Arduino-based Smart Battery Cell Research Platform

Engineering Research and Development Project

Christian Siegel 201503241

Contents

Chapter 1

Introduction

Lithium Ion (Li-Ion) batteries are utilized in most emerging electrically powered technologies such as electric vehicles, power tools or stationary energy storages. Large battery packs are made of battery cells connected in series and parallel to achieve a desired voltage and capacity. Due to their chemical properties, Li-Ion cells must be operated in strictly specified voltage, current and temperature ranges to avoid critical damages to the battery cells. State-of-the-art battery packs use a central Battery Management Systems (BMS) to monitor cell parameters and control the charge and discharge of cells in a battery pack. Since these BMS are tied to a specific battery pack architecture they need to be customly developed. A recent proposal is to move the battery pack management from a centralized BMS towards individual Cell Management Units (CMU) at the cell level [1]. This leads to more flexibility in composing the battery pack architecture and reduces development time. Furthermore, the reduced battery pack complexity simplifies efficient charge transfer between individual cells instead of traditional passive balancing to equalize the State of Charge (SoC) of all battery cells.

Recent research on smart battery cells was conducted on pricey custom development platforms incorporating powerful microcontrollers and real-time operating systems. Contribution of this project is to introduce a more accessible smart battery cell research platform based on the Arduino[1] platform using mostly publicly available hardware. Moreover, an analysis of the benefits and limits of the proposed prototype is conducted.

[1] https://www.arduino.cc/

Chapter 2

Platform Selection

The core of the smart battery cell research platform shall be some sort of microcontroller platform that needs to be selected. The final research platform shall be able to measure the cell voltage as well as perform passive cell balancing. At the same time the overall cost and external hardware requirements shall be kept low.

This chapter compares different communication technologies and microcontroller platforms to finally select an off-the-shelf development board as basis for the smart battery cell research platform.

Previous Work

Previous research was conducted on rather powerful and pricey hardware like a STM32F407 Cortex-M4 microcontroller board [1][2]. The microcontroller is connected to the battery cell through a custom designed sensor and control board, which can be interfaced using Serial Peripheral Interface (SPI). The invidual smart cells communicate via a Controller Area Network (CAN) bus connected to the microcontroller boards [1].

Communication

Especially communication poses a significant challenge, since wired busses like CAN, Ethernet, Inter-Integrated Circuit (I^2C) or Universal Asynchronous Receiver/Transmitter (UART) require a common ground potential. If the smart cells are connected in series there is no common ground and the bus has to be decoupled using e.g. optocouplers.

While I^2C and UART interfaces are usually integrated in state-of-the-art microcontrollers, CAN and Ethernet require additional hardware controllers. These controllers increase complexity, price and power consumption of the research platform. Moreover, UART is designed for 1:1 connections and would restrict the network topology.

[3] and [4] propose and demonstrate the use of contactless communication via capacitive coupling for distributed monitoring and controlling of battery cells. This could also be used for communication between smart battery cells. Main disadvantage of this approach is the need for additional communication hardware which is not available off-the-shelf.

[5] proposes wireless communication for monitoring lead acid vehicle batteries. The emerging trend of an Internet of Things (IoT) leads to a range of hardware manufacturers offering low-cost wireless system on chip (SOC) solutions. Devices like the *Espressif Systems ESP8266*[1]

[1] https://espressif.com/en/products/hardware/esp8266ex/overview

and the *Texas Instruments CC3200*[2] comprise a regular microcontroller combined with a WiFi module. A full TCP/IP stack provides a well known and convenient to use protocol suite for distributed systems.

Microcontroller

With a focus on wireless solutions the selection of microcontroller units (MCU) can be narrowed down to the ones contained in the ESP8266 and the CC3200. The previously used STM32F407 microcontroller is also added to table 2.1 for comparison.

Table 2.1: Comparison of MCU properties [6][7][8]

	ESP8266	**CC3200**	**STM32F407**
Core	Tensilica L106	ARM Cortex-M4	ARM Cortex-M4
Clock	80 – 160 MHz	80 MHz	168 MHz
RAM	∼50 kB usable in station mode	≤256 kB	192 kB
Flash	external	external	512 kB – 1024 kB
I/Os	16	27	82 – 140
ADC	1x 10-bit	4x 12-bit	16 – 24x 12-bit
I²C	1	1	3
SPI	1	1	3
UART	1	2	2

All MCUs have a 32-bit core. While the CC3200's core can be clocked at maximum 80 MHz, the ESP8266's core clock can be increased up to 160 MHz reaching the clock speed of the STM32F407. With around 50 kB of usable RAM in station mode the ESP8266 has significantly less RAM than the STM32F407 or the CC3200 which provide 192 kB and 256 kB respectively. Both the ESP8266 and the CC3200 don't come with internal Flash. The STM32F407 cleary surpasses both wireless SOCs in regard to peripheral interfaces. The CC3200 still has almost twice the amount of general purpose I/Os than the ESP8266 and four ADCs instead of one. The CC3200's ADCs also come with higher 12-bit precision instead of 10-bit.

Power Consumption

Since the CMU is powered by a battery cell, its power consumption is an important figure to maximize the smart battery cell's capacity. Table 2.2 shows the power consumption of both wireless SOCs and the STM32F407 for comparison.

Table 2.2: Comparison of power consumptions [6][7][8]

	ESP8266	**CC3200**	**STM32F407**
Tx	∼145 mA	∼215 mA	-
Rx	56 mA	59 mA	-
Modem idle/sleep	15 mA	15.3 mA	36 mA
Deep-Sleep	10 µA	4 µA	1.7 µA

The ESP8266 typically draws less current while sending than the CC3200 with ∼145 mA and ∼215 mA respectively. However, the transmission current heavily depends on the WLAN

[2] http://www.ti.com/product/CC3200

protocol and the transmission power, which can be configured. The receiving current difference is mimimal with typically 56 mA for the ESP8266 and 59 mA for the CC3200. When the wireless module is put to sleep both wireless SOCs also draw almost the same current with 15 mA and 15.3 mA respectively. This is only half the power consumed by the STM32F407. In the deepest sleep mode all devices use $\leq$10 µA.

Development Board

To avoid the need of pricey custom printed circuid boards for the smart battery cell research platform, a range of publicly available development boards for the ESP8266 and the CC3200 platform are investigated.

Since the ESP8266 is more common on the privat market there are great amounts of different development boards available. Basic ESP8266 modules comprised of the ESP8266 chip, external flash and an antenna are available for less than $5.[3] However, these simple modules usually can't be used with breadboards and don't contain convenient pin-headers for flashing or a reset button. The named features are added by breakout boards like the *Adafruit HUZZAH ESP8266 Breakout*, which extends the basic ESP8266 ESP-12 module with buttons, LEDs, 5 V compatible UART interface, 3.3 V voltage regulator and breadboard compatible pin breakouts for less than $10.[4] The *SparkFun ESP8266 Thing* additionally comes with an on-board Li-Po charging circuit and a socket for SparkFun's prismatic Li-Po batteries for $15.95.[5] When using the *SparkFun ESP8266 Thing* only the passive cell balancing circuit and a voltage divider for cell voltage measurement with the on-board ADC need to be added externally.

The development boards offered for the CC3200 platform are more focused on industrial evaluation and development. The *SimpleLink Wi-Fi CC3200 LaunchPad* available for $29.99 comprises various additional hardware.[6] This, for example, includes a FTDI USB driver for programming and debugging. Other boards like the *RedBearLab WiFi Micro CC3200 Development Board* are targeted to private users. It has an Arduino form factor. However, this board also costs around twice as much as the most expensive ESP8266 board listed previously.[7]

Development Environment

Both ESP8266 and CC3200 can be programmed using the Arduino environment. This enables a flat learning curve and knowledge transfer from previous Arduino experience.

The ESP8266 Arduino core can be added to a standard Arduino installation as a third-party platform package.[8] For the CC3200 a whole fork of Arduino, named *Energia*, has to be installed.[9]

By using the common Arduino API the written software is independent from the hardware platform. This increases the portability of the software. In case of future hardware requirements changes this allows to easily switch to another platform.

[3] https://www.amazon.com/, effective 2016-10-15.
[4] https://www.adafruit.com/products/2471, effective 2016-10-15.
[5] https://www.sparkfun.com/products/13231, effective 2016-10-15.
[6] http://www.ti.com/tool/cc3200-launchxl, effective 2016-10-15.
[7] http://www.makershed.com/products/wifi-cc3200-board, effective 2016-10-15.
[8] https://github.com/esp8266/Arduino
[9] https://github.com/energia/Energia

Platform Choice

The CC3200 surpasses the ESP8266 in regards of available RAM and hardware peripherals. The ESP8266 scores with a lower transmission current, better suited development boards and a significantly lower price tag. The inferior hardware peripherals of the ESP8266 still meet the requirements of the smart battery cell research platform. This is why the ESP8266 in combination with *SparkFun ESP8266 Thing* board are chosen as basis for the smart battery cell research platform. Figure 2.1 shows the *SparkFun ESP8266 Thing*.

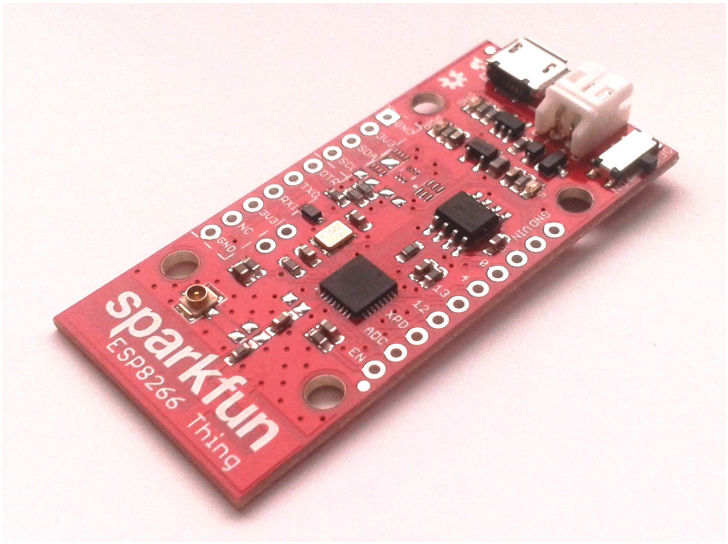

Figure 2.1: SparkFun ESP8266 Thing

Chapter 3

Additional Hardware

The ESP8266 development board needs to be extended with additional hardware to form a smart battery cell. The needed components are outlined in the following sections.

Smart Battery Cell

Li-Po Cell

The *SparkFun ESP8266 Thing* has a JST connector for Li-Po batteries. Prismatic Li-Po cells with a JST connector can also be sourced from *SparkFun*. The capacities of the offered battery cells range from 40 to 6000 mAh. A Li-Po battery with a capacity of 2000 mAh is chosen to be able to power the CMU several hours without recharging. This capacity also comes closest to 18650 cylindrical cells which typically provide around 2500 mAh. Furthermore, the battery has a built-in protection circuit to prevent damage to the cell if used improperly. Figure 3.1 shows the used Li-Po cell.

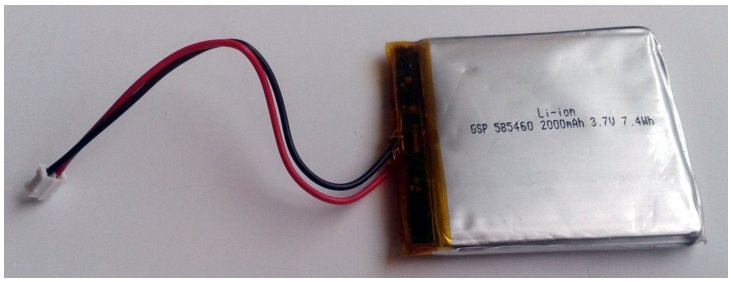

Figure 3.1: Prismatic 2000 mAh Li-Po cell

Battery Breakout

The *SparkFun ESP8266 Thing* board doesn't provide a breakout pin for V_{BAT}. However, there is a *not connected* (NC) pin in the FTDI pin header (`FTDI_BASICPTH`) [9]. A jumper wire from the JST battery connector (`JP2`) to the FTDI pin header maps V_{BAT} to the NC breakout pin. Figures 3.2 and 3.3 show the changed headers schematic and the *SparkFun ESP8266 Thing* with added jumper wire respectively.

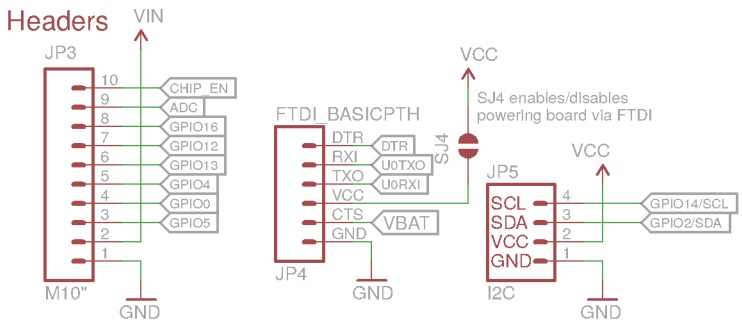

Figure 3.2: Headers section of the *SparkFun ESP8266 Thing* schematic with added V_{BAT} breakout.

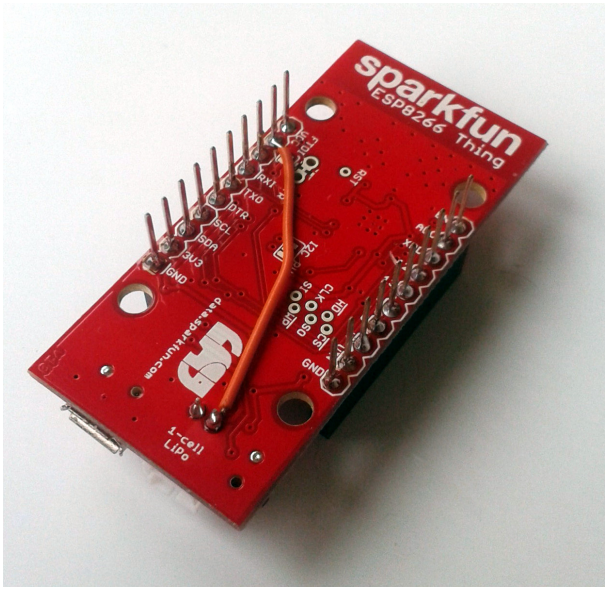

Figure 3.3: *SparkFun ESP8266 Thing* with jumper wire to breakout V_{BAT}.

Voltage Measurement

The voltage of the Li-Po battery cannot directly be measured with the ESP8266's ADC. To map the maximum voltage of the battery (4.305 V) to the maximum ADC voltage (1 V) a voltage divider is used. Figure 3.4 shows the schematic of the voltage divider.

Equation 3.2 shows the voltage ratio with $R1 = 100\,k\Omega$ and $R2 = 22\,k\Omega$.

$$U_{BAT} = \frac{R1 + R2}{R2} \cdot U_{ADC} \tag{3.1}$$

$$U_{ADC} = \frac{11}{61} \cdot U_{BAT} \tag{3.2}$$

To prevent the discharge of the battery through the resistors while the ESP8226 is switched of, the voltage divider is not wired to ground but `GPIO4`. This pin can be configured to output *low* (ground) while powered on. If the ESP8266 is powered off, `GPIO4` has a high impedance and the cell is not discharged.

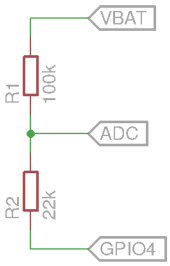

Figure 3.4: Voltage divider schematic.

Passive Balancing

If cells are connected in series the charging or discharging must be stopped when any of the cells reaches its upper or lower voltage range limits. This requires the cells to be balanced externally. An easy to implement balancing method is *passive balancing*. It uses switchable resistors to bypass cells which already reached their maximum voltage during charging [10].

Passive balancing is realized using a 4 Ω, 10 W power resistor and a FQP30N06L n-channel MOSFET. The resistor discharges the cell with 0.925 A at 3.7 V. The MOSFET gate is connected to `GPIO5` which also controls a LED on the development board [9]. Thus, the state of the passive balancing can easily be observed. A pull-down resistor ensures $V_{GS} < V_{GS,off}$ while the ESP8266 is switched off.

Figure 3.5 shows the schematic of the passive balancing circuit.

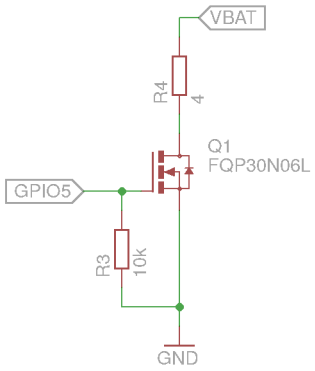

Figure 3.5: Passive balancing circuit schematic.

Shield

All additional hardware mentioned so far can be connected to the development board using a prototyping breadboard. For increased convenience when working with multiple smart battery cells, the hardware is combined to a smart battery cell extension shield for the *SparkFun ESP8266 Thing*. For easy creation and modification a perfboard is used.

Besides the voltage divider and passive balancing circuit the shield also comprises a two-rowed pin header to plug the development board into as well as a terminal for V_{BAT} and ground. The terminal circuit can be interrupted with a jumper. This can be useful if multiple smart battery cells are connected in series while e.g. connected to the same PC for programming. Without disconnecting the series connection this would cause a hazardous potential difference between the ground potentials connected to the PC.

Figures 3.6 and 3.7 show the whole shield schematic and a picture of the assembled shield respectively.

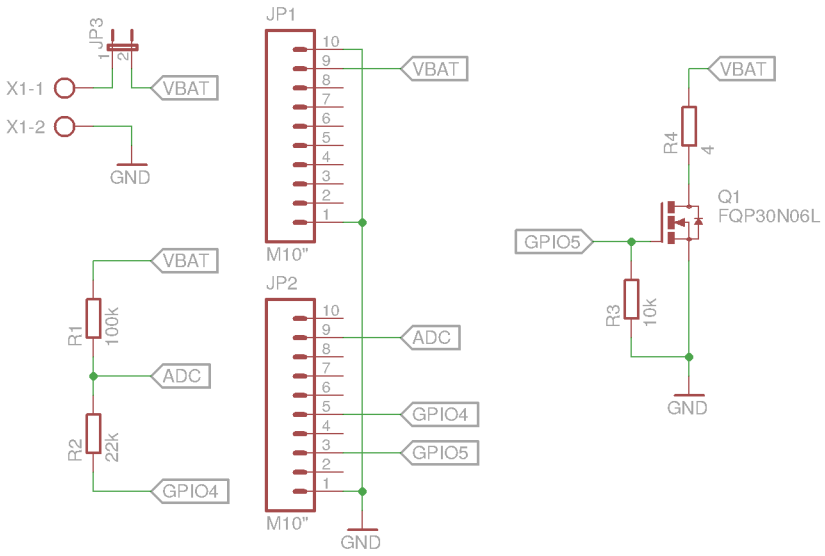

Figure 3.6: Smart battery cell shield schematic.

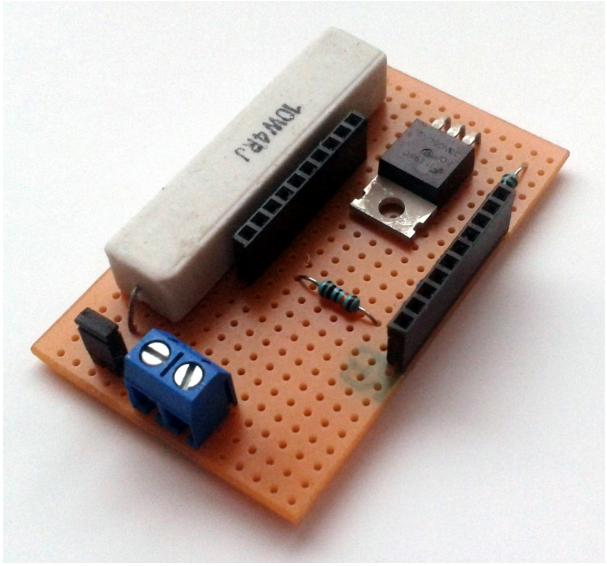

Figure 3.7: Assembled smart battery cell shield.

FTDI Adapter

To upgrade the ESP8266's firmware it has to be downloaded to the flash storage via UART. A convenient way to do this is by using a FTDI *USB to serial* integrated circuit (IC). This interface can also be used to communicate with the device using a serial terminal.

Since the operating voltage of the ESP8266 is 3.3 V the adapter should operate at the same voltage.

The bootloader for downloading the software is only active for a short time after a reset and then starts the actual firmware. Thus, the device needs to be reset before downloading the

firmware. With the FTDI header's *Data Terminal Ready* (DTR) pin beeing wired to the reset pin of the ESP8266, the reset can be automated if the adapter provides a DTR line [9].

SparkFun's *SparkFun FTDI Basic Breakout - 3.3V*[1] provides the required properties and is therefore used to upgrade the smart battery cell's firmware.

As described in 3.1.2, V_{BAT} is mapped to the NC pin of the FTDI pin header. This collides with the *Clear To Send* (CTS) pin of the FTDI adapter which is not required for this application. To prevent damages to the FTDI IC, this pin must not be connected to the modified pin header. Figure 3.8 shows a picture of the *SparkFun FTDI Basic Breakout - 3.3V* with omitted CTS and 3V3 power pin.

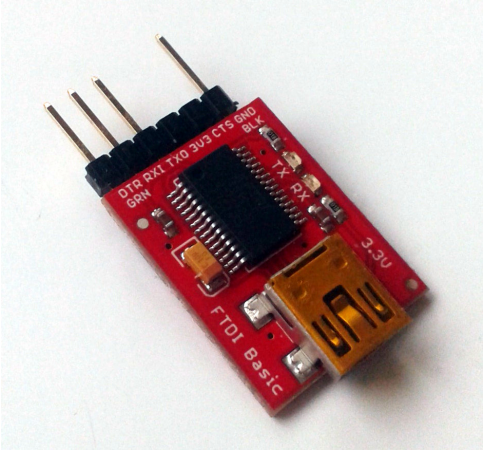

Figure 3.8: *SparkFun FTDI Basic Breakout - 3.3V* with omitted CTS and 3V3 power pins.

Consumer

To have a simple consumer for up to five smart battery cells connected in series a 24 V, 5 W light bulb is used. It is mounted on a perfboard together with an on-off switch and a terminal. Figure 3.9 shows the consumer circuit board.

Gateway

The ESP8266 can use an existing wireless network (station mode) or provide a network itself (access point mode). This allows to operate the smart battery cells in a mesh network. Among other things, this requires routing strategies between the single nodes, which adds complexity to the distributed communication of the battery pack. For this reason, a single wireless access point is used for all cells. This access point also acts as gateway between the smart battery cells and the outside world.

A *Raspberry Pi 3 Model B*[2] single-board computer is used as the gateway. Using the *Raspbian GNU/Linux* operating system it can be configured to act as wireless router with Dynamic Host Configuration Protocol (DHCP). Furthermore, it can run a gateway software that acts as gateway between the wireless cell network and a network connected via the ethernet interface.

Figure 3.10 schematically depicts four smart battery cells connected in series and forming a battery pack. The common gateway provides the WiFi network for the CMU communication and acts as relay between the battery pack and some sort of web service.

[1] https://www.sparkfun.com/products/9873
[2] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

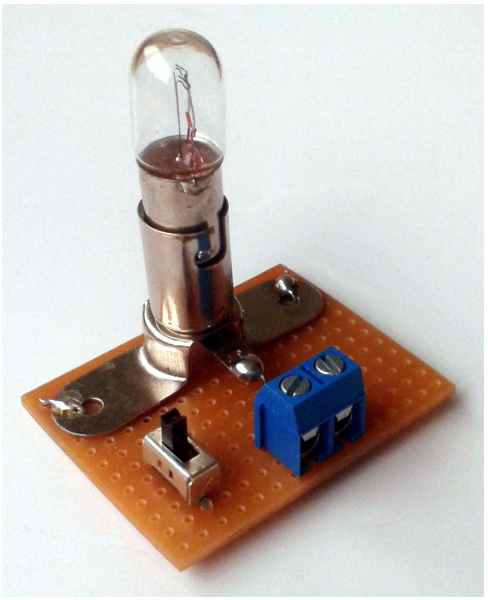

Figure 3.9: Consumer circuit board.

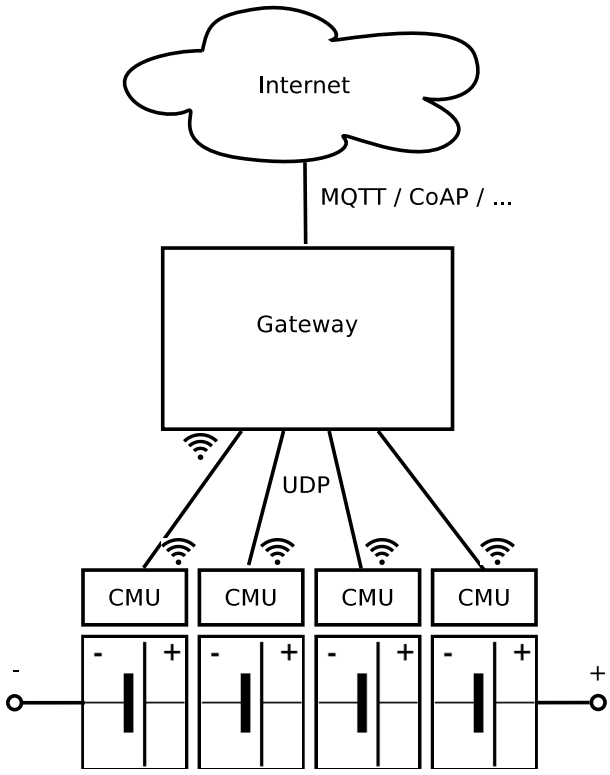

Figure 3.10: Deployment diagram of a battery pack comprising four smart battery cells connected in series and a common gateway.

Chapter 4

Software

Cell Management Unit

The CMU software uses the Arduino application programming interface (API) and is thus roughly structured in the `setup()` and the `loop()` part.

During the setup the serial interface and general-purpose inputs/outputs (GPIO) are initialized. Furthermore, the CMU connects to the gateway network with preconfigured service set identifier (SSID) and passphrase. After establishing a connection to the network a `WifiUDP` socket is initialized and bound to port 4444.

During the loop phase the CMU periodically broadcasts its state to the network. The state comprises the current cell voltage in mV, balancing state (on = 1 / off = 0), and balancing mode (manual = 0 / automatic = 1). The state message has a size of five bytes. Table 4.1 illustrates the message structure.

Table 4.1: Cell state message

Byte	**1**	**2**	**3**	**4**	**5**
	Message ID = 0	Voltage [mV]		Balancing state	Balancing mode

All CMUs continously check for incoming messages and handle them. If the state message of another CMU is received, this smart battery cell's state is cached in order to calculate the battery packs average voltage. Since no topology information is available, the single identifier of a CMU is its IP address assigned by the gateway. In a class C network only the last octet of the address varies and can thus be used as unique ID as shown in the following example.

$$192 \,.\, 168 \,.\, 0 \,.\, \underbrace{42}_{\substack{CMU \\ ID}}$$

This ID can also be used as array index to store the state information of all cells in the network. Every CMU's state in the array is marked with a millisecond timestamp of the last update. If an entry is older then two seconds, this entry is ignored when calculating aggregated values. This allows to physically reconfigure the battery pack by adding or removing smart battery cells.

The number of cells in a network is bounded by the CMU's RAM size to hold the state array. Furthermore, in a class C network the number of assignable addresses is limited to 254, which should be sufficient in this research context. To overcome memory limits in large battery packs,

CMUs could be clustered in multicast groups and just share aggregated information between these groups.

To control the passive balancing of the smart battery cells another message type is defined. The balancing control message has a size of three bytes. The structure of the message is shown in table 4.2.

Table 4.2: Balancing control message

Byte	1	2	3
	Message ID = 1	Balancing state	Balancing mode

This message is intended to be sent from the gateway to change the smart battery cell's balancing behaviour. If the balancing mode is set to *manual* the balancing circuit can be turned on and off by setting the balancing state. On the other hand, if the balancing mode is set to *automatic* the cell continously compares its cell voltage to the aggregated battery pack voltage. If the own voltage is higher than the average, the passive balancing is automatically activated. Thus, the balancing state parameter has no effect.

Gateway

The gateway software is written in Java using Apache Ant to automate the build process and Apache Ivy to resolve dependencies. The software is structured in four main parts consisting of configuration, cell communication, web communication and the battery pack domain model.

Figure 4.1 shows the main classes for web and cells communication as well as the battery pack domain model.

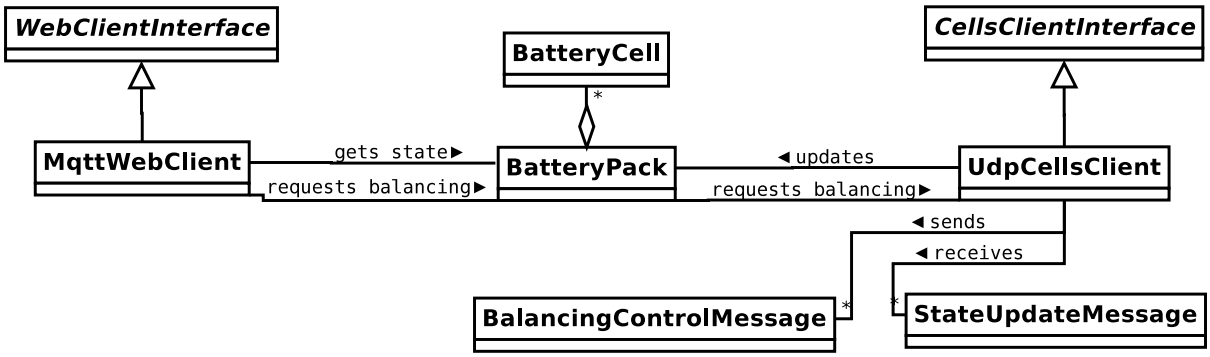

Figure 4.1: Gateway classes for web and cells communication and battery pack domain model.

The cells client listens for broadcasted CMU state messages on the network and updates the model of the battery pack accordingly. The cells client also forwards change requests of the balancing state and/or mode to the individual CMUs.

The web client gets the state of the battery pack and its single cells from the model. This information can then made accessible via an arbitrary protocol. As a proof of concept a MQTT client is implemented. It periodically publishes every single cell state as well as the minimum, maxmimum and average battery pack voltages. It also subscribes to a balancing topic for every smart battery cell and forwards these requests to the model.

In order to enable convenient changess of configuration parameters a YAML file is used. It is loaded at startup by the `ConfigLoader` which subsequently creates the `Config` object (see figure 4.2).

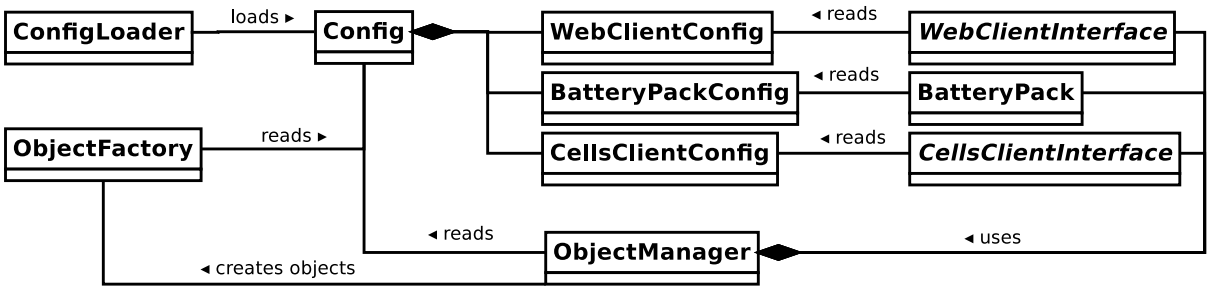

Figure 4.2: Configuration and object management classes.

The configuration is then read by the `ObjectManager` which uses the `ObjectFactory` to instantiate the objects depicted in figure 4.1. Individual configuration parameters like e.g. the cell communication port are accessed by the according objects themselves.

Appendix B describes the single steps to setup the Raspberry Pi 3 to provide the WiFi network and run the gateway software.

User Interface

The most basic user interface can be a class that implements the `WebClientInterface` and periodically outputs the battery pack state to the Java console.

As mentioned in section 4.2 a MQTT client was implemented to publish the cell state to a MQTT broker. A simple web based user interface is used to display this information. The *Eclipse Paho JavaScript Client*[1] MQTT library allows to connect to a MQTT broker via WebSockets. The received cell and aggregated pack voltages are then displayed using the *Highcharts*[2] charting library. The balancing mode can be controlled via buttons that trigger the publishing of a control message to the MQTT broker.

Figure 4.3 shows a screenshot of the user interface showing four battery cells.

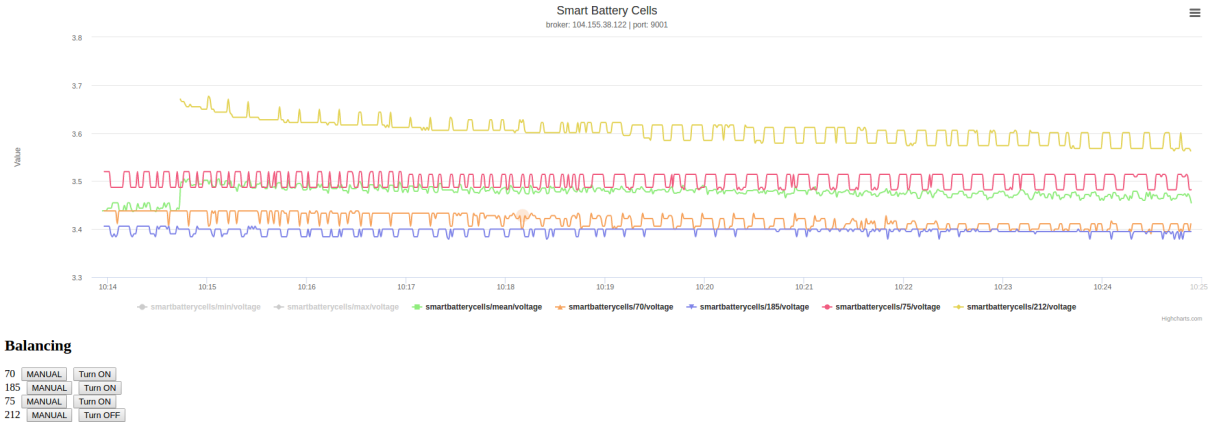

Figure 4.3: Screenshot of the web based user interface.

The cell with ID 212 (yellow) was added to the battery pack during runtime. This caused the average battery pack voltage (green) to rise. The passive balancing of the newly added smart battery cell was manually activated which causes a faster voltage drop over time.

[1] https://eclipse.org/paho/clients/js/
[2] http://www.highcharts.com/products/highcharts

Appendix C describes the setup of the cloud server consisting of the open source MQTT broker *Mosquitto*[3], the actual web interface and a *Nginx*[4] Hypertext Transfer Protocol (HTTP) server to provide it.

[3] https://mosquitto.org/
[4] https://nginx.org/

Chapter 5

Conclusion

This project introduced a Arduino-based smart battery cell research platform. The platform comprises an ESP8266 WiFi SOC on the inexpensive *Sparkfun ESP8266 Thing* development board. Additional hardware for cell voltage measurement and passive balancing is combined on a simple smart battery cell shield. The CMUs communicate via UDP/IP on a WiFi network created by a Raspberry Pi 3 Model B acting as gateway. The gateway runs a software to relay cell state information and balancing control messages between the cell network and a web service. As a proof-of-concept the communication between the battery pack and a graphical web interface was implemented via the MQTT protocol.

The smart battery cell research platform software currently only supports basic features which are to be researched and extended in the future. Furthermore, the ADC showed significant jitter which has to be reduced by additional hardware and/or filtering software.

Figure 5.1 shows four smart battery cells connected in series powering a light bulb.

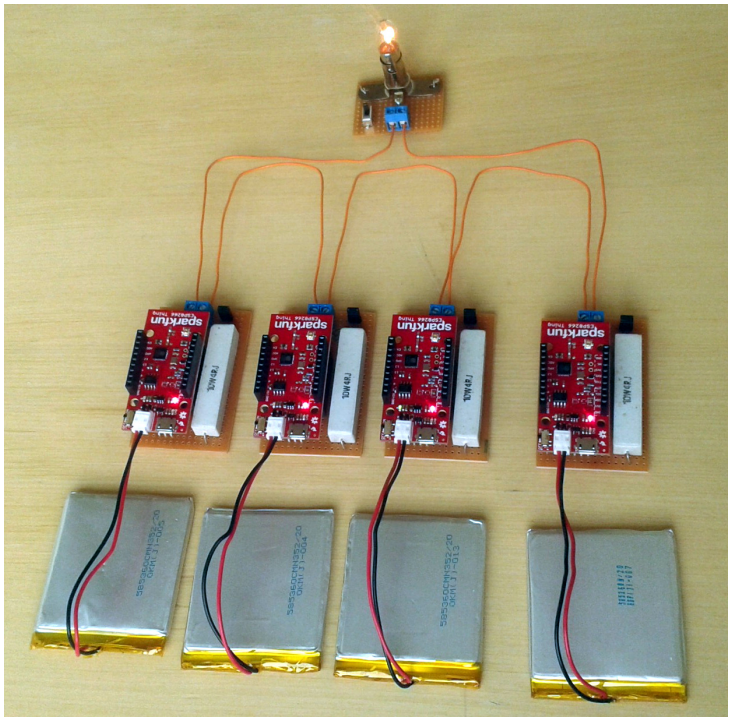

Figure 5.1: Four smart battery cells connected in series powering a light bulb.

Bibliography

[1] S. Steinhorst, M. Lukasiewycz, S. Narayanaswamy, M. Kauer, and S. Chakraborty, "Smart cells for embedded battery management," in *Proceedings of the 2nd International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA 2014)*, pp. 59–64, 9 2014.

[2] S. Steinhorst and M. Lukasiewycz, "Topology identification for smart cells in modular batteries," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2015)*, pp. 1–4, 3 2015.

[3] N. Martiny, T. Mühlbauer, S. Steinhorst, M. Lukasiewycz, and A. Jossen, "Digital data transmission system with capacitive coupling for in-situ temperature sensing in lithium ion cells," *Journal of Energy Storage*, vol. 4, pp. 128–134, 2015.

[4] V. Lorentz, "Electrical energy storage: Distributed battery monitoring." EES 2012 Electrical Energy Storage Exhibition Munich, Nov. 2012. http://media.nmm.de/91/lorentz_fraunhofer-iisb_14.11.2012_10.30_26767091.pdf.

[5] M. Schneider, S. Ilgin, N. Jegenhorst, R. Kube, S. Püttjer, K. R. Riemschneider, and J. Vollmer, "Automotive battery monitoring by wireless cell sensors," in *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, pp. 816–820, May 2012.

[6] Espressif, "ESP8266EX Datasheet 2016 v4.9." https://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf, 2016. Accessed: 2016-08-30.

[7] Texas Instruments, "CC3200 SimpleLink™ Wi-Fi® and Internet-of-Things Solution, a Single-Chip Wireless MCU." http://www.ti.com/lit/gpn/cc3200, Feb. 2015. Accessed: 2016-08-30.

[8] STMicroelectronics, "STM32F407/417." http://www.st.com/en/microcontrollers/stm32f407-417.html. Accessed: 2016-08-30.

[9] Jim Lindblom, "SparkFun_ESP8266_Thing." Schematic, https://cdn.sparkfun.com/datasheets/Wireless/WiFi/SparkFun_ESP8266_Thing.pdf, May 2015. Accessed: 2016-10-16.

[10] M. Isaacson, R. Hollandsworth, P. Giampaoli, F. Linkowsky, A. Salim, and V. Teofilo, "Advanced lithium ion battery charger," in *Battery Conference on Applications and Advances, 2000. The Fifteenth Annual*, pp. 193–198, IEEE, 2000.

Appendix A

Zip Archive

Folder	**Description**
cmu	CMU software
gateway	Gateway software
www	MQTT web interface
eagle	CadSoft EAGLE design files
datasheets	Datasheets of the hardware components

Appendix B

Setup Gateway

The following setup commands refer to a Raspberry Pi 3 Model B running a fresh install of Raspbian GNU/Linux 8 (jessie).

Update Raspbian

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo rpi-update
```

Configure WiFi Access Point

Install the *hostapd* software access point daemon and the *dnsmasq* network infrastructure packages with the following command.

```
$ sudo apt-get install hostapd dnsmasq
```

Stop the DHCP client *dhcpcd* from configuring `wlan0` by adding the following line to the end of the configuration file `/etc/dhcpcd.conf`.

```
denyinterfaces wlan0
```

Edit the `wlan0` part of the inteface configuration file `/etc/network/interfaces` so that it looks like the following in order to configure a static class C network.

```
allow-hotplug wlan0
iface wlan0 inet static
    address 192.168.0.1
    netmask 255.255.255.0
    network 192.168.0.0
    broadcast 192.168.0.255
```

Open the *hostapd* configuration with `/etc/hostapd/hostapd.conf` and change its contents to the following in order to configure the WiFi access point.

```
# This is the name of the WiFi interface we configured above
interface=wlan0
```

```
# Use the nl80211 driver with the brcmfmac driver
driver=nl80211

# This is the name of the network
ssid=SmartCellGateway

# Use the 2.4GHz band
hw_mode=g

# Use channel 6
channel=6

# Enable 802.11n
ieee80211n=1

# Enable WMM
wmm_enabled=1

# Enable 40MHz channels with 20ns guard interval
ht_capab=[HT40][SHORT-GI-20][DSSS_CCK-40]

# Accept all MAC addresses
macaddr_acl=0

# Use WPA authentication
auth_algs=1

# Require clients to know the network name
ignore_broadcast_ssid=0

# Use WPA2
wpa=2

# Use a pre-shared key
wpa_key_mgmt=WPA-PSK

# The network passphrase
wpa_passphrase=allcellsarebelongtome

# Use AES, instead of TKIP
rsn_pairwise=CCMP
```

For *hostapd* to load this configuration on startup open the file `/etc/default/hostapd` and replace the line

```
#DAEMON_CONF=""
```

with the following.

```
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

Configure *dnsmasq* by replacing the contents of `/etc/dnsmasq.conf` with the following.

```
interface=wlan0
listen-address=192.168.0.1
bind-interfaces
domain-needed
bogus-priv
dhcp-range=192.168.0.2,192.168.0.254,24h
```

Reboot the Raspberry Pi.

You should now be able to connect to the WiFi with SSID *SmartCellGateway* using the pre-shared key '`allcellsarebelongtome`' as configured above.

Install Apache Ant and Ivy

Install Apache Ant and Ivy by executing the following commands

```
$ sudo apt-get install ant ivy
$ sudo ln -s /usr/share/java/ivy.jar /usr/share/ant/lib
```

Run Gateway Daemon

Copy the gateway daemon sources from appendix A to the Raspberry Pi and run the daemon with the following command.

```
$ ant daemon
```

Appendix C

Setup Cloud Server

Configure Firewall

Add the following firewall rules to your system.

Protocol	Port	Description
TCP	80	HTTP
TCP	1883	MQTT
TCP	9001	Websocket MQTT

Install Docker Engine

Follow the instructions on https://docs.docker.com/engine/installation/linux/ubuntulinux/ to install the latest Docker Engine needed to run docker images.

Install Docker Compose

Execute the following commands in a terminal to install the Docker Compose version 1.8.1.

```
$ curl -L https://github.com/docker/compose/releases/download/1.8.1/docker-
    compose-‘uname -s‘-‘uname -m‘ > /tmp/docker-compose
$ sudo mv /tmp/docker-compose /usr/bin
$ sudo chmod 755 /usr/bin/docker-compose
```

Create Docker Compose File

Create a docker compose file named `mosquitto+nginx.yml` with the following content.

```
version: ’2’
services:
  mqtt:
    container_name: "mqtt"
    image: toke/mosquitto
    ports:
      - 1883:1883
      - 9001:9001
```

```
  web:
    container_name: "web"
    image: kyma/docker-nginx
    ports:
      - 80:80
    volumes:
      - ~/www:/var/www
```

It runs two docker containers. The first container, named `mqtt`, starts the open source MQTT broker *Mosquitto* listening on port 1883 for MQTT and port 9001 for Websocket MQTT client connections. The second container, named `web`, starts the open source static HTTP web server *Nginx* listening on port 80 and providing the contents of the directory `~/www`.

Create Web Content

Create at least a `index.html` file in `~/www`. You can, for example, use the *Eclipse Paho JavaScript Client*[1] library to interface the MQTT broker on the cloud server. Appendix A contains an `index.html` which displays the cell voltages in a graph and allows to control the cell balancing via MQTT.

Start MQTT Broker and HTTP Server

Execute the following command to start the docker containers with Docker Compose:

```
$ docker-compose -f mosquitto+nginx.yml up -d
```

[1] https://eclipse.org/paho/clients/js/