Swakeup

PROJECT REPORT

within the lecture of
Microcontroller Programming

at Uppsala University
in the Departement of Information Technology

Elmar van Rijnswou (elmarvanrijnswou@hotmail.com) and Maximilian Stiefel (stiefel.maximilian@online.de)
Deadline: 2017-03-10 24:00

Processing Period:	December 2016 - March 2017
Supervisor:	Uwe Zimmermann (uwe.zimmermann@angstrom.uu.se)

Contents

List of Figures

List of Tables

1 Introduction

Idea

It is a well-known fact, that it is quite dark in Sweden in the winter. In a strong winter every source of light is a source of happiness. This wakeup light, which is based on a strong light source (10 W RGB LED), is able to give one the optimal start into a dark winter day. The *Swakeup* (from engl. "Swedish Wakeup Light") is communicating to the user through the light. It does not simply wake one up, but also gives one information about Facebook, latest mails, calendar and weather. The user interface consits besides of a big 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 you are doing before you go to sleep? Right! You look on your phone. That is why *Swakeup* comes with a USB charger for your e.g. phone as well.

Sytem Overview

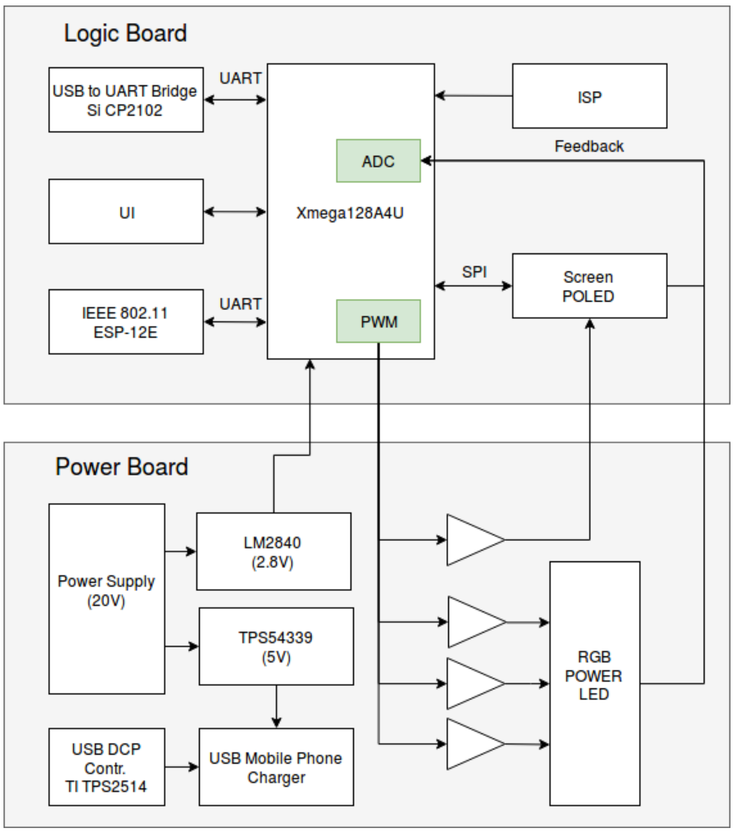


Figure 1.1: Blockdiagram of the Swakeup wakeup light.

In fig. 1.2 the actual system architecture from an abstract point of view is displayed. One can clearly see, that the system is divided into two physical boards. The logic board consists of the μC, a serial connection infrastructure, an OLED screen, an *IEEE 802.11* module, a ISP programming infrastructure and some LEDs/a button (UI).

The power board takes the 20 V input of a low-cost power supply and breaks it down to 2.8 V (Vcc), 5 V for phone charging and the power which is needed for the RGB LED.

The partitioning of the system functionalities on two boards has a bunch of advatages: Two people worked on this project, so each could develop a PCB; It is quite common to seperate signals from power lines out of EMI reasons; There was simply not eanough space on one two-layer board for the whole system keeping a base area of 5 cm x 4 cm.

The two boards are connected together through four headers. By these headers an electrical and mechanical connection is maintained. The headers allow a feedback from the LED driver on the power board to the logic board (ADC). Also the control variable (PWM) comes via the headers from the μC to the LED driver. The 2.8 V Vcc is produced and regulated on the power board and is also availabe at one header. Another voltage available from one header is the OLED driver voltage.

2 Hardware

Logic Board

Power Board

The board design has been made in *KiCAD*. *Git* was used for version control. In the schematics (Appendix A) one can see that the whole board consists of three main building blocks: Connectors, a LED driver with feedback and two step-down converters. A part of the LED driver is also "abused" to drive the OLED.

Microcontroller Power Supply

It is the LM2840 which in combination with a simple voltage divider ensures the 2.8 V for Vcc. All step-down converters use the same inductor with a value of 33 uH. It is a low-cost, quite small, shielded inductor which is ment to be used for switching power supplies. Moreover all step-down converters are enhanced with a SMD schottky diode and a of course at least one SMD capacitor for smoothing the output signal. As it is good practice to do so all ICs are making use of decoupling capacitors.

Designated USB Charging Port

For charging ones phone the TS30012, another step-down converter IC, is used. The feedback voltage divider of this IC is already integrated and does not need to be provided externally as the IC provides fixed 5 V output. The output is connected to a USB connector type A. This IC can deliver up to 2 A. An interesting feature of the phone charging circuitry on the power board is the "Dedicated Charging Port" (DCP) functionality. The TPS2514 is a small, easy-to-use, 6-pin component, which complies to the USB standard and a majority of the minefield of propriatary standards to signal a DCP. Now you might ask: What does this mean? Well this means, that if you connect your IPhone, it will know, that it can draw more than 100 mA, which is the minimum current provided by a normal USB 2.0 port. Otherwise the current drawn by the phone will be limited. The charging functionality can be turned on and off via a GPIO pin. The TS30012 comes in a QFN16 package (pad pitch of 0.5 mm) to save space.

HW Debugging

For testing purposes a lot of test points have been included into the design. Futhermore there are LEDs for different voltages (e.g. Vcc).

RGB LED Driver

The LED driver consists of an actual power electronics part and a feedback part. The main idea is, that the current driven through the three color channels of the RGB LED (see fig. 2.1) can be controlled by software (PID controller). In the power electronics part there are three analog circuits. Each circuit mainly consists of a p-channel MOSFET, which is switched by a NPN bipolar transistor. This bipolar transistor gets its intput signal from the μProcessor (PWM). By pulling the 20 V to GND the PMOS "sees" a negative gate-to-source voltage and opens. The additional bipolar transistor ensures, that the gate-source capacity is charged fastly as soon as the PWM NPN blocks. In this case the base is pulled up via $10\,\mathrm{k}\Omega$ to 20 V and as long as the collector (which has the same potential as the gate of the PMOS) does not also have 20 V the NPN keeps pumping charges into the gate-source capacity. The simple silicon diode ensures that the gate-charging NPN has no effect as soon as the PWM NPN opens. The rest of the

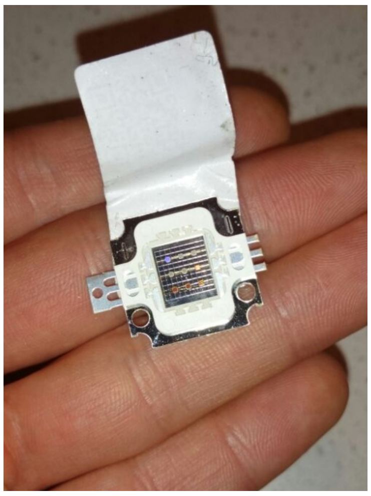


Figure 2.1: 10 W RGB LED

circuit is again a standard step-down converter. At the output of every single color channel power circuit one can see a shunt resistor of 0.1 Ω. This shunt resistor is combined with a simple low-pass filter. The feedback signal is amplified by a differential amplifier which comes after the filter. The signal is supposed to be between 0 V and 1 V (if the internal reference voltage is used). By doing so one can use all bits of the ADC and therefore supress quantization noise.

So as there are three color channels (red, green and blue) one needs three operational amplifiers. Hence the LM324QT, which provides four operational amplifiers, has been chosen. Of course the decision has been made to take QFN16 packaging once more to save even more space. An additional operational amplifier was now ready to be used as a digital-analog converter to drive the OLED screen (this was referred to earlier as "abuse"). For this reason a low-pass filter is attached to the input of the fourth opamp. Apart from that the opamp is configured just like a standard non-inverting amplifier. By software the OLED screen brightness is steerable. The feedback from the OLED is generated by a voltage divider.

3 Software

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 don'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.

Code Structure

The code is structured in a layered approach. This has multiple advantages. 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.

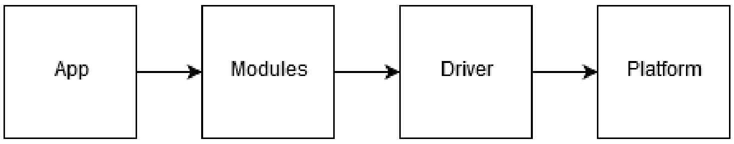


Figure 3.1: The layered approach

In fig. 3.1 the structure is visualized and it can be seen how every layer depends on a lower level layer. Every block can be swapped with different components while the system keeps running, if implemented well without breaches of layering.

To make this layered code structure a reality, a small operating system has been developed which allows for 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.

In fig. 1.2 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 properly 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.

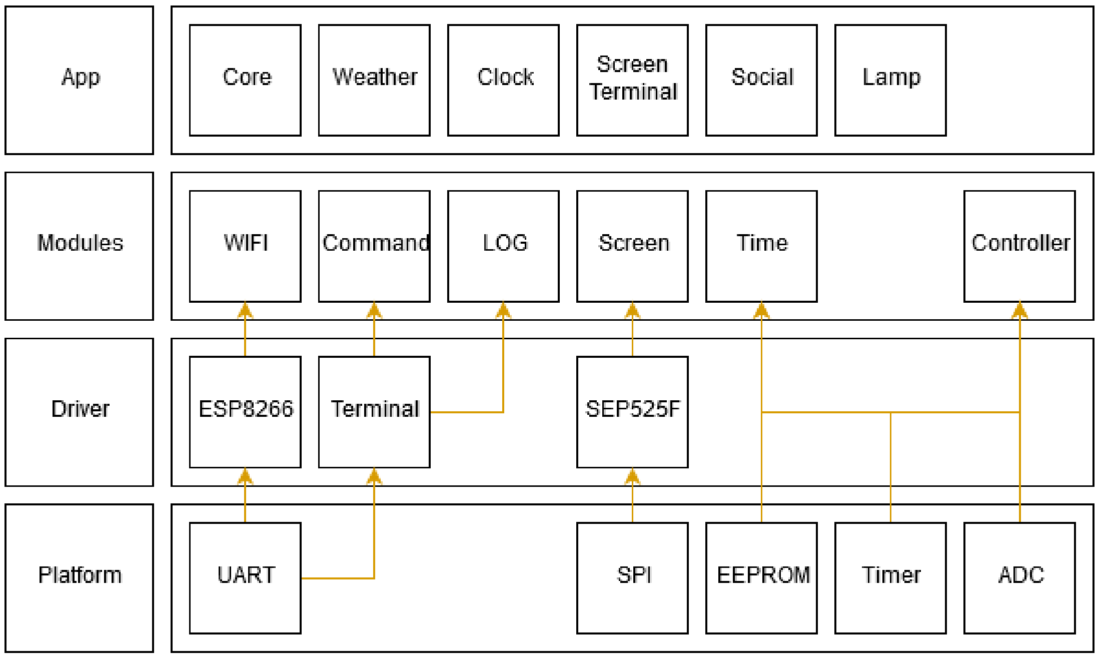


Figure 3.2: The system structure and hierarchy

above in fig. 3.1 this translation from system design to a more refined code structure is shown. A brief description of the required functionality for every block follows.

App layer

The core in the App layer takes care of putting the system in the right state, and decides when to show which application on the screen. Furthermore it establishes the WI-FI connection and requests the data from the Internet that's required for all the other applications. Also it translates commands coming from the UART port into actions. The weather application takes data thats fed from the core, and shows the weather at a given position on the screen. The clock application listens to second pulses and shows the time on the screen, together with an alarm. The time is shown both analogue and digital. The time can be read from the Time module, or be set trough the Core, when it synchronizes with the Internet. The Screen Terminal application redirects all the log output and shows it on the screen. This can be used for debugging. Facebook and mail notifications are shown with the Social application. Finally the Lamp application houses information about different sunsets, and accounts for the right amount of red, green and blue in the light when the alarm goes off.

Module layer

The WI-FI module makes it possible to translate actions as getting the weather and the current time, to udp and tcp requests. The Command module interprets data given from the Terminal and executes a given function paired with the command sent. Everything is written to the terminal via the Log module, which adds extra information to every message, such as the file name, line number and time. The screen module contains functions to draw shapes and images. Time reads the latest known time from the EEPROM after a total power failure, and keeps track of the current time via second pulses. To allow fades in the light, and stable output, a Controller module is available. This reads out the current voltage with which the lamp is being driven, and adjusts the PWM signal accordingly.

Driver layer

The ESP8266 Driver contains all the communication with the ESP8266 chip and keeps track of the network state. Its also possible to update the firmware of the ESP8266. Terminal is a software driver, with no external hardware dependencies by default. It can, however, use the SEP525F as output for text. The text can be formatted with the Terminal driver. In order to set up the screen and draw pixels, a screen driver is required. The oled screen that is being used, has a SEP525F driver ic. For this a driver has been written.

Platform layer

Every required 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 EEPROM allows to save some settings and time and date on the clock. 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 as last, the ADC peripheral. This scales the voltage read from the input pins and provides it to whomever needs the ADC values.

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.

Realization

The function of this chapter is to get a deeper understanding of the modules and code that has been written. Due to time constraints and hardware failure, the following modules were not realized: "Lamp, Social, Wi-Fi, Controller, ADC, EEPROM". Thus these modules will not be discussed in this chapter.

platform

We start with the most tightly with the microcontroller integrated layer.

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.

```
uint8_t uart_add_delimiter(char delimiter, USART_t * port);
static void callback(Event * event, uint8_t * data) {
        if(event == EVENT_UART_DELIMITER){
        struct UartDelimiter * delimiter = (struct UartDelimiter*)data;
                if (delimiter->port == &ESP_UART_PORT) {
                        //Read buffers etc
                }
        }
}
static uint8_t init(void) {
        uart_add_delimiter('\n', &ESP_UART_PORT);
        event_addListener(&EVENT_UART_DELIMITER, callback);
        return 1;
}
```

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 B. The following 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 pitfalls 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 be looked at, this would eliminate the need for a CPU at all and give the system all the time to process the other tasks. There needs to be some caution, as DMA has overhead on low data. This should be investigated.

Timer

The Timer design is incomplete, and only houses a RTC. Due to a hardware failure, the internal crystal is being used. Which offers 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. If the internal counter overflows the period of 1023 a second pulse event will get fired and the run time will get incremented.

Driver

One step up gives the communication with external hardware.

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've 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.

```c
static void SEPS525F_reg(int idx, int value) {
        SEPS525F_PORT.OUTCLR = SEPS525F_CSB | SEPS525F_RS;
        spi_write_blocked(idx);
        SEPS525F_PORT.OUTSET = SEPS525F_RS;
        SEPS525F_PORT.OUTCLR = SEPS525F_CSB;
        spi_write_blocked(value);
        SEPS525F_PORT.OUTSET = SEPS525F_CSB;
}
```

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 **Source**.

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 tinyprint library **Source** 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. In a next version this own firmware will be developed, however for now the standard firmware is used. UART is used to communicate with the module, via an Hayes command set based protocol**Source http://nemesis.lonestar.org/reference/tele**
The commands implemented allow for basic HTTP get requests. Due to time constraints it was not possible to finalize the usage of the module. Networks can be scanned, and a connection with a wireless action point can be established. But there is no functionality beyond that

Modules

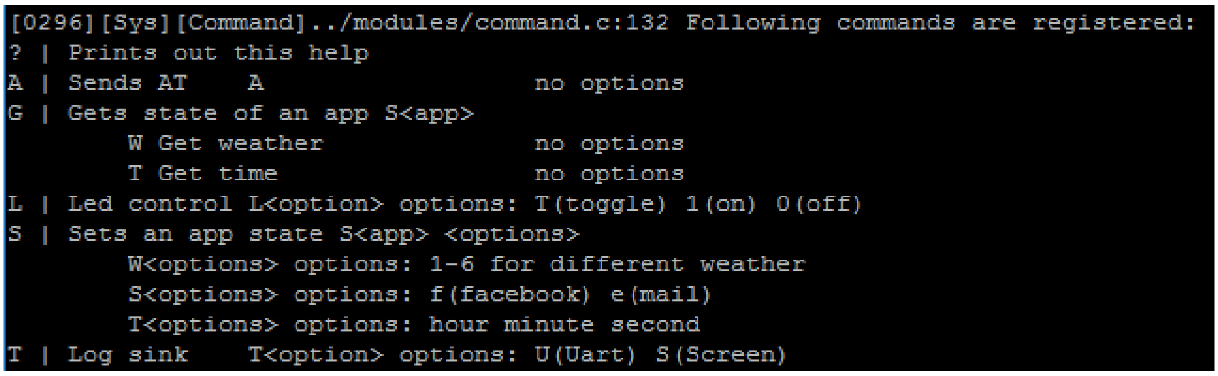
Although all the drivers talked about before make use of the Module system, they are not taken place on the Module layer. This is a naming convention error.

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:

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

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. 3.3.3 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 3.3: 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:

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

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:

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

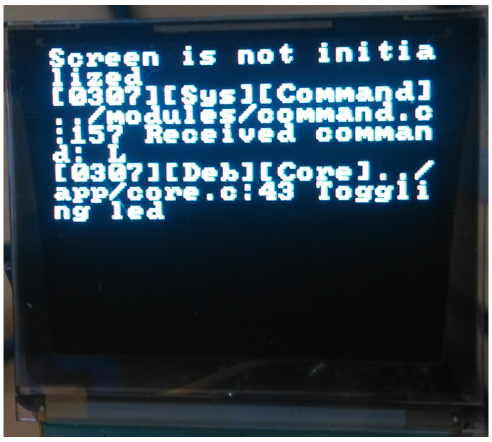
This is required to keep track of where the logging happens. There are multiple levels of logging, which currently all have the same effect other than the name, except for the "LOG_ERROR" function. The error log function halts the system. In a future build there should be a possibility to set a general log level so that log statements with less importance than the general log level wont 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. 3.3.3.



```
Screen is not initia
lized
[0307][Sys][Command]
../modules/command.c
:157 Received comman
d: L
[0307][Deb][Core]../
app/core.c:43 Toggli
ng led
```

Figure 3.4: 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.

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 thats 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.

App

This layer is the least elaborated, as it has the last priority since it wouldn't work without the layers before it regardless.

clock

The clock application draws a clock, analogue and digital. It gets update every second pulse, and also redraws in that case. The SPI connection is not capable enough to show a smooth analogue clock as is, which causes slight stutters when drawing. This behavior could be resolved when programming even more efficiently, one way of tackling this problem would be to draw the lines first to a buffer, which then is drawn as an image. This would provide a more efficient pipeline for drawing the clock, at the cost of memory. Otherwise a compromise will have to be made, and the analogue clock could be replaced by a large digital one. Or removing the second digit. The positions of the lines are static and calculated beforehand.

weather

The weather application currently does nothing else than draw an image based on a weather condition given. It makes use of one large image consisting multiple tiles for different weather conditions (rain, snow, sun, overcast) which are being drawn via screen routines.

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. 3.3.4 below.

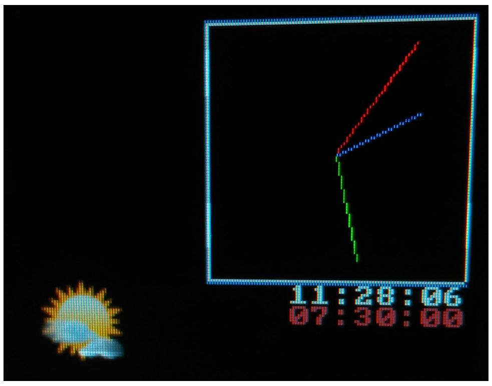


Figure 3.5: Displaying the clock and the weather

4 Status Quo and Outlook

What works? What does not work?

The hardware right now is in revision 2.0. There has been one revision before the current one. This revision has been fabricated in january 2017 and the design process began in november 2016. There were some mistakes on the first hardware revision, which were not fixable so easily, especially because of the fact, that everything is quite small. Table 4.1 gives an overview of the current satus of the hardware.

HW Block	Working	Problem
USB Charging	✓	
OLED Driver	✓	
Vcc for μC	✓	
IEEE 802.11	✓	
USB2UART	✓	
LED Driver		Wrong footprint assignment
Crystal		Wrong pin assignment
USB DCP		Further tests necessary

Table 4.1: Hardware Overview: What works? What does not?

Table 4.2 deals with an overview about the status of the software.

SW Block	Working	Problem
UART	✓	
SPI	✓	
EPROM	✓	
Timer	✓	
ADC		
PWM		
ESP8266	✓	
Terminal	✓	
SEP525F	✓	
Wifi		
Command	✓	
Log	✓	
Screen	✓	
Timekeeper	✓	
Controller		
Core	✓	
Weather	✓	
Clock	✓	
Social		

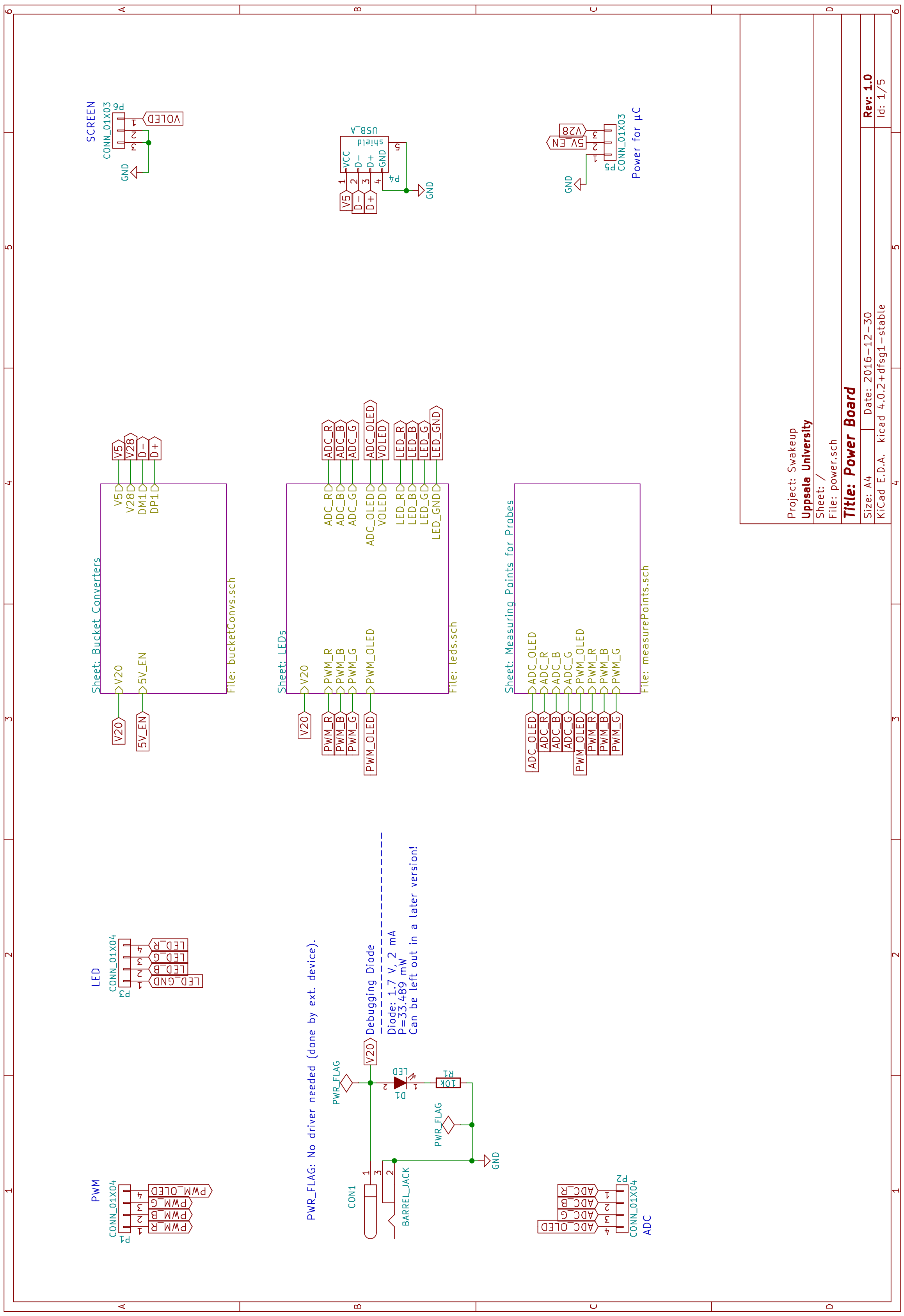
Table 4.2: Software Overview: What works? What does not?

Outlook

Hopefully the LED driver will work with the next revision. The new boards have not arrived until the deadline. As soon as the hardware is working, the controllers will be implemented in software.

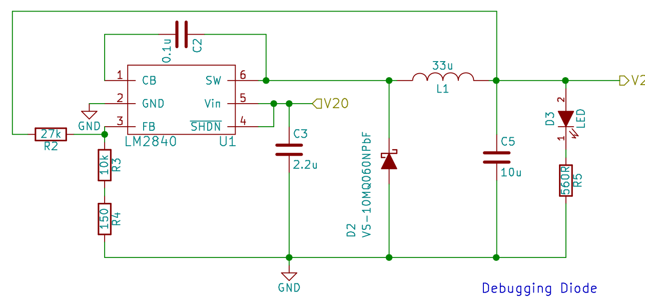
Aditional funtionality will be implemented e.g. connecting you calendar to the device and seing your daily agenda when you wake up. Also there is no housing yet.

A Schematics Power Board



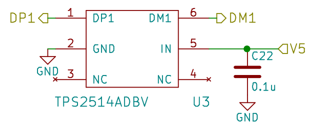
2.8 V for µC

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!

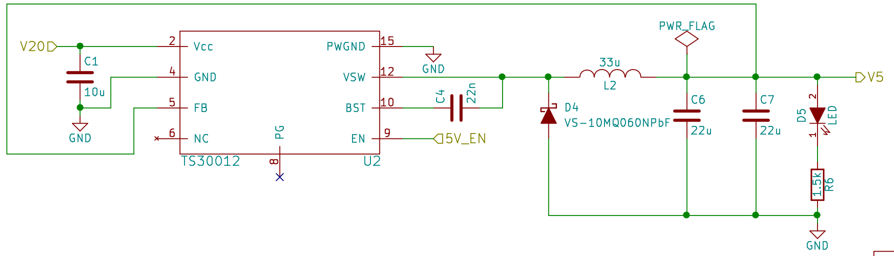


USB Dedicated Charging Port Control.

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



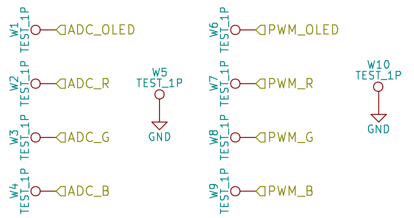
Debugging Diode

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



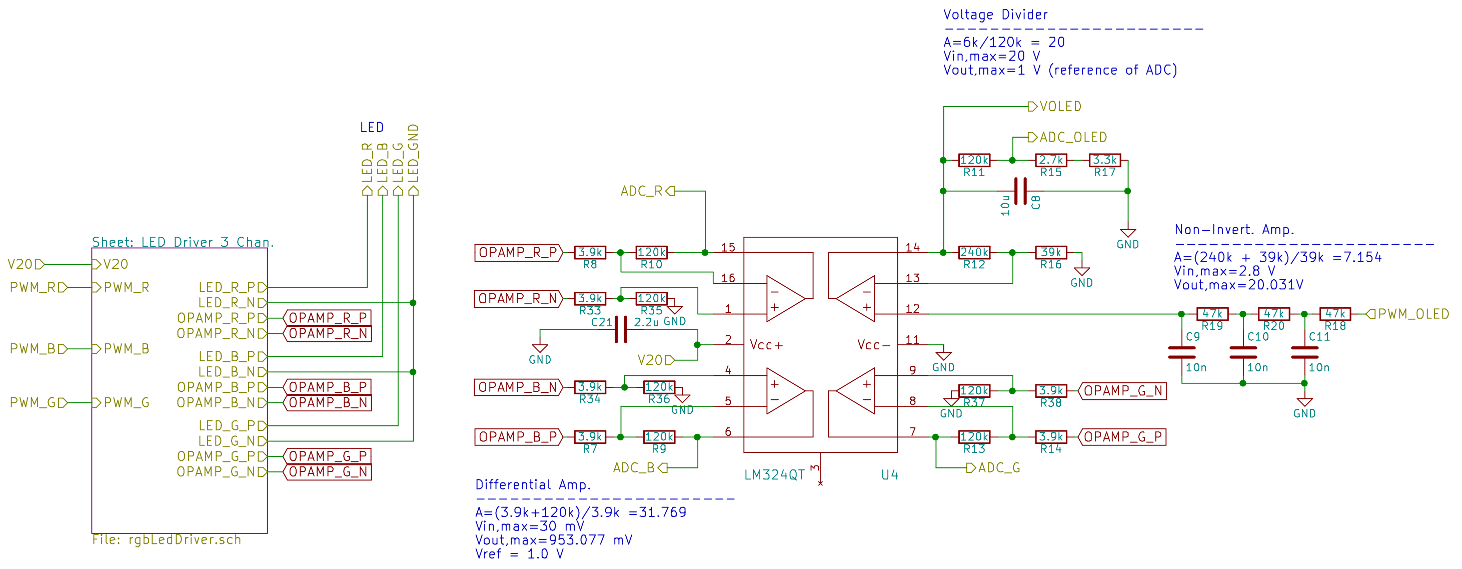
Test Points for the FBs.
Can be left out in a later version!

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

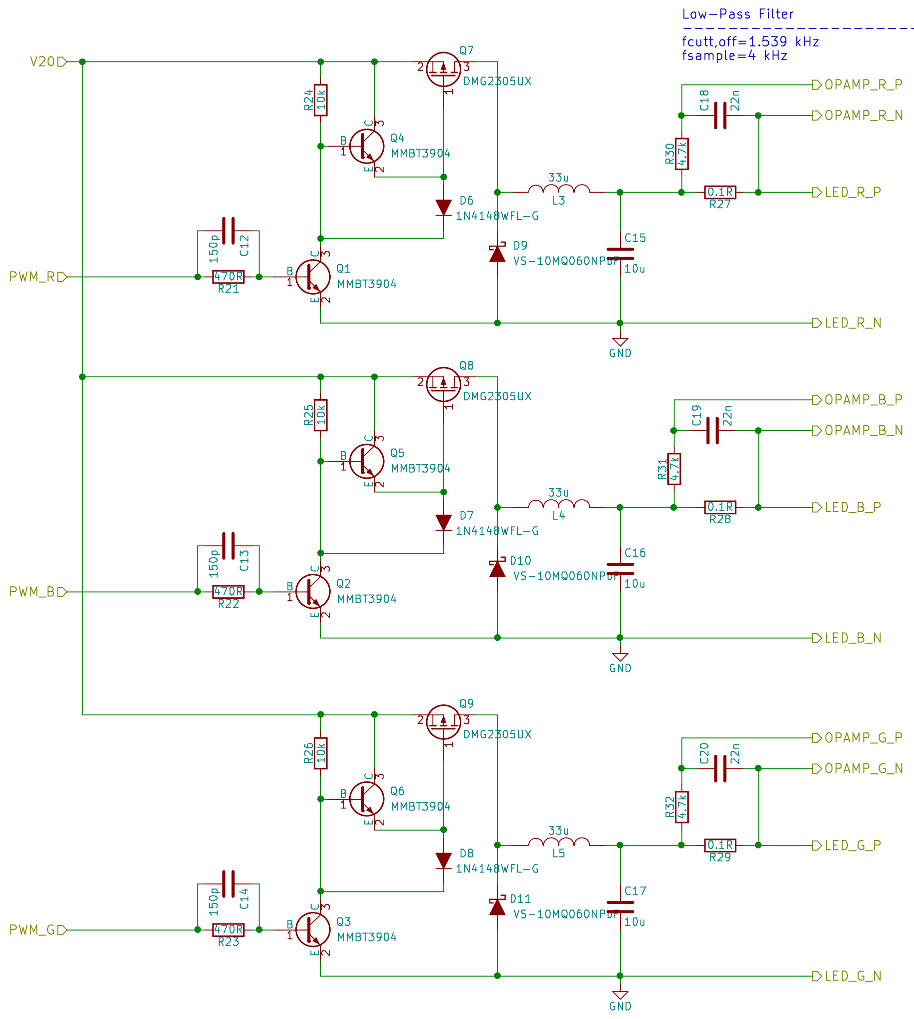
Project: Swakeup
Uppsala University
Sheet: /Measuring Points for Probes/
File: measurePoints.sch

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



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

B USART interrupt generation

```c
#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_DREINTLVLO_bm);\
        }\
}
```