above in fig. 4.3.2 this translation from system design to a more refined code structure is shown. A brief description of the required functionality for every block follows.

4.3.3 Platform layer

Every used peripheral has a platform driver. The UART driver has an input and output buffer, and is being used with interrupts. Two UART channels are being used, one for the ESP8266 and one for the USB-Serial converter. The SEP525F has a SPI interface without a need or possibility to read, the SPI driver contains only blocking write. This should be changed to DMA and interrupt transfers. The timer sets up the real time counter for the second pulses, and has functions to set the PWM of every channel for the power board. And lastly, the ADC peripheral. This scales the voltage read from the input pins and provides it to whoever needs the ADC values.

UART

The UART module delivers all the communication to the USB-Serial converter and ESP8266. The code has been kept as generic as possible so that adjustments to the external hardware has no influence to the driver code. It would even be possible to move to a different xmega model with more USART ports since a macro is used for generating the interrupt code. Every USART port has a status, an array with delimiters, an in and output buffer, a sending status and an id. The status contains variables for the buffers, and allows for a ring-buffer usage. A ringbuffer is chosen as it has little overhead compared to lists or queues, the order of the data matters, and we don't expect to go outside of our given buffer capacity. The delimiters are used for the USART to listen to certain characters on the receiving data. Once there is a match an event will be fired. The example below shows the usage of this delimiter.

```
1  uint8_t uart_add_delimiter(char delimiter, USART_t * port);
2  static void callback(Event * event, uint8_t * data) {
3  if(event == EVENT_UART_DELIMITER){
4  struct UartDelimiter * delimiter = (struct UartDelimiter*)data;
5  if (delimiter->port == &ESP_UART_PORT) {
6  //Read buffers etc
7  }
8  }
9  }
10 static uint8_t init(void) {
11 uart_add_delimiter('\n', &ESP_UART_PORT);
12 event_addListener(&EVENT_UART_DELIMITER, callback);
13 return 1;
14 }
```

In this example, a certain module will tell the UART module to listen for a new line character, and the module will subscribe to the event. The UART module passes the delimiter information with it, so that there is knowledge about how much data can be read since the last delimiter event.
The interrupts are generated with the macros that can be found at appendix C. The following code can be used to generate the interrupt code for every USART channel:

```
USARTRXCISR(USARTE0, DEBUG_UART,     USARTE_ID, received);
USARTDREISR(USARTE0, DEBUG_UART,     USARTE_ID);
USARTRXCISR(USARTD1, ESP_UART_PORT,  USARTD_ID, );
USARTDREISR(USARTD1, ESP_UART_PORT,  USARTD_ID);
```

Since the system could be writing to the buffer that is being handled in the interrupt, a lock has been implemented to prevent unexpected outcome. One lock waits while the lock signal is freed, while the other lock function returns a 0 upon failure to acquire the lock.

```
#define lock(id) while (outBufferLock[id]); outBufferLock[id] = 1
#define unlock(id) outBufferLock[id] = 0

static inline uint8_t softlock(uint8_t id) {
if (outBufferLock[id]) return 0;
outBufferLock[id] = 1;
return 1;
}
```

The USART on both channels have the same settings: 8 bit words, medium level interrupts and a baudrate of 115200. This baudrate can go higher once the whole system is tested with good results.

SPI

The SPI driver is incomplete as is, a lot of performance optimizations can and should be done. One of the biggest disadvantages at the moment is that it is blocked writing. When a lot of data is being sent consecutively by the SPI driver, the event buffer fills up and might even get full. Interrupt based design should be looked at and investigated, as this would still allow the events to be handled. The caveat with this, however, is that a large buffer has to be allocated for the SPI. And memory is costly. DMA is another technique to solve the problem, this would eliminate the need for a CPU at all and give the system all the time to process the other tasks. One needs to be cautious, as DMA has overhead on low data amounts.

Timer

The Timer design is incomplete, and only houses a RTC. Due to a hardware failure, the internal crystal is being used. Which implies a worse accuracy than an external one. However since the most recent time can be retrieved from the internet this is not a major issue. Future additions to the Timer module include: Alarm function for timeouts on waiting, PWM functionality and using the 32 RTC with the external crystal. The Timer module is set up to use it's overflow interrupt. The period is set to 1023. As soon as the counter reaches 1024, a second pulse event will get fired and the run time will get incremented.

4.3.4 Driver layer

SEPS525F

As mentioned before, the screen uses a SEPS525F IC. This driver drives allows for driving screens with a resolution of up to 160x128 pixels with 18 bit combined color. This means that there are 6 bits for blue, 6 bits for green and 6 bits for red. However this would require to send 3 bytes for 2 bits of color precision. There is also a 16 bits combined color option available, which has been used in this driver. 5 bits for blue, 6 bits for green and 5 bits for red. A 24 bit color (8 bits per color, 0-255) can be converted with the following define:

```
#define SEPS525F_TO656(r,g,b)((r>>3)<<11)|((g>>2)<<5)|(b>>3)
```

The SEPS525F has three data interfaces that could be used: SPI, RGB, Parallel. In retrospect a parallel interface would have given major performance advantages. However due to time constraints a SPI interface has been chosen. The clock frequency of the SPI is set at the CPU speed divided by two. Which is a clock of 8 MHz when not in any power saving mode.

The SEPS525F has two modes, a data mode and a command mode. With the command mode a register can be set. In order to achieve this the register will be written first, while clearing the RS and the CS pin. Once the register is written, the RS pin will be set while keeping the CS pin low. After the data is written the CS pin will be set.

```
1 static void SEPS525F_reg(int idx, int value) {
2 SEPS525F_PORT.OUTCLR = SEPS525F_CSB | SEPS525F_RS;
3 spi_write_blocked(idx);
4 SEPS525F_PORT.OUTSET = SEPS525F_RS;
5 SEPS525F_PORT.OUTCLR = SEPS525F_CSB;
6 spi_write_blocked(value);
7 SEPS525F_PORT.OUTSET = SEPS525F_CSB;
8 }
```

This can be seen in the code above. The driver gives the possibility to draw a single pixel, but it's significantly faster to write multiple pixels at once. For this a region that will be drawn on has to be set. Then the driver ic will expect a certain amount of pixels, which have to be written as data. According to the datasheet special scrolling features should be available, however this is not explained later on in the datasheet. There are a lot of parameters that can be set for the screen, such as duty cycle, frame rate, driving currents. The explanation of all these can be found in the datasheet, just like the recommended values. The given initialization sequence of the datasheet did not work, so an Arduino library has been ported successfully[3].

Terminal

The Terminal driver gives the possibility to format strings, and outputs the formatted strings to a sink. The default sink writes to the USB-Serial IC via the UART driver. Formatting is done via the tinyprintf library[4] and implements the 'd' 'u' 'c' 's' 'x' 'X' formats.

ESP8266

The ESP8266 is a low cost Wi-Fi module which can be flashed with own firmware. This firmware is written using Sming in C++, more on this in chapter **CHAPTER**. In order to establish communication, protocol buffers are used. The ESP8266 driver implements a stream that is used by the Control Module which will be explained in the next section. The XMega will also serve as a bridge to flash the firmware on the ESP8266, as the module has to be detached from the board to program it right now.

4.3.5 Module Layer

Although all the drivers talked about before make use of the Module system, they are not located on the Module layer. This is a naming convention mistake.

Command

The Command module has the purpose of setting up an accessible possibility to execute commands received by the USB-Serial connection. It is possible to register up to 26 commands. One for every letter in the alphabet. A command can be registered by use of the following code:

```
1 command_hook_description(
2 'T', &terminalCommand, "Log sink    T<option> options: U(Uart) S(Screen)\0"
3 );
```

It takes 3 arguments, the first one being the letter the command is tied to. The second one is a function pointer for the callback. And the third one is a usage description that can be requested by the user by writing a '?' to the terminal. In fig. 4.3.5 an example output can be seen.

```
[0296][Sys][Command]../modules/command.c:132 Following commands are registered:
? | Prints out this help
A | Sends AT    A                   no options
G | Gets state of an app S<app>
        W Get weather               no options
        T Get time                  no options
L | Led control L<option> options: T(toggle) 1(on) 0(off)
S | Sets an app state S<app> <options>
        W<options> options: 1-6 for different weather
        S<options> options: f(facebook) e(mail)
        T<options> options: hour minute second
T | Log sink    T<option> options: U(Uart) S(Screen)
```

Figure 4.3: Response of a question mark

It's up to the developer to parse the string that is passed with the command. An example input string could be:"ST 12 23 34" which sets the time to 12:23:34. The callback will receive the same string as thats sent, minus the command character. Some helper functions have been written such as "command_next_int" to make parsing easier. Parsing the example string is done in the following code:

```
1 uint8_t index = 1;
2 if(data[index-1] == 'T'){
3 uint32_t hour = command_next_int(&index, data, len);
4 uint32_t minute = command_next_int(&index, data, len);
5 uint32_t second = command_next_int(&index, data, len);
6 }
```

Control

The name for this module might be confusing, but it controls all the inter system communications. The Control module defines streams and has the capability to encode and decode Protocol buffers messages. At the moment of writing only one stream is implemented, which is the ESP8266 stream. This is used to talk with the ESP. Eventually all the communication to the PC should also happen via a stream. There is a small protocol beneath the protocol buffers. Protocol buffers itself does not offer a possibility to send and receive multiple messages, so a length of the outgoing data has to be sent first. Other than the length of the messages, the protocol also contains a prefix of 4 bytes, so that the stream can get up to date again if there is some sort of data loss. Google does not use a protocol buffers implementation for C, this is why NanoPB https://github.com/nanopb/nanopb has been used as an external library. NanoPB has a smaller code footprint, and has options to handle and print out error messages.

Light

MAX

PID

MAX

Voltage

MAX

LOG

Almost every driver or module has a dependency on the LOG driver. In order to use the logging functionality, a file first has to initialize the logger. this is done by calling:

```
1 LOG_INIT("Core");
```

This is required to keep track of where the logging happens. There are multiple levels of logging, which currently all have the same effect other than the name, except for the "LOG_ERROR" function. The error log function halts the system. In a future build there should be a possibility to set a general log level so that log statements with less importance than the general log level will not be written. Every log is translated to the following three statements:

```
#define LOG_INTERNAL(LEVEL, MSG, ...) \
log_message("[%04d][%s][%s]%s:%d ",timer_runTime(),LEVEL,log_name,__FILE__,__LINE__);    \
log_message_p(PSTR(MSG), ##__VA_ARGS__);                                                  \
log_message("\r\n")
```

An example like:

```
LOG_SYSTEM("Received command: %c", command);        //command is a character
```

Produces the following string on the sink: "[2727][Sys][Command]../modules/command.c:157 Received command: T". The string contains the absolute runtime, what kind of log it was (debug, system, warning), the filename and the file line. The strings are saved in the flash to save memory. If the sink for the terminal changes, so does the output for the logger, as its depending on the terminal. The screen can be used to log to. This can be seen in fig. 4.3.5.

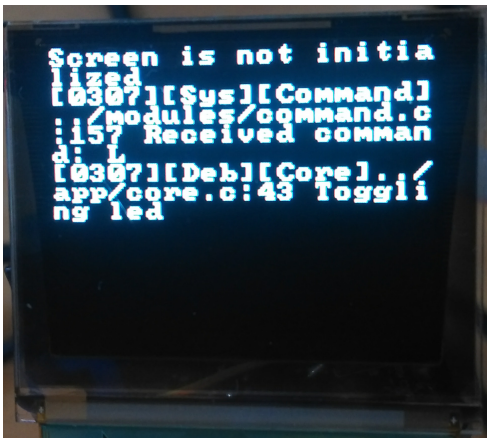


Figure 4.4: Redirecting the output to the screen

However this has to be used carefully, as writing to screen takes significantly longer than outputting it over UART. So the system might freeze in case of too much text to show. If needed it's also possible to disable all the logging for certain log levels. This can be useful if there is no interest in debug messages from a module. This can be done by calling LOG_LEVEL_SET with the minimum wanted log level. Lastly there is a functionality to adjust the formatting of the log on the output. For logging on the screen it might not be desirable to show all file names and line numbers. This is done by calling the log_set_display function.

```
log_set_display(LEVEL_NAME); // Only show the level (debug, warning) and module name
log_set_display(DEFAULT_LOG_FORMAT); // Returns to default formatting
```

Screen

The Screen module contains routines for drawing shapes, images and text on the screen. Currently implemented shapes include: Rectangle, filled rectangle, circle, filled circle, pixel, and line. These shapes are drawn with the color that is currently set. Images can be drawn both from flash and from memory. Better practice reads them out from flash, as there is little memory available. A start and stop function is available. This can be used to denote whether the shapes should be filled, or whether only the outline should be drawn. Another future addition can speed up the drawing significantly when a data start and data end command will be sent upon calling start and stop. This is not implemented in the current version.

Time

The Time keeps track of time offline. It does so by keeping track of hours minutes and seconds, and incrementing them accordingly upon every second pulse generated by the Timer driver. The time can also be set via a function.

4.3.6 App layer

This layer is the least elaborated, as it has the last priority since it would not work without the layers before it regardless.

Clock

The clock application draws a digital clock, alarm and date. It listens to time updates, and updates the time unit that has been updated, whether that is second minute hour or a combination of them. **MAX HERE**

Weather

The weather application draws a temperature, unit and an icon based on the weather situation (rain, snow, sun, overcast) which are being drawn via screen routines.

Mail

The mail application draws mail icons and shows the 4 most recent emails with the first 5 letters of the sender and first 5 letters of the subject.

Status

The status application draws the status bar. This reflects whether the alarm is set, whether the light is on and what the signal strength is

Console

The console application controls the screen terminal, in the future it will also be capable of receiving commands over WI-FI, however that functionality is not in yet due to time constraints.

Core

The Core translates all the commands given to actions. The debug LED can be toggled, AT commands can be sent, the sink for the terminal can be changed, time and weather can be set. The core also decides where to draw what application. Currently this results in fig. 4.3.6 below.

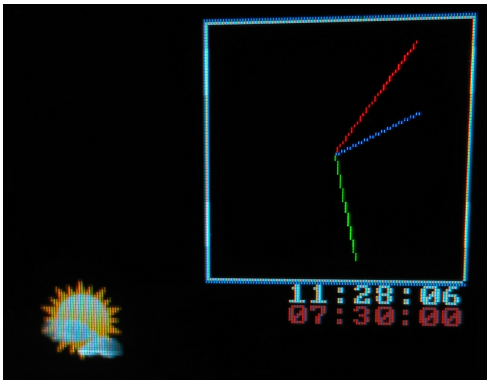


Figure 4.5: Displaying the clock and the weather

4.4 Software Server

To obtain the most recent time, weather and email, a server backend has been made. This server backend is accessible trough a RESTful interface. An application can fire post requests with protobuf messages in the body. This will be decoded on the server and the right response will be given. The server backend is written in Java, and hosted on Heroku https://www.heroku.com/ . To accelerate RESTful java interface development, the spark framework has been chosen to develop with http://sparkjava.com/ .

4.5 Software ESP

5 Evaluation

5.1 Hardware

The hardware right now is in revision 2.0. Revision 3.0 is currently under developemnet. Revision 3.0 will be the third recision. The current revision ,where all the software is running on is revision 2.0. It has been fabricated in January 2017 and the design process (including revision 1.0) began in november 2016. There were severe mistakes on the first hardware revision, which were not fixable so easily, especially because of the fact, that everything is quite small. In the second hardware revision there are still smaller mistakes. Moreover upgrades shall be taken into account for the revision 3.0.

5.1.1 Issues

The feedback mechanism, which is crucial of course for any kind of controll loop does not work in the case of the RGB channels. This is because the LM324 quadruple operational amplifier is connected to a single power supply. In this case it is necessary to connect the shunt resistors according to the current flow direction to the differential amplifier. This has been confused and needs to be fixed.

The supply voltage of 2.8 V does not seem to be longer suitable for the application. The system crashes, when the supply voltage is below approximately 3.1 V. The easy fix for the next revision is a supply voltage of 3.3 V. Further investigations have to be made here.

The used coils have to be checked once more if the current rating fits. The coil used for the USB charger is definitely not suitable. It catches flames, when the a current above 600 mA is drawn from the charging prot.

5.1.2 Upgrades

A few low-pass filters shall be moved from the power board to the logic board (saves space).

A ADC converter channel has to be tied to GND for calibration, which can be traced back to some special properties of the 12 bit ADC of the Xmega.

Also a buzzer shall be part of the next revision to ensure, that the user wakes up.

Since the RGB LEDs are quite cheap and the driver circuitry turned out to be pretty powerfull, more LEDs (2-4 in total) shall be added to the system.

A housing is an essential part of such a product. That is why a small box will be constructed, which fits both boards, a couple of LEDs and the screen.

5.2 Software

he software is working and almost finished on all different levels. One could discuss about the basic principles of the operating system implemented. For instance if a timer triggers an event every 10 ms the listeners i.e. the tasks, which are supposed to be executed with this period will experience a heavy jitter. This jitter is dependent on how many listeners this event currently has and how many other events have to be handeled by the kernel at the same time. If a task implemented takes too much time to execute or executes to frequently, the system crashes. One has to be carefull with e.g. PID controllers as those

can generate heavy workload, which can not be handeled anymore by the ressources the Xmega provides. The operating system does in that sense not provide real-time functionality.

5.2.1 Issues

It is crucial to make the system safer. Through a software malfunction it is quite easy to burn LEDs or the screen. This needs to be avoided. Time needs to be invested into the developement of safety mechanisms to avoid the destruction of the hardware.

The code size needs to be optimized. More than 80 % of the flash of the Xmega is currently used. There is not too much space anymore for upgrades.

In the end four PID controllers shall run parallely on the platform. The facilities for debugging and tuning a PID controller are give, but the fine tuning of K_P, K_I and K_D still has to be done for the four different controllers.

5.2.2 Upgrades

The software upgrades mainly orientate on the hardware upgrades, which will be made with hardware revision 3 (cf. 5.1.2).

6 Conclusion

Bibliography

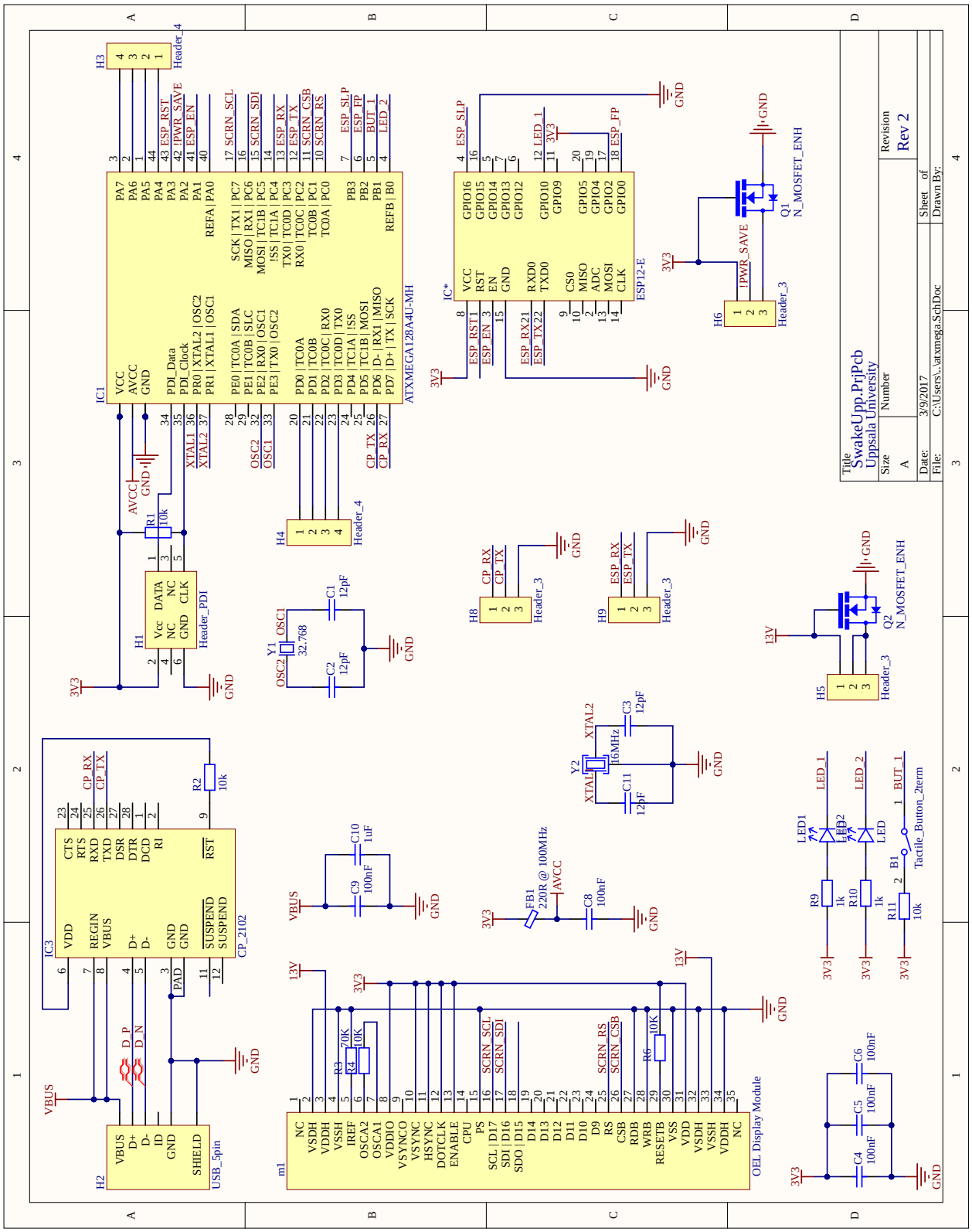
[1] *Visit Sweden*. (2017). Time and daylight in Sweden, [Online]. Available: `https://visitsweden.com/time-and-daylight-hours/`.

[2] *Sveriges Radio*. (2014). Are Swedes more suicidal than most? [Online]. Available: `http://sverigesradio.se/sida/artikel.aspx?artikel=5924063`.

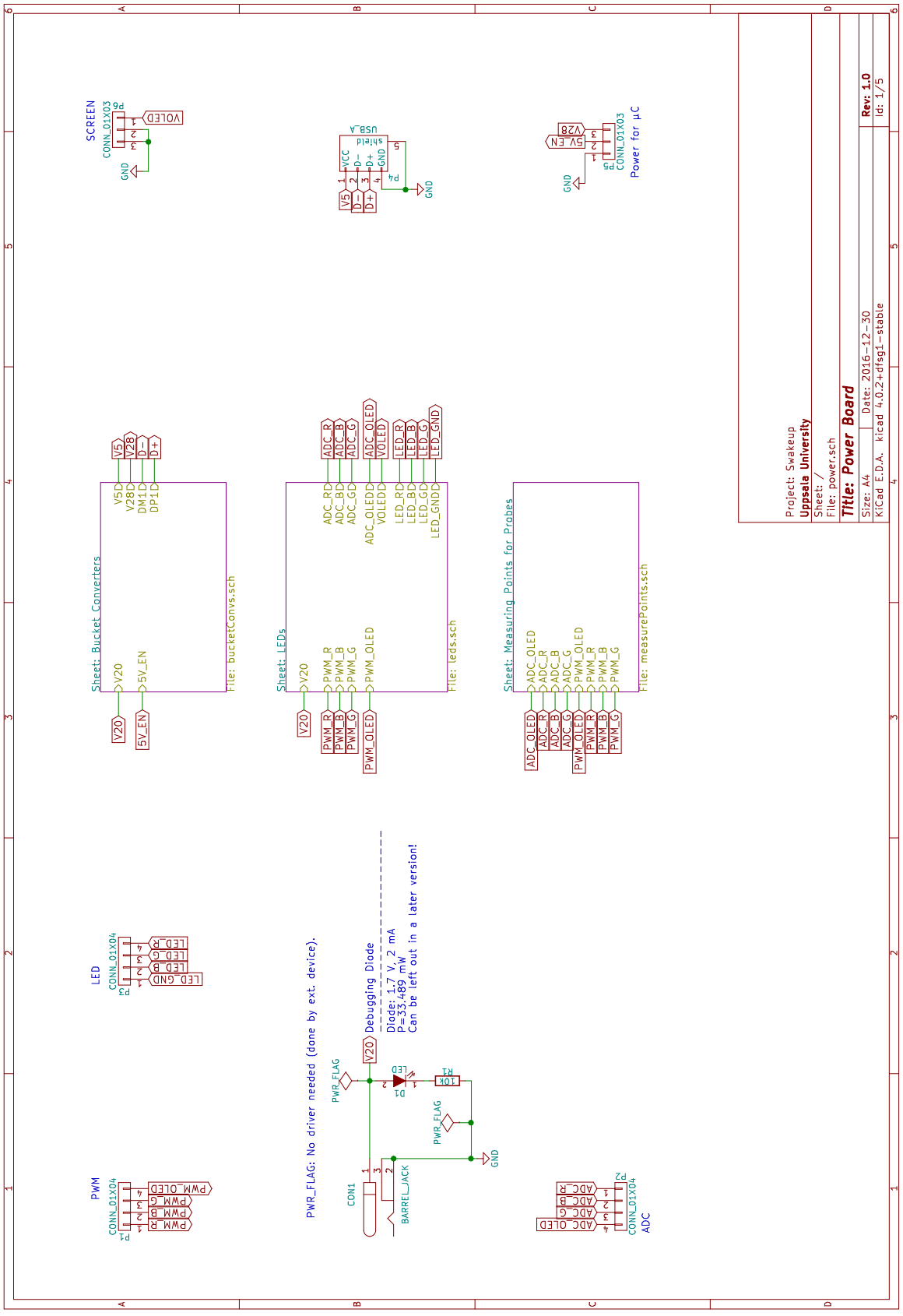
[3] *Tomaž Šolc*. (2013). Arduino library for the SEPS525 OLED shield, [Online]. Available: `https://github.com/avian2/SEPS525-OLED-Library` (visited on 02/02/2017).

[4] *Kustaa Nyholm*. (2012). A tiny printf for embedded applications, [Online]. Available: `http://www.sparetimelabs.com/tinyprintf/tinyprintf.php` (visited on 02/13/2017).

A Schematics Logic Board

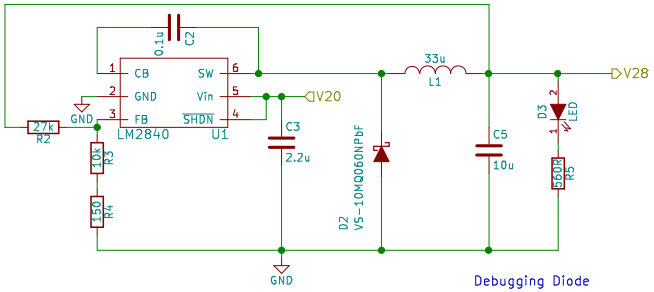


B Schematics Power Board



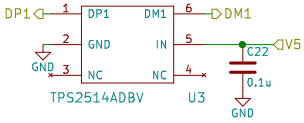
2.8 V for µC

Vout=0.765 V (1 + (27k/ 10.15k))=2.8 V
27k/10.15k= 2.66



Debugging Diode

Diode: 1.7 V, 2 mA
P=2.161 mW
Can be left out in a later version!

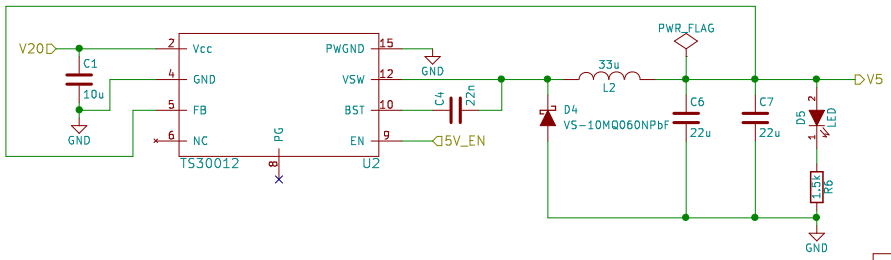


USB Dedicated Charging Port Control.

Simple SOT-23-6 IC for detecting propriatary and open standards used by a device and providing the corresponding electrical signature at the data lines (voltage or impedance).

5V for Mobile Phone

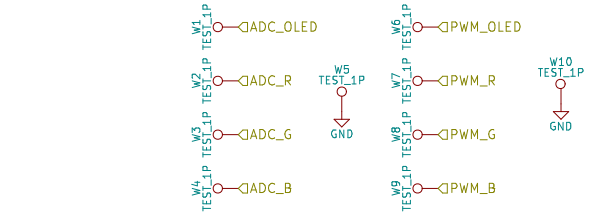
Vout,max=5 V
Aout,max = 2A



Debugging Diode

Diode: 1.7 V, 2 mA
P=7.26 mW
Can be left out in a later version!

Uppsala University		
Sheet: /Bucket Converters/ File: bucketConvs.sch		
Title: Bucket Converters for 5 V and 2.8 V		
Size: A4	Date: 2016-12-30	**Rev: 1.0**
KiCad E.D.A. kicad 4.0.2+dfsg1-stable		Id: 2/5



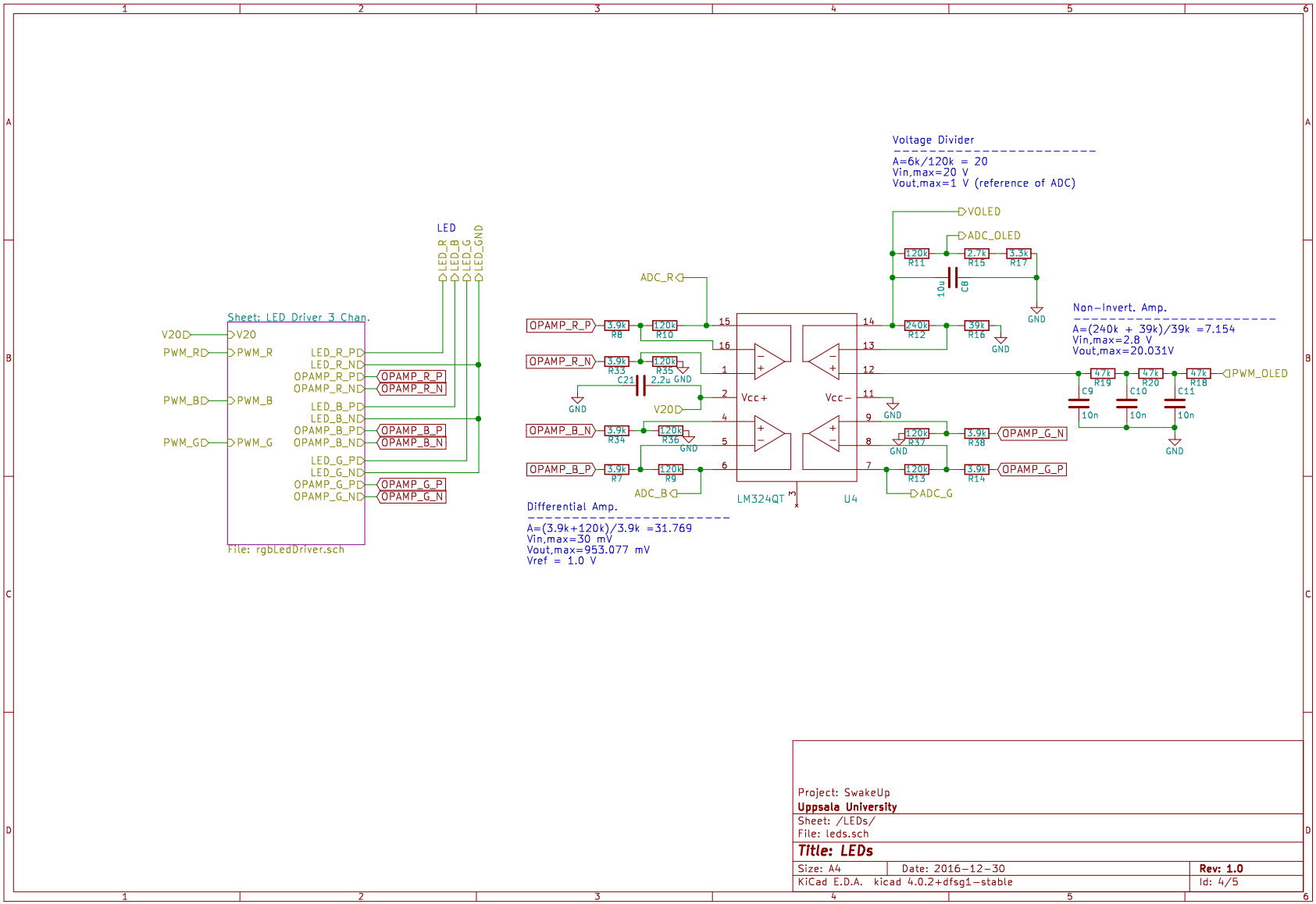
Test Points for the FBs.
Can be left out in a later version!

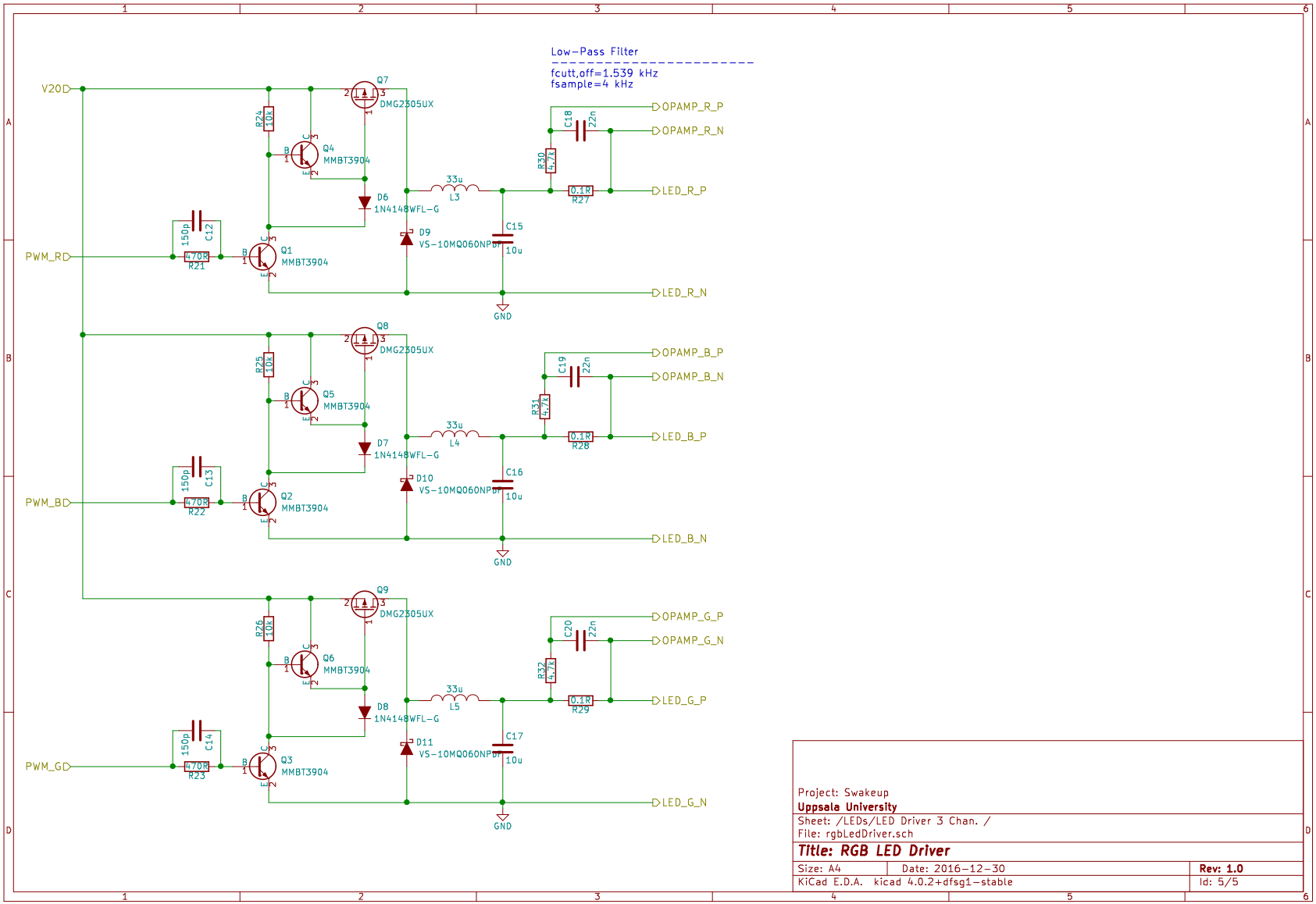
Test Points for the PWM signals.
Can be left out in a later version!

Project: Swakeup
Uppsala University
Sheet: /Measuring Points for Probes/
File: measurePoints.sch

Title: Test Points

Size: A4	Date: 2016-12-02	**Rev: 1.0**
KiCad E.D.A. kicad 4.0.2+dfsg1-stable		Id: 3/5


LED
LED_R
LED_B
LED_G
LED_GND
Sheet: LED Driver 3 Chan.
V20
PWM_R
PWM_B
PWM_G
LED_R_P
LED_R_N
OPAMP_R_P
OPAMP_R_N
LED_B_P
LED_B_N
OPAMP_B_P
OPAMP_B_N
LED_G_P
LED_G_N
OPAMP_G_P
OPAMP_G_N
File: rgbLedDriver.sch
Voltage Divider
A=6k/120k = 20
Vin,max=20 V
Vout,max=1 V (reference of ADC)
VOLED
ADC_OLED
ADC_R
ADC_B
ADC_G
120k R11
2.7k R15
3.3k R17
10u C8
GND
240k R12
39k R16
Non-Invert. Amp.
A=(240k + 39k)/39k =7.154
Vin,max=2.8 V
Vout,max=20.031V
47k R19
47k R20
47k R18
PWM_OLED
C9 10n
C10 10n
C11 10n
OPAMP_R_P
OPAMP_R_N
OPAMP_B_N
OPAMP_B_P
3.9k R8
120k R10
3.9k R33
120k R35
C21 2.2u
3.9k R34
120k R36
3.9k R7
120k R9
Vcc+
Vcc-
V20
LM324QT
U4
120k R37
3.9k R38
120k R13
3.9k R14
OPAMP_G_N
OPAMP_G_P
Differential Amp.
A=(3.9k+120k)/3.9k =31.769
Vin,max=30 mV
Vout,max=953.077 mV
Vref = 1.0 V
Project: SwakeUp
Uppsala University
Sheet: /LEDs/
File: leds.sch
Title: LEDs
Size: A4
Date: 2016-12-30
Rev: 1.0
KiCad E.D.A. kicad 4.0.2+dfsg1-stable
Id: 4/5



Project: Swakeup
Uppsala University
Sheet: /LEDs/LED Driver 3 Chan. /
File: rgbLedDriver.sch

Title: RGB LED Driver

Size: A4	Date: 2016-12-30	**Rev: 1.0**
KiCad E.D.A. kicad 4.0.2+dfsg1-stable		Id: 5/5

C USART interrupt generation

```
#define USARTRXCISR(NAME, PORT, USART_ID, REC_FC)            \
ISR(NAME##_RXC_vect) {                                  \
        uint8_t read = PORT.DATA;                        \
        if (writeInBuf(read, &PORT)) {                   \
        REC_FC(read);                                       \
        uint8_t i = 0;                                  \
        for (; i < UART_MAX_DELIMITERS; i++) {      \
                if (delimiters[USART_ID][i].delimiter != 0) {  \
                        delimiters[USART_ID][i].length++;          \
                        if (read == delimiters[USART_ID][i].delimiter) {   \
                                delimiters[USART_ID][i].port = &PORT;       \
                                event_fire(&EVENT_UART_DELIMITER,        \
                                SYSTEM_ADDRESS_CAST (&delimiters[USART_ID][i])); \
                                }   \
                        }   \
                }   \
        } else {/*buffer full */     \
                CP_PORT.CTRLA &= ~(USART_RXCINTLVL_LO_gc);  \
        }   \
}

#define USARTDREISR(NAME, PORT, USART_ID)\
ISR(NAME##_DRE_vect) {              \
        uint8_t size = uartStatus[USART_ID].outBuffer_size;    \
        if (size > 0) { \
                if (softlock(USART_ID)) {\
                        uint8_t tail = uartStatus[USART_ID].outBuffer_tail;\
                        PORT.DATA = outBuffer[USART_ID][tail];  \
                        uartStatus[USART_ID].outBuffer_size--; \
                        tail++; \
                        if (tail >= UART_MAX_OUT_BUFFER) tail = 0;\
                        uartStatus[USART_ID].outBuffer_tail = tail;\
                        unlock(USART_ID);  \
                }\
        } else {\
                sending[USART_ID] = 0;\
                PORT.CTRLA &= ~(USART_DREINTLVL0_bm);\
        }\
}
```