Delft University of Technology

Bachelor Graduation Project Electrical Engineering
EE3L11

Simulating the Energy Transition

Raspberry Pi Hardware Hub

Authors:
J R de Waal (5068843)
N Rauf (4662636)

May - July 2023

TU Delft Delft University of Technology

Abstract

Before the project was started, there was an energy system integrator demonstrator[1]. This demonstrator was purely software, there was no physical interaction possible with it. It demonstrated the energy system by simulations. The objective of this project was to design and built peripherals that represents the components of an energy system. During the project, four peripherals were made that represents a solar panel, a wind turbine, a load and a storage indicator. A hub is placed on top of the Raspberry Pi, all the peripherals are connected to the Raspberry Pi via this hub. The peripherals consisted of two parts a representation of the energy technology and a microcontroller to do the necessary signal processing. For the communication SPI protcol was used.

Keywords: Atmega8, Energy system integrator demonstrator, Raspberry Pi, Arduino, Hardware Hub

Preface

The main achievement of this project is to provide peripherals for an energy transition demonstrator. These peripherals should be a representation of sustainable energy technologies, a load representation and a storage representation, that gives insight and a better understanding on how sustainable energy is generated and consumed. There were four peripherals made that represents: a solar panel, wind turbine, storage and a load.

We would like to thank our supervisor dr. Milos Cvetcovic for his support and feedback during the project. Also, we would like to thank ing. AMJ Slats and ing. M. Schumacher for their support when using the Tellegen Hall. Last but not least dr. ir. Ioan Lager for organizing and managing the Bachelor Graduation Project

Contents

1 Introduction

1.1 state-of-the-art analysis

As the electricity supply is transitioning to renewable generation, we are faced with new challenges[2]. At the moment, lots of work is being done by grid operators to balance supply with demand[3], something that becomes increasingly complex due to the intermittent generation from renewable. This process is usually not noticed by the general public. One example when the public experienced the consequences from an imbalance in the system was in 2018, when an unbalance caused many clocks to run six minutes behind[4].

Current hardware available to educate about renewable energy[5], [6] focus on the generation and consumption, but most neglect the time varying behavior. Some digital demos [7], [8] do focus on the balance between generation and consummation over time. While multiple tools are used by grid operators to predict loads and generation, no hardware kit that demonstrates the challenges caused by the time varying behavior of the power grid is available to the public.

This project builds upon previous projects[1] that try to generate a simulation of the time varying behavior of power flows. It also uses existing communication standards[9].

1.2 Problem Definition

At the moment there exist an Energy System Integration Demonstrator that does the demonstration in software. What should be done is building hardware components that creates a more insightful, interactive and a better way of understanding how sustainable energy system functions together in terms of energy generation, energy consumption and energy storage. The objective is to design and make small scale representations of two sustainable energy technologies, a storage indicator and a load representation. In the future it is desired to provide additional sustainable energy technologies since other countries could then use it too. For example, countries with mountainous regions depend on hydropower. If the civilians of that country want to make use of this product, it is desired from their point of view that a representation of hydropower is included in it. Furthermore, to give the peripherals an aesthetic look, 3D model structures for the peripherals should be provided. For this project it was not thought about to interconnect the hardware, perhaps that would be an exciting challenge for the future. At the end of the project, the four peripherals were made, however, the 3D models were not managed to be printed.

A constraint to take into account is that the Raspberry PI has a voltage limit of 3.3V. At anytime this voltage limit should not be passed.

1.3 System Overview

The setup for running the demonstration consists of three parts. First there is the Raspberry Pi. This is a small single board computer running Linux. The Raspberry Pi component acts as the foundation of the system on which other components are, some literally, build on top of. It has an extra connector, the GPIO (General Purpose Input Output) port. This port has no predefined function[1] and can be configured in software[10]. Via this port the hardware can communicate with the simulation software running on the Raspberry Pi.
The next component is the hardware hub. The hardware hub is mounted on top of the Raspberry Pi and forms the bridge between the Raspberry Pi and the peripherals. It has one connector attached to the GPIO port and number of connectors where peripherals can be attached. It also has some features that will be explained in chapter 4.

[1] Some pins are reserved for dedicated functions

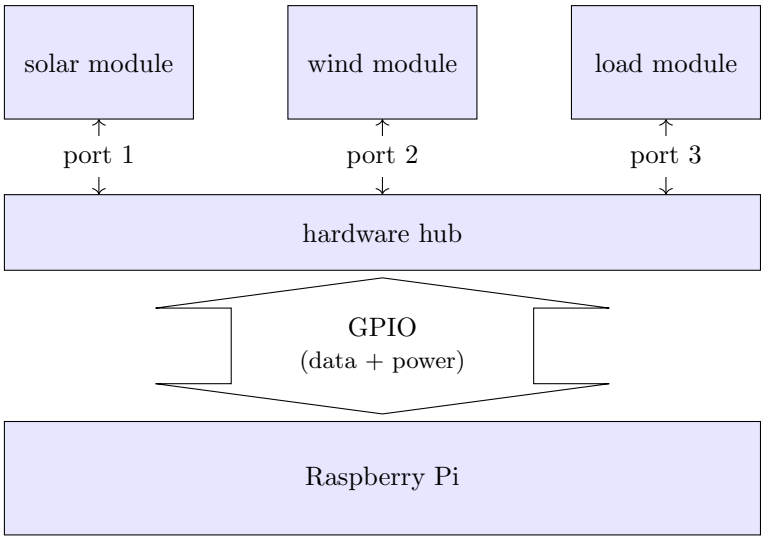


Figure 1.1: System overview example

The last part are the peripherals. Up to six peripherals can be connected to the hardware hub. The peripherals are devices that represent generation, storage or consumption. The connected peripherals appear in the simulation. By connecting a combination of peripherals, changes are made in the simulation. Peripherals may also function as sensors to read values as wind speed or solar intensity. Details of the different peripherals are given in chapter 5.

Figure 1.1 shows how all components come together.

1.4 Outline

The report starts with stating the program of requirements in chapter 2. After that, the different options of communication protocols are discussed in chapter 3. Then, in chapter 4, the requirements of the hardware hub are stated and a profound explanation on the design choices are made, the same is done for the peripherals in chapter 5. Finally, the design is being tested and implemented, this is narrated in chapter 6. To mark the end of the report a conclusion is provided.

2 Program of Requirements

The requirements of the project can be divided into a few subsection. Key words in *ITALIC* in this section are used to indicate requirement levels, following RFC 2119[11].

2.1 Functional Requirements

- The peripherals *MUST* represent either a solar, wind, storage or load.
- The peripherals of solar and wind *MUST* give an output, that is feed into the software, when an input is giving to the peripheral.

2.2 Ecological Embedding in the Environment

- Voltage *MUST* be in the safe desired range
- The Hardware hub *SHOULD* have some form of overcurrent protection
- The Hardware hub *SHOULD* protect the raspberry pi against damage
- *SHOULD NOT* cause damage when connected to itself on the peripheral side

2.3 System Requirements

- The software interface on the Raspberry Pi *MUST* be compatible with Python
- The hardware hub *MUST* provide power to any connected peripherals
- The hardware hub *MUST* allow communication between Raspberry Pi and peripherals
- The hardware hub *MAY* automatically detect when a peripheral is connected and unconnected.
- Hardware hub *MAY* detect the type of peripheral

3 Communicating

This chapter describes how the peripherals communicate with the Raspberry Pi.

3.1 Communication Options

The assignment encouraged the use of an existing communication protocol. This section lists some communication standards and their advantages and disadvantages.

- **USB**
  Universal serial bus, best known by its abbreviation USB is a standard that is used to exchange data and power between a host and peripherals[12]. It has the advantage that it is already implemented in most modern computers, removing the dependency on the Raspberry Pi. With this protocol being relatively complicated, it is difficult to develop new hardware. Because of the time constraint of the project, this option was dropped.

- **Serial**
  Asynchronous serial communication, often referred to as serial, contains a range of protocols where two sides can send data without having a line that carries a clock signal[13], [14]. Without flow control, both sides can start sending data at any time. It needs at least two wires for communication, one for both directions. The limitation is that this protocol can only be used for a direct link between two devices. Each link uses a dedicated hardware component on both sides, the UART[1] (universal asynchronous receiver-transmitter). This is no problem for the peripherals; because only a single link is needed and most microcontrollers have an UART available. The problem is that the Raspberry Pi only has a single UART available on the GPIO port[15]. This can be solved by using multiplexers to alternate communication with multiple devices, but any data send while not listening or listening to another device will be lost.

- **I^2C**
  Inter-Integrated Circuit or I^2C is a bus communication protocol invented by Philips[16]. It can be used to establish communication between a great number of host using only two lines. The Raspberry Pi has built in hardware support for this protocol[15], [17]. This seems a reasonable option. However, this protocol requires all nodes to have a unique address configured. This makes it hard to support dynamically plugging in new hardware, as this would require dynamic allocation of addresses.

- **SPI**
  Serial Peripheral Interface (SPI) uses a communication bus consisting of two data lines and a clock line[18], [19]. At least one extra line is necessary to activate the peripheral for communication, creating a total of 4 signal lines that have to be connected to each peripheral. The Raspberry Pi has two hardware interfaces for this bus available on the GPIO port[15], [17]. This was seen as most suitable for this project. Details will be discussed in the remainder of this chapter.

3.2 SPI Bus

The data link with the peripherals is a SPI (Serial Peripheral Interface) bus. This bus has three signals connected to all peripherals in parallel. One other signal is dedicated to select the peripheral that is active in the communication. The number of signals connected to the master (the Raspberry Pi) is at least equal to the number of peripherals plus three. Figure 3.1 shows how the lines are connected and the signal direction. This can be extended to any number of peripherals.

[1] UART is sometimes used as term to refer to any protocol using asynchronous serial communication[13]

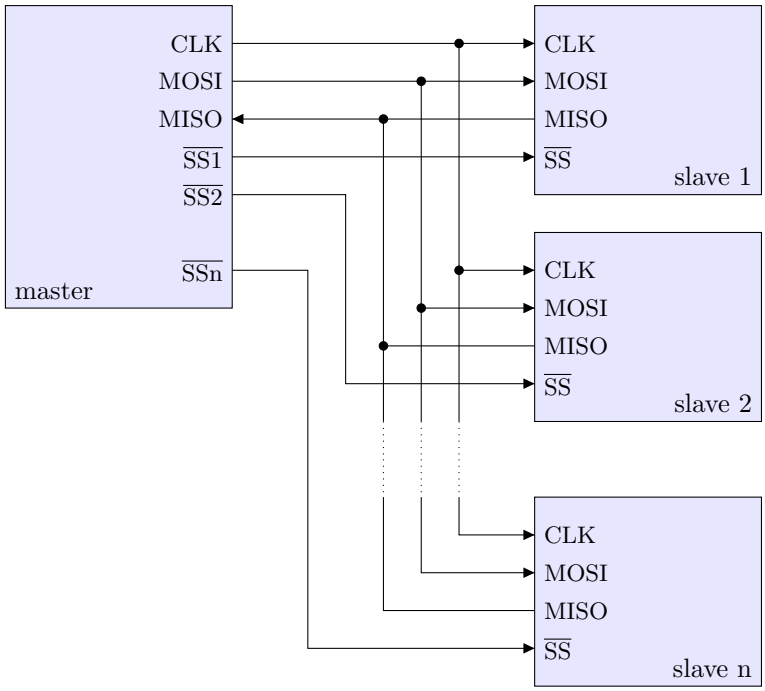


Figure 3.1: Diagram showing the different signals and signal directions in a SPI bus system

The three shared signals are SCLK,[2] MOSI and MISO. SCLK (Serial clock) is the clock signal that determines the speed of the data transmissions. The MOSI (Master Out Slave In) and MISO (Master In Slave Out) lines carry the data between the master and the peripheral. One bit of data can be transmitted in both directions per clock pulse. It has to be noted that multiple devices can send data on the MISO line. To prevent conflicts, the peripheral has to switch to high impedance mode when it is not signaled to be active in the communication.

One additional signal is needed to select the peripheral that is participating in the communication. $\overline{\text{SS}}$ (Slave Select)[3] is a active low line used to signal to the peripheral that it has to communicate. Because this signal is used to activate communication on a single peripheral, one line cannot be shared. Therefore, each peripheral needs its own select line. Other peripheral can observe the communication, but are supposed to ignore everything that happens while not selected.

3.2.1 SPI Modes

SPI has 4 different modes, labeled Mode 0 to Mode 3[18]–[20]. The difference between the modes is the clock signal. On most system this is configured using two settings, the CPOL bit and the CPHA bit. All possible configurations are shown in Table 3.1. By changing CPOL (clock polarity), high and low levels on the clock line are swapped. This also results in a falling edge becoming a rising edge, and vice versa. The CPHA (clock phase) bit sets trigger (rising or falling edge) for reading and sending data. Both triggers can be inverted by the CPOL setting.

Under normal circumstances, both sides need to be configured to use the same mode. This changes when the clock line gets inverted between two nodes. To correct for clock signal arriving inverted at the receiving node, both sides can be configured with a different setting for CPOL. Both inversions of the clock signal cancel each other out, ensuring that data is read on the correct moment.

3.3 Transmission Protocol

SPI only provides a way to send bytes in two directions. The content and meaning of these bytes still needs to be established. To do this, all devices are programmed to use the protocol explained below. This protocol was adapted for this project, but was inspired by existing protocols, for example the one used by the ENC28J60 network chip[21].

[2]Sometimes alternative names are used in (product) descriptions. Common alternative labels for SCLK are SCK, CLK and SCL. All refer to the same signal in the context of SPI

[3]This signal can also be labeled as CS (Chip Select) or CE (Chip Enable)

Table 3.1: SPI Modes with CPOL and CPHA. Adapted from [18]

SPI Mode	CPOL	CPHA	Clock in Idle State	Sample Data on	Shift the Data on
Mode 0	0	0	Logic low	rising edge	falling edge
Mode 1	0	1	Logic low	falling edge	rising edge
Mode 2	1	0	Logic high	rising edge	falling edge
Mode 3	1	1	Logic high	falling edge	rising edge

Table 3.2: Identification codes for the peripherals

peripheral type	ID (decimal)
solar module	101
wind module	102
storage module	105
load module	106

The communication protocol uses frames. The frame starts with the falling edge of the $\overline{\text{SS}}$ line. All communication is reset at the start of the frame. This serves as a way to mark the start of the first byte. The frame can contain multiple bytes. The frame ends at the rising edge of $\overline{\text{SS}}$. The interpretation of bytes depends on the position of the byte in the frame. Bytes are transmitted with the MSB[4] first.

Both sides can send data at the same time. The first byte send by the peripheral is used to indicate the type of peripheral, see Table 3.2 for types and their codes. This byte is left unused in other protocols[20]. Because no external input is needed for the peripheral to send this byte, it can be used at the start of the communication. This increases the efficiency by reducing the time needed to transfer one frame.

The first byte send from the hardware hub is used to indicate if the peripheral should store the next byte in its memory. Only the last (LSB[5]) bit of this byte is used, the other bits are discarded by the peripheral, but can be used for other functions in future versions. The timing diagram for this protocol is shown in Figure 3.2.

This protocol does not include error detection. If a peripheral is unplugged during communication, the Raspberry Pi may receive a wrong value. This has to be dealt with in software.

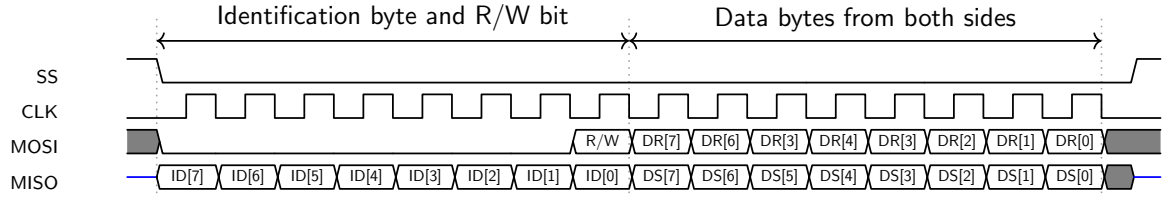


Figure 3.2: Template timing diagram for a frame containing data

3.3.1 Reading Data

The read operation can be used to retrieve data from the peripheral. This is used to read sensor values from peripheral that have one. If the last bit of the first byte is "0", this indicates a read operation. The second byte send by the peripheral contains the requested data. This can be the current sensor value or the value stored in memory. The peripheral must ignore the following bytes without updating the value stored in memory.

3.3.2 Writing Data

The write operation is used to update the data stored in the peripheral. This can be used to set the SOC in the load module. If the last bit of the first byte is "1", this indicates a write operation. If the peripheral receives a byte before $\overline{\text{SS}}$ is set high, this byte is stored in memory. If $\overline{\text{SS}}$ is set high before a new byte is received, communication is reset without updating the value in the peripheral. This can be used to detect the type of the peripheral and prepare it for storing a new value, but abort the operation if the peripheral is not the type that is expected.

[4] most significant bit
[5] least significant bit

4 Hardware Hub

The hardware hub is the link between the Raspberry Pi and the Peripherals. This chapter describes the design process of the hardware hub.

4.1 Requirements

The following requirements where set for the hardware hub. Because the hardware hub will become part of a system, some of the requirements follow from the function that this part has to fulfill in the system. Key words in *ITALIC* in this section are used to indicate requirement levels, following RFC 2119[11].

- It *MUST* be possible to connect and disconnect peripherals to the hardware hub.
  - The process of connecting or disconnecting peripherals *SHOULD* be easy.
  - The number of peripherals that can simultaneously be attached *SHOULD* be maximized.
- The hardware hub *MUST* provide power to all connected peripherals.
  - The hardware hub *SHOULD* turn off power when there is an overcurrent condition.
- All voltages used *MUST* be safe to touch.
- The hardware hub *MUST* be able to be attached to a Raspberry Pi.
  - All electrical connections to the Raspberry Pi *SHOULD* be made via the GPIO port.
  - The hardware hub *MUST* conform to the electrical specifications set by the Raspberry Pi.
  - The hardware hub *SHOULD* fit on top of the Raspberry Pi.
  - The hardware hub *MAY* be compatible with other standards set by Raspberry Pi.
- The hardware hub *SHOULD NOT* be damaged by wrong connections at the connection points for the peripherals.

4.2 Powering The Peripherals

The hardware hub has to provide power to the connected peripherals. By connecting it to the Raspberry Pi via the GPIO port, it has access to the main (5 V) power supply, as well as the output from the 3.3 V regulator[15].

The output current from the 3.3 V regulator is limited, making is unsuitable for powering a large number of peripherals. The 5 V rail is connected to the power input of the Raspberry pi, and is primary limited by the adapter powering the Raspberry Pi. This is the best option for this project.

Alternatively, a system can be designed where a power supply is directly connected to the hardware hub. This offers the most flexibility. Power from the hardware hub can be used to power the Raspberry Pi. This option was not chosen because it makes the system less user friendly by having multiple power connections.

4.3 Protecting the Raspberry Pi

Because the GPIO pins of are directly connected to the main chip, they have the potential to break the Raspberry Pi when handled incorrectly[22]. To prevent damage to the Raspberry Pi, some measures are needed. The measures taken are discussed in the following sections.

4.3.1 Overcurrent Protection

Related to providing power to the peripherals, is to keep the current draw from the Raspberry Pi within the limits. Excessive current draw may damage the Raspberry Pi or cause issues with the power supply. To avoid having to replace components, a solution that resets automatically or can be electronically reset is preferred.

Two options were analyzed, a polyfuse and an electronic fuse. A polyfuse is a special type of PTC (positive temperature coefficient) resistor, placed in series with the circuit[23]. This device heats up when current flows through it. The resistance rises with temperature. At some point the increase in resistance leads to an increase of power dissipated in the device, creating more heat. When more heat is generated in the device than can be dissipated to the environment, the device starts to trip. When the device is in its tripped state, a low current keeps flowing to keep the device in its high temperature state. When all power sources are removed, the device cools down and returns to its normal state.

Because the polyfuse needs time to heat up, it cannot instantly interrupt the current. One alternative is the use of an eFuse (electronic fuse). An eFuse uses a transistor to interrupt the current, so it can function faster compared to other types of fuses[24], [25]. This makes an eFuse the most suitable choice for this project. The transistor and monitoring circuitry is packaged into an IC. EFuses can be bought specified for various voltages and maximum currents.

4.3.2 Reverse Voltage Protection

To prevent damage from an external voltage connected from the peripheral side, current in the reverse direction has to be blocked. Some eFuse ICs have support for this feature. If not, it has to be done with external components.

One way to do this using a diode. This option has some drawbacks, mainly the power lost by the voltage drop op the diode. Another way is using a Zero Volt Diode (ZVD). This is a circuit, usually contained in an IC, that acts an ideal diode without voltage drop[26]. This solves the power dissipation issue of regular diodes.

The final design has an eFuse and a ZVD.

4.4 Level Shift

The peripherals, including their microcontroller, run on 5 V. This introduces the need for level conversion to be compatible with the 3.3 V logic level of the GPIO port on the Raspberry Pi. The level shift on the outgoing signals is not always necessary, as any value above 1.8 V is read as logic high by the microcontroller[27, Figure 155]. Because the output current of the GPIO port has some restrictions[22], [28], the level converter may also act as a buffer that can handle more current compared to the GPIO port. This is necessary to make sure that the maximum current on the GPIO data pins cannot be exceeded. On the incoming signals, it has to transform the output signal from the peripheral to the input range of the Raspberry Pi.

Multiple possible solutions were analyzed. The different options with there advantages and disadvantages are listed below.

- **Resistive voltage divider**
  This option is the cheapest to implement. By matching the voltage drop over two resistors, a signal from a known level can be converted to a new level, provided that the signal source can output the required current. The output level needs to be below the input level, so it cannot be used both ways. If the input signal exceeds the level used in the design phase, the output also exceeds the intended level.

- **Zener voltage clamp**
  By replacing one of the resistors from the resistive voltage divider by an zenerdiode, the output can effectively be clamped to the level set by the zenerdiode[29], [30]. Just as by the resistive voltage divider, the output level must be below the input level.

- **Buffer/Coversion IC**
  Integrated circuits that have level conversion capabilities can be used to connect circuits with different operating voltages. They can be used for both the high to low and the low to high conversion. Some also have ability to automatically sense the direction of the signal[31]. They also provide isolation between the input and output.

- **Transistor Level Shifter**
  Transistor circuits can be used to increase signal strength. This technology can be used to change the level

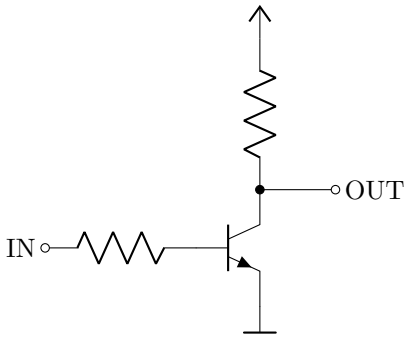


Figure 4.1: Schematic for the level conversion circuit

of digital signals[32]. By using different typologies, the input and output characteristics can be changed. This method has the disadvantage of requiring more components compared to other options.

- **Opto-Isolator**
  Light can be used to transfer information without an electrical connection[33], [34]. An opto-isolator (also known as optocoupler) consists of an light source and a detector. By applying a signal at the input, the resistance of the output can be modified. This is used as a switch, similar to the principle of a relay. Unlike a relay, opto-isolators don't have contacts or mechanical components that wear down by the switching process. The lack of mechanical components also allows for higher switching speeds compared to relays. In practice, opto-isolators use a relatively large area on the circuit board. This option provides near perfect isolation between output an input and does not require an common reference level.

The final design is a variation on the transistor level shifter. It has an open collector output with pullup resistor for the conversion of outgoing (SCLK and MOSI) lines. This is realized using a single transistor, as shown in Figure 4.1. This has the side effect of inverting the logic signal, something that can be fixed in software. While this option uses more components, the small size of the individual components results in better use of space available on the circuit board. Using open-collector outputs has the advantage over push-pull outputs that there is no excessive current when accidentally connecting two outputs or an output and ground together. After removing the short-circuit the system can continue to operate without permanent damage.

The same circuit is used for the conversion of the incoming (MISO) line. While other options are available, this makes use of the same component as used in other parts of the design. This simplifies manufacturing compared to using a new component like a zenerdiode. It also provides some level of protection for the Raspberry Pi, as an overcurrent on the external connection will damage the transistor instead of the Raspberry Pi.

4.5 Hot Plug Detect

The standard SPI bus system has no build in mechanism to detect if a peripheral is present or not. It is possible to scan all peripheral ports and look for ones that respond. This approach is slow as it takes at least the transfer time of one byte per possible peripheral. We found a solution to overcome this problem. By locating the pullup resistor for the $\overline{\text{SS}}$ line on the peripheral side of the connector, it is connected and disconnected together with the peripheral. The controller can sense the presence of this pullup resistor. It is not possible to directly sense this as a 5 Volt signal with the Raspberry Pi, it has to be converted to a 3.3 V signal. This can be done by creating a resistive voltage divider. However, this option was not preferred because it does not protect the Raspberry Pi pin in case of an overvoltage condition.

A circuit similar to the MISO buffer is made using Q1. An LED is placed in series with the pullup resistor to indicate the presence of a connected peripheral. The output goes low when a peripheral is connected. The final circuit is shown in Figure 4.2. This creates a voltage divider consisting of R1 and R2, combined with the Base-Emitter drop of Q1. with correct resistor values, this does not interfere with the logic circuity on the peripheral, as any value above 1.8 V reads as logic high[27, Figure 155]. Q2 can be used to pull the line low and set the active state. A limitation of this implementation is that does not allow for sensing while $\overline{\text{SS}}$ is in its active (Low) state. It also turns off the LED during the communication. This side effect shows when a peripheral is communicating, and can be considered quite useful.

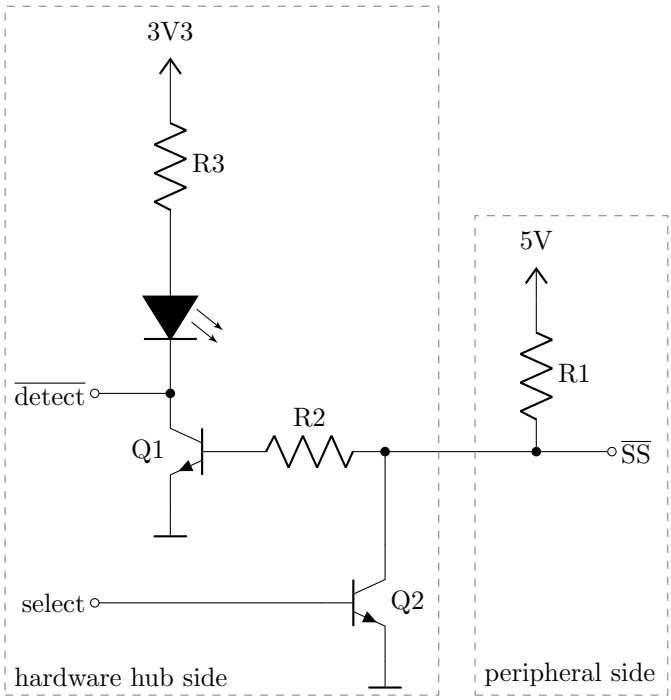


Figure 4.2: Schematic for the detection and select circuit

5 Peripherals

The energy system integration demonstrator has four peripherals: a solar panel representation, a storage representation, a wind turbine representation and a load representation. Each peripheral consists of two parts, namely, an electronic circuit to represent the energy technology and a microcontroller. The microcontroller will take care of the required processing of the signal coming from the electronic circuit before the signal is feed to the Raspberry Pi hub. The microcontroller that was selected is the Atmega8. The reason that an Atmega is chosen is that it is the simplest to program, otherwise too much time would be spent to learn how to program a microcontroller. The processing of the signal could be also done with an Arduino, however, by using a microcontroller the expenses would be less and the design would be smaller.

5.1 Solar Panel

5.1.1 Requirements

To mimic the behaviour of a solar cell, a circuit had to be designed whereby:

- The output signal increases when the illumination increases and decreases when the illumination decreases.

5.1.2 Design and Choices

There are two designs options: one with a photodiode and one with a LDR(Light Dependent Resistor). A photodiode generates a current when it is illuminated. A LDR is a type of resistor in which its resistance depends on the amount of illumination it receives, the resistance decreases when illumination increases. To realise the photodiode, more components were necessary. The photodiode needs a transimpedance amplifier[35] to convert the current coming from the photodiode to a voltage at the output of the solar cell circuit, while for the LDR design this was not necessary. Furthermore, the photodiode is more accurate compared to the LDR [36], [37]. However, a high accuracy is not necessary in this case. A LDR based circuit would function as desired and it did have a simpler design, for the time being it was beneficial to use the simpler design. In addition, the photodiode would have been relatively more costly to make since it needs more components. Hence, it was decided to build the LDR based solar panel representation circuit.

5.2 Storage

5.2.1 Requirements

- The storage peripheral should indicate the amount of energy that is stored.

5.2.2 Design and Choices

The amount of stored energy is refered as the State Of Charge (SOC). The SOC was given by a digital signal from the Raspberry Pi. It was decided to use five LEDs(Light Emitting Diode) to indicate the SOC, with each LED indicating to 20% charge. The five LEDs would be sufficient in our case to indicate the SOC. The colors that are used are red, yellow and green. The peripheral consist of two reds, two yellow and one green. In Table 5.1 the SOC corresponding to which LED or LEDs is shown.

Table 5.1: SOC in correspondence to a LED or LEDS

SOC	1%-20%	20%-40%	40%-60%	60%-80%	80%-100%
Red LED	on	on	on	on	on
Red LED	off	on	on	on	on
Yellow LED	off	off	on	on	on
Yellow LED	off	off	off	on	on
Green LED	off	off	off	off	on

5.3 Wind Turbine

5.3.1 Requirements

The following requirements apply to the wind turbine representation:

- The wind turbine representation should operate in two ways: by a manual input(rotating the blade by hand) and by receiving an input from the Raspberry Pi(blades rotate based on a signal).
- The rotation speed the motor should vary according to a giving input
- The motor should not exceeds an output voltage of 5V.
- The motor should be able to rotate at a low speed such that it is more realistic compared to a real wind turbine.

5.3.2 Design and Choices

There are two design option, namely, using a servo motor or a DC motor. The advantage of a servo motor is that it has a relatively low RPM so it can easily be implemented to rotate at low speeds. The disadvantage with a servo motor is that it can not function manually, i.e rotating the turbine by hand. For this reason the DC motor was selected. In order to stay as close to the 5V a 6V dc motor was used. The datasheet [38] was studied whether it was possible that it rotates at a low speed.

In order to vary the DC motor speed, the positive terminal of the motor should be connected to a PWM pin on the microcontroller, however, this pin will not be able to supply a sufficient amount of current to support the motor. The only pin capable to do so was the 5V supply pin. Since the variability of the motor speed was a necessity, this was not a valid option. To still use this pin, a NMOS transistor was added to the circuit. The NMOS could either be placed between the 5V line and the positive terminal or the negative terminal and the ground. Since, a NMOS does not function in an optimal way for high level switching, the latter option was selected, this was confirmed by simulating it in Falstad circuit simulator applet and reading out the voltage value. When the NMOS was placed between 5V and the positive terminal, the voltage could not reach or even come close to 5V while when the NMOS was placed between the negative terminal of the motor and the ground it reached the 5V.

Since the DC motor has inductive characteristics a flyback diode has to be placed in anti-parallel with the positive and negative terminal of the motor to prevent huge voltage spikes that could damage the motor or the circuit.

In order to switch between manual input and input from the Raspberry Pi, a manual switch was used.

5.4 Load

5.4.1 Requirements

The following requirements apply to the load representation:

- The load module must indicate the current load.

5.4.2 Design and Choices

To display the load, two 7-segment displays were used. Because the microcontroller has a limited number of IO pins, multiplexing was used to drive the displays. The display modules are internally wired in a common cathode configuration.

6 Prototype Implementation and Validation Results

6.1 Hardware Hub

A design for the hardware hub was made using KiCad. This design can be found in Appendix A. The design includes the level converters as described in section 4.4. It also contains protection for the power rail, as described in section 4.3.

As connectors for the peripherals, the 6P6C plug was used. This connector has 6 pins[39]. It was chosen because it is readily available from suppliers, but less commonly used in consumer products. This was done to avoid other cables being plugged in, something that may cause damage because the pinout for this application is not shared with other products.

The pinout for the connector was chosen in a way to minimise the possible damage to the hardware hub from connecting two ports using a cross-cable. The connections made by such a cable are shown in Table 6.1. There is no physical damage to the hardware hub or the Raspberry Pi by making any of these connections. Power gets connected to a data input and open collector outputs get connected to other open collector outputs or to ground. Protection of connected peripherals in not guaranteed. It also makes all communication impossible until the invalid connection are removed.

A eeprom IC was added to the design to make it compatible with standards set by Raspberry Pi[15], [17], [40]. This eeprom is used by the Raspberry Pi during startup to detect attached hardware, such as the hardware hub. This is not necessary for the functioning of the hardware hub, and was omitted from the prototype because of time constraints. The pcb has this connection available for future use.

A layout for the circuit board was made using this schematic. The manufacturing of the PCB[1] was done by an external company. All components were soldered by hand.

6.1.1 Testing

After assembly, the hardware hub was attached to a Raspberry Pi and tested in various ways.

The electronic fuse was tested by creating a short-circuit between power and ground on one of the connectors. This resulted in all connected peripherals losing power, but the Raspberry Pi did not turn off. This meant that the protection circuitry acted fast enough to isolate the Raspberry Pi. After removing the short-circuit, everything continued to function as expected.

To test the SPI signals, a Tektronix logic analyser was used. A microcontroler was attached to test the communication. All signal lines behaved as expected.

6.2 Peripherals

Before the circuits were soldered they were first tested with an Arduino and a breadboard. If the results were correct and as desired, the Arduino was replaced by the microcontroller. The microcontroller was connected to the Raspberry Pi. If at this testing round the results were as desired, the microcontroller and the electronic components were soldered to a board. The reason that it is tested on a bread board is that the circuit is simpler

Table 6.1: Connections resulting from connecting a cross cable

pin 1	Power	+5V	⟷	MISO	input	pin 6
pin 2	open collector	MOSI	⟷	SCK	open collector	pin 5
pin 3	ground	GND	⟷	SS	open collector	pin 4

[1] printed circuit board

to adapt if adaptation is needed and the reason that first the Arduino is used for testing is that coding on the Arduino is easier than on the microcontroller and if it is not functioning software wise the only place to look is the Arduino code instead of both the microcontroller and Raspberry Pi code. The code for the Arduino and the microcontroller were written in the Arduino IDE. At first it was tried to upload the code via the Arduino, however, this did not work so an alternative had to be found. This alternative was uploading the code via an AVR USB programmer. For this Zadig[41] was used to install the USB driver. The code is still written and compiled in the Arduino IDE but it is uploaded via the windows power shell. While uploading to the microcontroller the slave select(SS) pin should be connected to the reset pin. This connection can be made using a jumper header. This allows to use the same connector for both for connector to the hardware hub as for programming.

6.2.1 Solar Panel

Prototype Implementation

The LDR circuit consists of a 5V supply voltage from the microcontroller, which is powered by the Raspberry Pi, a resistor at the output and the LDR itself, in Figure 6.1 the design is shown.

In reality, the more illumination a solar panel receives the more electricity it generates. When the LDR is illuminated, its resistance will decrease which leads to a decrease of the voltage over the LDR and an increase in voltage over the resistor at the output. The output is taken over this resistor. The voltage at the output of the electronic circuit is analog, however, to feed it to the Raspberry Pi it should be a digital signal. For this purpose the microcontroller is used. The microcontroller has a built-in ADC which converts the analog signal into a digital signal. The digital signal is transmitted to the Raspberry Pi.

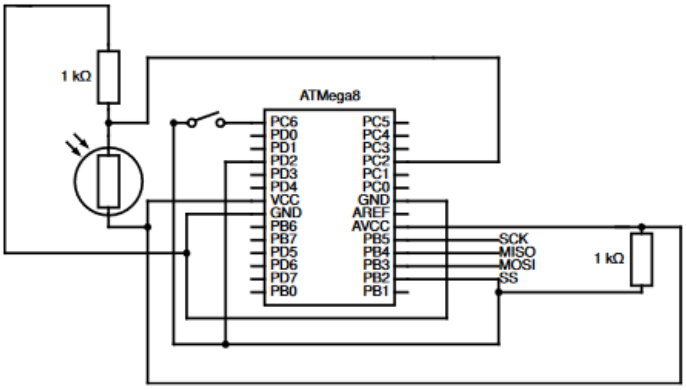


Figure 6.1: Circuit diagram of the solar panel representation.

The resistance was determined by using the voltage divider equation. This equation is shown in Equation 6.1. It was desired that the output of the circuit, be as close as possible to 0V when the LDR is completely shaded and as close as possible to 5V when it is illuminated with a huge light intensity, i.e. shinning a flash light perpendicular. This would give a more accurate and real-life representation of a solar panel. A resistor with a low, a high and a resistance with a common value were selected and used in Equation 6.1. The results are shown in Table 6.2 and Table 6.3. It can be observed that the higher the resistance, the closer the value of V_{out}, which is the voltage over the output resistance, is to 5V when illuminated with a high light intensity and the higher V_{out} is when the LDR is completely shaded. When low resistance is selected it would have a lower output voltage when completely shaded, however, a relatively lower output voltage is present when it is highly illuminated. So the resistance could be neither too high or too low. The 1kΩ resistor has a value sufficient close to 5V and has a relatively low V_{out}. Hence, the 1kΩ resistor was selected to be used.

$$V_{out} = V_{in} . \frac{R}{R_{LDR} + R} \tag{6.1}$$

Table 6.2: Output voltage when the LDR is completely shaded

	V_{out}
R_{LDR}=30kΩ, R=500Ω	0.082
R_{LDR}=30kΩ, R=1kΩ	0.16
R_{LDR}=30kΩ, R=5kΩ	0.71

Table 6.3: Output voltage when the LDR is maximally illuminated

	V_{out}
R_{LDR}=20Ω, R=500Ω	4.8
R_{LDR}=20Ω, R=1kΩ	4.9
R_{LDR}=20Ω, R=5kΩ	4.98

6.2.2 Testing

When testing with the Arduino, a simple code was written in the Arduino IDE and uploaded to the Arduino. The code instructed the ADC pin A0 on the Arduino to read the analog value at the solar cell output and convert this to a digital value. The digital value was then displayed on the laptop using the Arduino serial monitor. See section B.1 for the code. When testing with the microcontroller, the electronic circuit was connected to the microcontroller. The microcontroller carried out the same task as the Arduino. The microcontroller sent the digitalized output of the solar circuit when the Raspberry Pi inquired for it, the output was displayed on a screen. See section C.1 for the code. For the pin layout of the microcontroller the datasheet [27] was used. When the results were correct and as desired, the circuit was soldered.

Measurement Results

After soldering the circuit the voltage across the 1 kΩ was measured with a multimeter. The voltage was measured and observed whether it corresponds with the digital output of the Raspberry Pi. Obviously measuring all 255 values was not reasonable for the time being. Instead, six values were chosen, two at low range, two at middle range and two at high range. The reason for this is that in this case it could be confirmed that the solar circuit function correctly at the whole range and two values were chosen to have a second confirmation. The results are shown in Table 6.4

Table 6.4: Measurement results of solar cell circuit

Ratio of measured value	Ratio of digital value
0.4/5=0.08	18/255= 0.07
1.7/5=0.34	84/255= 0.33
2.3/5=0.46	114/255=0.45
3.7/5=0.74	181/255=0.71
4.6/5=0.92	230/255=0.90
4.9/5=0.98	247/255=0.97

The results are satisfying since the digital output ratio corresponds with the ratio of the measured analog value. It is noticed that the measured analog ratio and the ratio of the digital value are not exactly the same. A reason for this was that there could be a small lag between the signal that was shown on the computer and the signal that was measured(real time). However, since this circuit serves as an indication such small differences were not an issue.

6.2.3 Storage

Prototype Implementation

The electronic circuit of the storage indicator consists of a LED, connected to a pin on the microcontroller, and a resistor in series, connected to the 5V supply on the microcontroller. This is done five times in parallel. See Figure 6.2 for the circuit. The pin connected to the microcontroller is initially set to "HIGH" such that no

current flows through the circuit and so the LED stays off. When a specific LED needs to be turned on, the corresponding pin is set to "LOW" such that a current flow through the circuit and allows the LED light up.

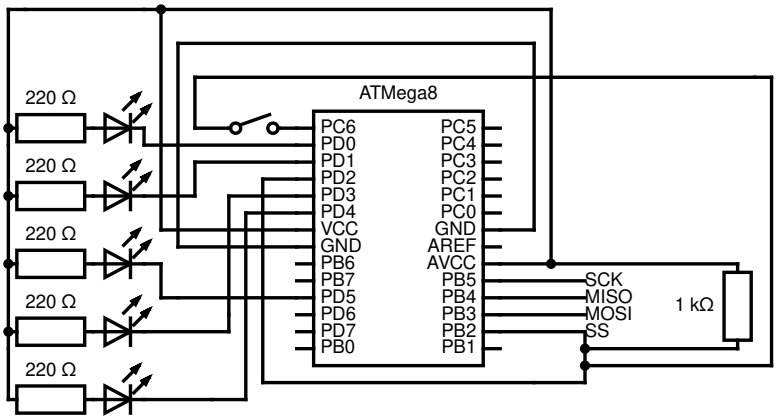


Figure 6.2: Circuit diagram of storage representation.

The maximum current through a LED is commonly 20mA and the voltage over the selected LEDs are between 1.8V to 2.2V [42], [43]. Using Equation 6.2 a resistor with a resistance of 160Ω would be sufficient but from experience it is better to choose a higher resistance out of safety. A resistance of 220Ω was selected. The current would be reduced to 10mA-15mA but from experience the LEDs will still function. The 20mA is a maximum, if a value under the 20mA is used the LEDs will still turn on.

$$R = \frac{V_{in} - V_{out}}{I} = \frac{5 - 1.8}{20m} = 160\Omega \tag{6.2}$$

6.2.4 Testing

For testing with the Arduino a code was written that would set a pin or pins to low depending on the SOC variable. This variable was programmed to change at a specific moment, the code is shown in section B.3. When testing with the microcontroller, a code was written that would set the desired pin or pins to low depending on a signal that carried the information of the SOC. The SOC was given by the Raspberry Pi. See section B.4 for the code.

Measurement Results

After soldering, the circuit underwent a final test to know if the circuit is functioning as desired. A few random SOC values were given from the Raspberry Pi and observed if the corresponding LED or LEDs turned on. The result was as desired since the corresponding LED or LEDs turned on. The code used for this is shown in section B.4

6.2.5 Wind Turbine

Prototype Implementation

Like stated in chapter 5, the NMOS should be placed between the negative terminal of the motor and the ground. Hence, the drain of the NMOS was connected to the negative terminal of the motor and the source of it was grounded, the gate was connected to PWM pin PB1 on the microcontroller. The positive terminal of the DC motor was connected to the VCC(5V) pin of the microcontroller. Depending on the PWM value the NMOS was turned on and off, in this way the motor speed was controlled.
The manual switch was configured as follows: set to one side, the switch provides a connection between ground and the negative side of the DC motor through one of its channels and through the other channel a connection between the ADC pin of the microcontroller and the positive terminal of the motor was made. If the switch is set to the otherside, the negative terminal of the DC motor makes a connection with the drain of the transistor through one of its channels and through the other channel the positive terminal is connected to the the VCC pin of the microcontroller. In Figure 6.3 the circuit diagram of the wind turbine representation is shown.

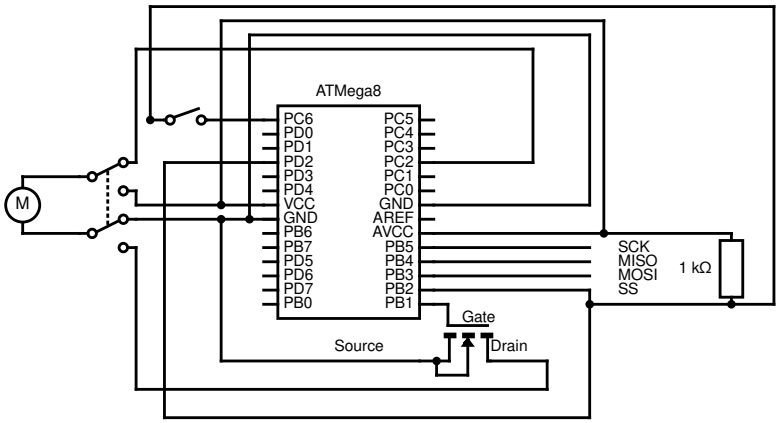


Figure 6.3: Circuit diagram of wind turbine representation.

6.2.6 Testing

The code for the Arduino and the microcontroller is shown in section B.5 and section B.2 respectively. For the PWM signal on the microcontroller pin 15, which is named PB1, was used. However, the microcontroller only recognises this pin as a PWM signal if it is defined as pin 9, which corresponds to an Arduino PWM pin [44].

Measurement Results

After soldering, a final test was conducted on whether the rotation speed increases when the PWM signal increases and decreases when the PWM signal decreases, and that the flyback diode is functioning as desired. The oscilloscope was used to test the flyback diode. The setup was as follows: one of the probes was connected to the positive terminal and one to the negative terminal of the motor. Since the negative terminal of the motor was not grounded(it was connected to the drain of the NMOS) it was not possible to use the ground probe. A solution was to use two cables and use one cable probe of each cable. Then the math mode on the oscilloscope was used to take the difference between the two signals. It was observed that there were no spikes. To test whether the rotation speed varies correctly. The code shown in section B.2 was used. The PWM signal was given by the Raspberry Pi. When the PWM signal was varied the rotation speed of the turbine changed accordingly. Hence, it was concluded that the wind turbine representation functioned as desired.

6.2.7 Load

Prototype Implementation

NPN transistors where used to multiplex displays. While other types of transistors may have been more suitable for this application, component availability and time constraints led to this design.

6.2.8 Testing

Tests showed that the module is capable of displaying numbers.

Measurement Results

No measurements were taken on this module

7 Discussion of Results

In the previous chapter the results of the individual peripherals were discussed. To give a general summary, the measured/observed results were satisfying since the outcome of the test were reasonable and as expected. In this chapter the focus will lie on the discussion of the results when the hardware was implemented with the simulation group. The setup was as follows: like testing the individual peripherals, a peripheral was connected to the Raspberry Pi. The code of the simulation group was uploaded to the Raspberry Pi. This code displayed a real-time graph of the variation in the output of the solar panel and wind turbine peripheral. It was observed that the graph depending on the amount of illumination increased or decreased for the solar panel and the graph increased or decreased depending on the rotation speed of the wind turbine. The rotation speed of the wind turbine was given by a manual input, i.e. the turbine was rotated by hand. The code for the Raspberry Pi is shown in section C.4. In conclusion, the integration with the simulation group functioned correctly.

8 Conclusion

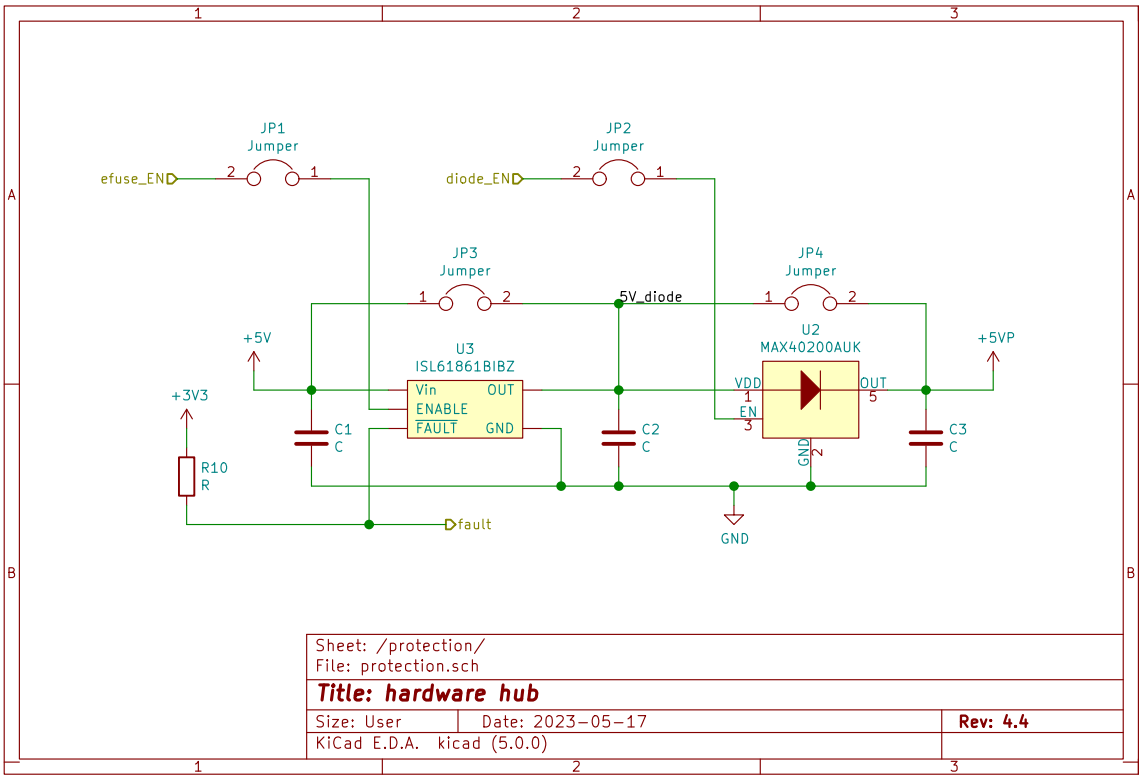
This project succeeded in building hardware compatible with the existing simulation software. Physical signals can be sensed by the peripherals and used by the simulation software running on the Raspberry Pi.

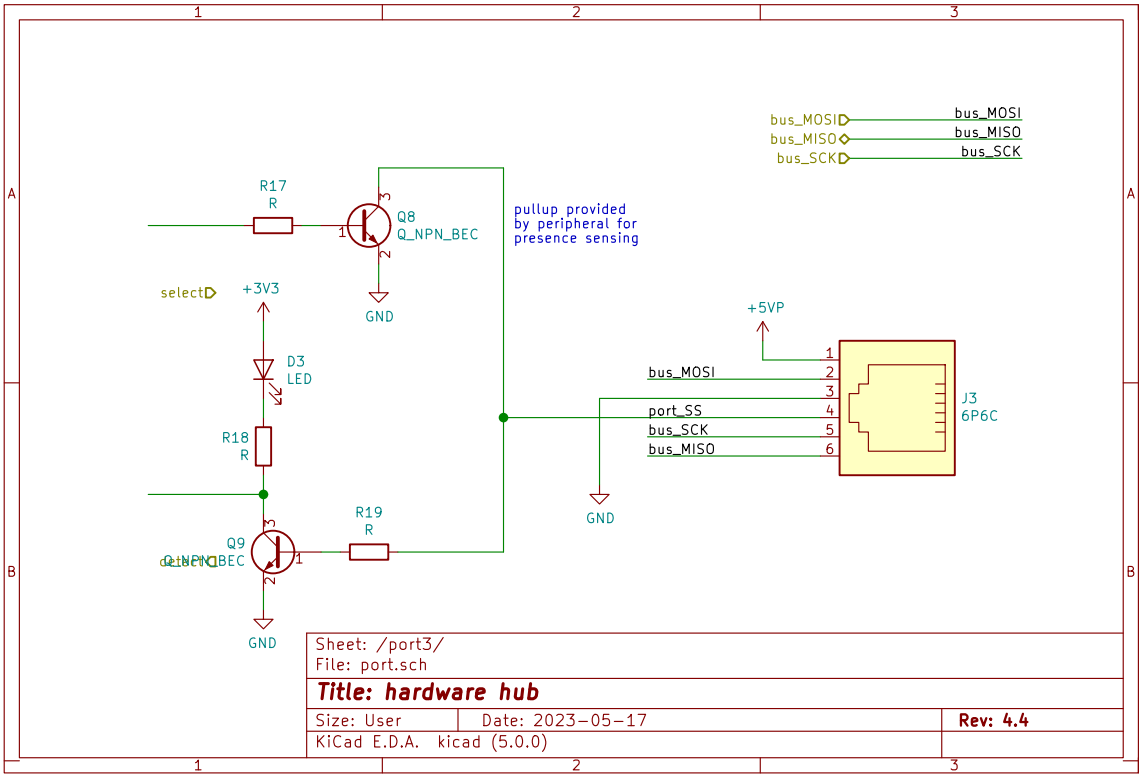
All peripherals were implemented using atmega8 microcontrollers. While other microcontollers may exist that are more suited for this application, both the availability and past programming experience for this architecture favoured this chip. Limitations of this chip where solved by creative hardware and software solutions.

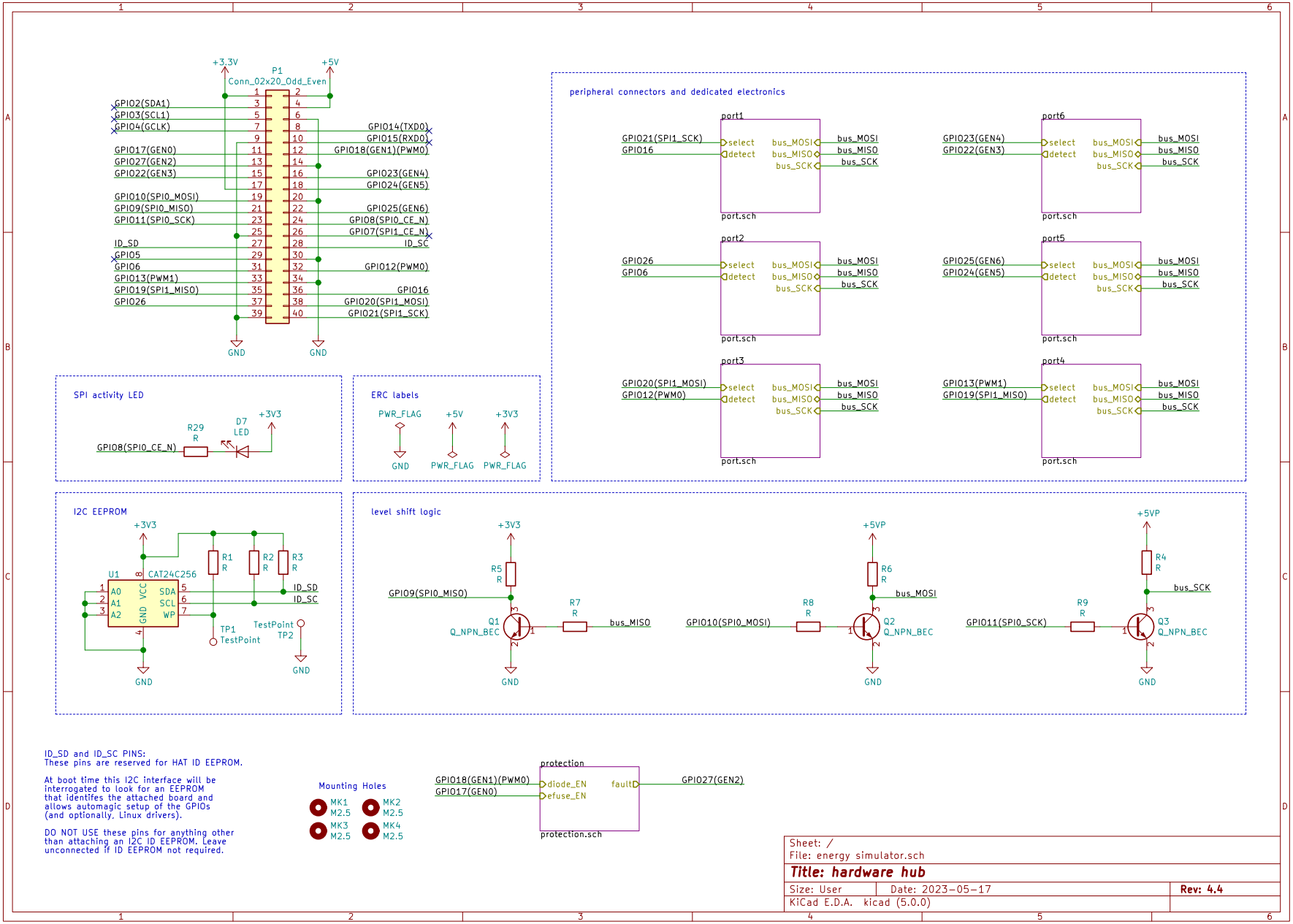
One thing we did not succeed in was creating 3D models for the peripherals. The plan was to put the peripherals in enclosures with shapes that represent the component that it simulates. While this addition does not bring big technical challenges, it was omitted in from the project because we could not achieve this in the time constraint. This is something that can be looked at in future versions of the project.

A

Full Hardware Hub Schematics




bus_MOSI
bus_MISO
bus_SCK
R17
R
Q8
Q_NPN_BEC
pullup provided
by peripheral for
presence sensing
select
+3V3
GND
D3
LED
R18
R
R19
R
Q9
Q_NPN_BEC
+5VP
port_SS
J3
6P6C
Sheet: /port3/
File: port.sch
Title: hardware hub
Size: User
Date: 2023-05-17
Rev: 4.4
KiCad E.D.A. kicad (5.0.0)


P1
Conn_02x20_Odd_Even
+3.3V
+5V
GPIO2(SDA1)
GPIO3(SCL1)
GPIO4(GCLK)
GPIO14(TXD0)
GPIO15(RXD0)
GPIO17(GEN0)
GPIO18(GEN1)(PWM0)
GPIO27(GEN2)
GPIO22(GEN3)
GPIO23(GEN4)
GPIO24(GEN5)
GPIO10(SPI0_MOSI)
GPIO9(SPI0_MISO)
GPIO11(SPI0_SCK)
GPIO25(GEN6)
GPIO8(SPI0_CE_N)
GPIO7(SPI1_CE_N)
ID_SD
ID_SC
GPIO5
GPIO6
GPIO12(PWM0)
GPIO13(PWM1)
GPIO19(SPI1_MISO)
GPIO16
GPIO26
GPIO20(SPI1_MOSI)
GPIO21(SPI1_SCK)
GND
GND
peripheral connectors and dedicated electronics
port1
port2
port3
port4
port5
port6
select
detect
bus_MOSI
bus_MISO
bus_SCK
port.sch
SPI activity LED
R29
R
D7
LED
+3V3
GPIO8(SPI0_CE_N)
ERC labels
PWR_FLAG
+5V
+3V3
GND
PWR_FLAG
PWR_FLAG
I2C EEPROM
U1
CAT24C256
A0
A1
A2
GND
VCC
SDA
SCL
WP
R1
R2
R3
TP1
TestPoint
TP2
level shift logic
R5
R6
R4
R7
R8
R9
Q1
Q2
Q3
Q_NPN_BEC
+5VP
bus_MOSI
bus_MISO
bus_SCK
ID_SD and ID_SC PINS:
These pins are reserved for HAT ID EEPROM.
At boot time this I2C interface will be interrogated to look for an EEPROM that identifes the attached board and allows automagic setup of the GPIOs (and optionally, Linux drivers).
DO NOT USE these pins for anything other than attaching an I2C ID EEPROM. Leave unconnected if ID EEPROM not required.
Mounting Holes
MK1 M2.5
MK2 M2.5
MK3 M2.5
MK4 M2.5
protection
diode_EN
efuse_EN
fault
protection.sch
Sheet: /
File: energy simulator.sch
Title: hardware hub
Size: User
Date: 2023-05-17
Rev: 4.4
KiCad E.D.A. kicad (5.0.0)

B Test Codes

This appendix contains the codes

B.1 Arduino Code Solar

```
int solar = A0;   // pin that read the solar output
int digitalValue = 0;  // variable to store the value of the
//converted to digital value of the solar output
float analogVoltage = 0.00;

void setup() {
  Serial.begin(9600);
}

void loop() {
  digitalValue = analogRead(solar);// read the analog value at the output
  //of the solar cell and convert it to a digital value
  Serial.print("digital value = ");
  Serial.print(digitalValue);        //print digital value on serial monitor
  //convert digital value to analog voltage to know if they corresponds to each other
  analogVoltage = (digitalValue * 5.00)/1023.00;
  Serial.print("  analog voltage = ");
  Serial.println(analogVoltage);
  delay(1000);

}
```

B.2 Microcontroller Code Wind Turbine

```
#include <SPI.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define PERIPHERALID 102
int pwm=9;            // from micro to hardware
volatile byte pwm_pi=0; //from pi to micro
int adc = PC2;
int digitalValue = 0;  // variable to store the value coming from the sensor

void setup(){

pinMode(9, OUTPUT);
pinMode(MISO, OUTPUT);
pinMode(PC2, INPUT);


  // have to send on master in, *slave out*
```

```
  // turn on SPI in slave mode
  SPCR |= _BV(SPE);

  // turn on interrupts
  SPCR |= _BV(SPIE);

  // set INT0 mode
    MCUCR |= _BV(ISC01);
    // set INT0 interrupt
    GICR |= _BV(INT0);

}

ISR (SPI_STC_vect)
{
 pwm_pi=SPDR;
 SPDR=digitalValue;

}  // end of interrupt service routine (ISR) SPI_STC_vect

// INT0 interrupt routine
ISR (INT0_vect)
{
    SPDR = PERIPHERALID;
}  // end of interrupt service routine (ISR) INT0_vect


void loop() {
  analogWrite(pwm, pwm_pi);
  digitalValue = analogRead(adc);// read the value from the analog channel
}
```

B.3 Arduino Code Storage

```
int red1= 2;
int red2= 3;
int yellow1= 4;
int yellow2= 5;
int green= 6;
int percentage;

void setup(){
pinMode(2, OUTPUT);
pinMode(3, OUTPUT);
pinMode(4, OUTPUT);
pinMode(5, OUTPUT);
pinMode(6, OUTPUT);
digitalWrite(red1, HIGH);
digitalWrite(red2, HIGH);
digitalWrite(yellow1, HIGH);
digitalWrite(yellow2, HIGH);
digitalWrite(green, HIGH);
percentage=0;
}

void loop()
{ percentage=0.20;
```

```
  if(percentage= 0.20);
  digitalWrite(red1, LOW);
  delay(1000);

  percentage=0.40;
  if(percentage= 0.40);
  digitalWrite(red2, LOW);
  delay(1000);

  percentage=0.60;
  if(percentage= 0.60);
  digitalWrite(yellow1, LOW);
  delay(1000);

  percentage=0.80;
  if(percentage= 0.80);
  digitalWrite(yellow2, LOW);
  delay(1000);

  percentage=1;
  if(percentage= 1);
  digitalWrite(green, LOW);
  delay(1000);

}
```

B.4 Microcontroller Code Storage

```
#include <SPI.h>
int red1= PD0;
int red2= PD1;
int yellow1= PD5;
int yellow2= PD3;
int green= PD4;
volatile byte soc=0;
#define PERIPHERALID 105

void setup(){
pinMode(PD0, OUTPUT);
pinMode(PD1, OUTPUT);
pinMode(PD5, OUTPUT);
pinMode(PD3, OUTPUT);
pinMode(PD4, OUTPUT);
pinMode(MISO, OUTPUT);
digitalWrite(red1, HIGH);
digitalWrite(red2, HIGH);
digitalWrite(yellow1, HIGH);
digitalWrite(yellow2, HIGH);
digitalWrite(green, HIGH);

  // have to send on master in, *slave out*

  // turn on SPI in slave mode
  SPCR |= _BV(SPE);

  // turn on interrupts
  SPCR |= _BV(SPIE);
```

```
  // set INT0 mode
    MCUCR |= _BV(ISC01);
    // set INT0 interrupt
    GICR |= _BV(INT0);

}

ISR (SPI_STC_vect)
{
  soc=SPDR;
  //SPDR=42;


}  // end of interrupt service routine (ISR) SPI_STC_vect

// INT0 interrupt routine
ISR (INT0_vect)
{
    SPDR = PERIPHERALID;
}  // end of interrupt service routine (ISR) INT0_vect


void loop()
{ //20%
  if(soc>0 && soc<=50){
  digitalWrite(red1, LOW);
  digitalWrite(red2, HIGH);
  digitalWrite(yellow1, HIGH);
  digitalWrite(yellow2, HIGH);
  digitalWrite(green, HIGH);
  }
  //40%
  if(soc>50 && soc<=100){
  digitalWrite(red1, LOW);
  digitalWrite(red2, LOW);
  digitalWrite(yellow1, HIGH);
  digitalWrite(yellow2, HIGH);
  digitalWrite(green, HIGH);
  }

  if(soc>100 && soc<=150){
  digitalWrite(red1, LOW);
  digitalWrite(red2, LOW);
  digitalWrite(yellow1, LOW);
  digitalWrite(yellow2, HIGH);
  digitalWrite(green, HIGH);
  }

  if(soc>150 && soc<=200){
  digitalWrite(red1, LOW);
  digitalWrite(red2, LOW);
  digitalWrite(yellow1, LOW);
  digitalWrite(yellow2, LOW);
  digitalWrite(green, HIGH);
  }

  if(soc>200 && soc<=255){
  digitalWrite(red1, LOW);
```

```
  digitalWrite(red2, LOW);
  digitalWrite(yellow1, LOW);
  digitalWrite(yellow2, LOW);
  digitalWrite(green, LOW);
  }

}
```

B.5 Arduino Code for the Wind Turbine

```
int WT_hand=A0;
int pwm=3;
int digitalValue=0;

void setup() {
 pinMode(A0, INPUT);
 pinMode(3, OUTPUT);
 Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
digitalValue= analogRead(WT_hand);

Serial.println(" WT_hand= ");
Serial.print(digitalValue);
delay(1000);

analogWrite(3,150);

}
```

C Final Codes

C.1 Code for the solar module

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define PERIPHERALID 101

int main(void)
{
    // solar specific code
    ADMUX = 0b01100010;                    // set input channel
    ADCSRA |= _BV(ADPS1) | _BV(ADPS1);     // divide clk by 8
    ADCSRA |= _BV(ADEN) | _BV(ADFR) | _BV(ADSC);   // start adc

    SPCR |= _BV(SPE);       // turn on SPI in slave mode
    DDRB |= _BV(DDB4);      // set MISO as output
    SPCR |= _BV(SPIE);      // set SPI interrupt

    MCUCR |= _BV(ISC01);    // set INT0 mode
    GICR |= _BV(INT0);      // set INT0 interrupt

    sei();                  // turn on global interrupts

    while(1)
        ;  // do nothing
    return 0;
}

volatile uint8_t bytenumber;

ISR (SPI_STC_vect)      // SPI interrupt routine
{
    bytenumber++;
    if (bytenumber == 1){
        SPDR = ADCH;  // get the most recent value from the adc
    }
}

ISR (INT0_vect)         // INT0 interrupt routine
{
    SPDR = PERIPHERALID;
    bytenumber = 0;
}
```

C.2 Code for the storage module

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define PERIPHERALID 105

volatile uint8_t SOC;
void setSoc(uint8_t state);
volatile uint8_t bytenumber;
volatile uint8_t RW;

int main(void)
{
    SPCR |= _BV(SPE);       // turn on SPI in slave mode
    DDRB |= _BV(DDB4);      // set MISO as output
    SPCR |= _BV(SPIE);      // set SPI interrupt

    MCUCR |= _BV(ISC01);    // set INT0 mode
    GICR |= _BV(INT0);      // set INT0 interrupt

    DDRD |= 0b00111011;     // set pins as output

    sei();                  // turn on global interrupts

    while(1){
        setSoc(SOC);          // display the current SOC
    }
    return 0;
}

ISR (SPI_STC_vect)      // SPI interrupt routine
{
    bytenumber++;
    if (bytenumber == 1){
        RW = SPDR & 0x01;
    }
    else if (bytenumber == 2){
            if (RW){
                SOC = SPDR;
            }
        SPDR = SOC;
    }
    else
    {
        SPDR = bytenumber*10;
    }
}

ISR (INT0_vect)         // INT0 interrupt routine
{
    SPDR = PERIPHERALID;
    bytenumber = 0;
}

void setSoc(uint8_t state){     // output to the leds
    PORTD &= ~(1 << 0);     // rood 1 PD0
```

```
    if (state > 50)         // rood2 PD1
        PORTD &= ~(1 << 1);
    else
        PORTD |= (1 << 1);

    if (state > 100)        // geel 1 PD5
        PORTD &= ~(1 << 5);
    else
        PORTD |= (1 << 5);

    if (state > 150)        // geel 2 PD3
        PORTD &= ~(1 << 3);
    else
        PORTD |= (1 << 3);

    if (state > 200)        // groen PD4
        PORTD &= ~(1 << 4);
    else
        PORTD |= (1 << 4);
}
```

C.3 Code for the load module

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define F_CPU 1000000UL  // 1 MHz
#include <util/delay.h>

#define PERIPHERALID 105

volatile uint8_t load;
volatile uint8_t bytenumber;
volatile uint8_t RW;
void setDisplay(uint8_t arg);

int main(void)
{
    SPCR |= _BV(SPE);       // turn on SPI in slave mode
    DDRB |= _BV(DDB4);      // set MISO as output
    SPCR |= _BV(SPIE);      // set SPI interrupt

    MCUCR |= _BV(ISC01);    // set INT0 mode
    GICR |= _BV(INT0);      // set INT0 interrupt

    DDRD |= 0b11111011;     // set pins as output
    DDRB |= 0b10000011;

    sei();                  // turn on global interrupts

    while(1){
        PORTB &= 0b11111100;       // all off
        setDisplay(load/10);       // set display
        PORTB |= 0b00000010;       // left display on
        _delay_ms(1);

        PORTB &= 0b11111100;        // all off
        setDisplay(load%10);        // set display
```

```
        PORTB |= 0b00000001;        // right display on
        _delay_ms(1);
    }
    return 0;
}

ISR (SPI_STC_vect)      // SPI interrupt routine
{
    bytenumber++;
    if (bytenumber == 1){
        RW = SPDR & 0x01;
    }
    else if (bytenumber == 2){
            if (RW){
                load = SPDR;
            }
        SPDR = load;
    }
}

ISR (INT0_vect)         // INT0 interrupt routine
{
    SPDR = PERIPHERALID;
    bytenumber = 0;
}

void setDisplay(uint8_t arg){
    PORTD &= 0b00000100;    // clear all segment outputs
    PORTB &= 0b01111111;
    switch (arg){           // set the correct bits for the number
        case 0:
            PORTD |= 0b01111010;
            PORTB |= 0b10000000;
            break;
        case 1:
            PORTD |= 0b00000010;
            PORTB |= 0b10000000;
            break;
        case 2:
            PORTD |= 0b10111000;
            PORTB |= 0b10000000;
            break;
        case 3:
            PORTD |= 0b10101010;
            PORTB |= 0b10000000;
            break;
        case 4:
            PORTD |= 0b11000010;
            PORTB |= 0b10000000;
            break;
        case 5:
            PORTD |= 0b11101010;
            PORTB |= 0b00000000;
            break;
        case 6:
            PORTD |= 0b11111010;
            PORTB |= 0b00000000;
            break;
```

```
        case 7:
            PORTD |= 0b00100010;
            PORTB |= 0b10000000;
            break;
        case 8:
            PORTD |= 0b11111010;
            PORTB |= 0b10000000;
            break;
        case 9:
            PORTD |= 0b11101010;
            PORTB |= 0b10000000;
            break;
    }
}
```

C.4 Code Raspberry Pi for integration

```
# code voor de hardware
import RPi.GPIO as GPIO
from time import sleep
import spidev
import random



GPIO.setmode(GPIO.BCM)
select_pins = [21, 26, 20, 13, 25, 23]
detect_pins = [16, 6,  12, 19, 24, 22]



ports = [
    {
    "name" : "j1",
    "select" : 21,
    "detect" : 16,
    },
    {
    "name" : "j2",
    "select" : 26,
    "detect" : 6,
    },
    {
    "name" : "j3",
    "select" : 20,
    "detect" : 12,
    },
    {
    "name" : "j4",
    "select" : 13,
    "detect" : 19,
    },
    {
    "name" : "j5",
    "select" : 25,
    "detect" : 24,
```

```
    },
    {
    "name" : "j6",
    "select" : 23,
    "detect" : 22,
    },
    ]



GPIO.setup(17, GPIO.OUT) # diode_EN
GPIO.setup(18, GPIO.OUT) # efuse_EN
GPIO.output(17, GPIO.HIGH)
GPIO.output(18, GPIO.HIGH)



for pin in select_pins:
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)
for pin in detect_pins:
    GPIO.setup(pin, GPIO.IN)




spi = spidev.SpiDev()
spi.open(0,0)
spi.max_speed_hz = 10000
spi.mode = 0b10



num = 0



def getSolarPower():
    total = 0
    for port in ports:
        if GPIO.input(port["detect"]) == GPIO.LOW:
            x = [0x00,]
            GPIO.output(port["select"], GPIO.HIGH)
            x = [255^i for i in x] # NOT4
            x = spi.xfer2(x)
            x = [255^i for i in x] # NOT4
            print(x)
            print(x[0])
            if (x[0] == 101):
                x = spi.xfer2([0,])
                x = [255^i for i in x] # NOT4
                total += x[0]
            GPIO.output(port["select"], GPIO.LOW)
    total /= 0xFE
    print("solar power = " + str(total))
    return (total)
```

```
def getWindPower():
    total = 0
    for port in ports:
        if GPIO.input(port["detect"]) == GPIO.LOW:
            x = [0x00,]
            GPIO.output(port["select"], GPIO.HIGH)
            x = [255^i for i in x] # NOT4
            x = spi.xfer2(x)
            x = [255^i for i in x] # NOT4
            print(x)
            print(x[0])
            if (x[0] == 102):
                x = spi.xfer2([0,])
                x = [255^i for i in x] # NOT4
                total += x[0]
            GPIO.output(port["select"], GPIO.LOW)
    total /= 0xFE
    print("wind power = " + str(total))
    return (total)
```

```
def setBatterySOC(newstatus):
    SOC = int(newstatus * 255)
    total = 0
    for port in ports:
        if GPIO.input(port["detect"]) == GPIO.LOW:
            x = [0x01, ]
            GPIO.output(port["select"], GPIO.HIGH)
            x = [255 ^ i for i in x]  # NOT4
            x = spi.xfer2(x)
            x = [255 ^ i for i in x]  # NOT4
            print(x)
            print(x[0])
            if (x[0] == 105):
                x = [SOC, ]
                x = [255 ^ i for i in x]  # NOT4
                x = spi.xfer2(x)
                x = [255 ^ i for i in x]  # NOT4
                total += x[0]
                print("battery found")
            GPIO.output(port["select"], GPIO.LOW)
    total /= 0xFE
    print("new SOC = " + str(SOC))
```

Bibliography

[1] A. Fu, R. Saini, R. Koornneef, A. van der Meer AMD Peter Palensky, and M. Cvetkovic, "The Illuminator: An Open Source Energy System Integration Development Kit."

[2] H. Hodson, "Electric grids, The ultimate supply chains," *The Economist, Technology Quarterly*, Apr. 2023.

[3] *Grid stability, Equilibrium between production and consumption*. [Online]. Available: `https://www.swissgrid.ch/en/home/operation/regulation/grid-stability.html` (visited on 03/14/2023).

[4] (Mar. 2018), [Online]. Available: `https://www.drax.com/electrification/electricity-causing-clocks-europe-run-slowly/` (visited on 06/14/2023).

[5] *Renewable energy science education kit*, Manual found as part of kit created by Horizon Education, Horizon Fuel Cell Technologies. [Online]. Available: `https://www.horizoneducational.com/renewable-energy-science-kit/p1218`.

[6] Kristiel, *Tools for teaching renewable energy*, Oct. 2022. [Online]. Available: `https://blog.studica.com/teach-renewable-energy` (visited on 06/14/2023).

[7] *Tennet power flow simulator*. [Online]. Available: `https://netztransparenz.tennet.eu/fileadmin/user_upload/Our_Key_Tasks/Innovations/loadflow/index.html`.

[8] *Power the grid*, video game demo where you need to balance supply with demand. [Online]. Available: `https://claudioa.itch.io/power-the-grid`.

[9] *Embedded systems - common protocols*. [Online]. Available: `https://en.wikibooks.org/wiki/Embedded_Systems/Common_Protocols`.

[10] S. Butler. "What is gpio, and what can you use it for?" (Apr. 2022), [Online]. Available: `https://www.howtogeek.com/787928/what-is-gpio/`.

[11] S. O. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, Mar. 1997. DOI: `10.17487/RFC2119`. [Online]. Available: `https://www.rfc-editor.org/info/rfc2119`.

[12] Wikipedia contributors, *Usb — Wikipedia, the free encyclopedia*, 2023. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=USB` (visited on 05/01/2023).

[13] E. Peña and M. G. Legaspi, "Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter," *Analog Dialogue*, vol. 54, no. 4, pp. 51–55, Dec. 2020. [Online]. Available: `https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html`.

[14] Wikipedia contributors, *Asynchronous serial communication — Wikipedia, the free encyclopedia*, [Online; accessed 15-June-2023], 2023. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Asynchronous_serial_communication`.

[15] *The raspberry pi gpio pinout guide*, interactive pinout guide for the Raspberry Pi GPIO port. [Online]. Available: `https://pinout.xyz/`.

[16] "Tweedraads-bussysteem met een kloklijndraad en een datalijndraad voor het onderling verbinden van een aantal stations," NL-8005976-A, Oct. 1980.

[17] B. J. and S. Hymel, *Raspberry pi spi and i2c tutorial*. [Online]. Available: `https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial/all` (visited on 04/18/2023).

[18] P. Dhaker, "Introduction to spi interface," *Analog Dialogue*, vol. 52, no. 3, pp. 49–53, Sep. 2018. [Online]. Available: `https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html`.

[19] M. Hughes. "Back to basics: Spi (serial peripheral interface)." (Feb. 2017), [Online]. Available: `https://www.allaboutcircuits.com/technical-articles/spi-serial-peripheral-interface/`.

[20] "Spi bus." (Jul. 2016), [Online]. Available: `https://www.mikroe.com/blog/spi-bus`.

[21] *Enc28j60 data sheet*, Microchip, 2008. [Online]. Available: `https://www.microchip.com/en-us/product/ENC28J60`.

[22] *Rpi tutorial eghs:gpio protection circuits*. [Online]. Available: https://elinux.org/RPi_Tutorial_EGHS:GPIO_Protection_Circuits (visited on 04/02/2023).

[23] D. Ashton. "The polyfuse: Set-and-forget protection for circuits." (Aug. 2022), [Online]. Available: https://www.elektormagazine.com/articles/polyfuse.

[24] R. Panguloor, *Basics of efuses*, Texas Instruments, Dec. 2016. [Online]. Available: https://www.ti.com/lit/an/slva862a/slva862a.pdf.

[25] S. Cording. "The modern fuse: The classic fuse get a modern upgrade." (Aug. 2022), [Online]. Available: https://www.elektormagazine.com/articles/polyfuse.

[26] E. Seale, *Power switching circuits*, The "Zero Volt Diode", Oct. 2018. [Online]. Available: http://solarbotics.net/library/circuits/misc_switching.html (visited on 05/01/2023).

[27] *Atmega8(l) - complete datasheet*, Atmel, Feb. 2013. [Online]. Available: https://www.microchip.com/en-us/product/ATmega8.

[28] G. van Loo, *Gpio pads control*, explanation of the BCM2835 drive strength setting. [Online]. Available: https://matt.ucc.asn.au/mirror/electron/GPIO-Pads-Control2.pdf.

[29] G. W. Green, M. R. Arcoleo, and P. Sevalia, "Voltage level translator circuit," US-5691654-A, Dec. 1995. [Online]. Available: https://image-ppubs.uspto.gov/dirsearch-public/print/downloadPdf/5691654.

[30] M. Cook, *Raspberry pi breakout board*. [Online]. Available: http://www.thebox.myzen.co.uk/Raspberry/Breakout.html (visited on 05/03/2023).

[31] S. Curtis and D. Moon, *A guide to voltage translation with txb-type translators*, Texas Instruments, Mar. 2010. [Online]. Available: https://www.ti.com/lit/an/scea043/scea043.pdf.

[32] R. H. Adlhoch, "Cmos level shifter," US-4150308-A, Apr. 1979. [Online]. Available: https://image-ppubs.uspto.gov/dirsearch-public/print/downloadPdf/4150308.

[33] I. G. Akmenkalns, Endicott, R. J. Wilfinger, Poughkeepsie, and A. D. Wilson, "Four terminal electro-optical logic device," US-3417249-A, Dec. 1968. [Online]. Available: https://image-ppubs.uspto.gov/dirsearch-public/print/downloadPdf/3417249.

[34] R. F. Graf, *Modern dictionary of electronics*, en, 7th ed. London, England: Newnes, Aug. 1999.

[35] A. Montange, *Structured Electronics Design*, en, 2.3. Mar. 2023.

[36] M. K. Saini, "Difference between photodiode and light dependent resistor (ldr)," 2022. [Online]. Available: https://www.tutorialspoint.com/difference-between-photodiode-and-light-dependent-resistor-ldr.

[37] P. Yadav, "Ldr vs photodiode: Difference and comparison," 2023. [Online]. Available: https://askanydifference.com/difference-between-ldr-and-photodiode/.

[38] (), [Online]. Available: https://nl.farnell.com/pro-elec/pel00884/solar-motor-low-intertia/dp/3383479.

[39] datasheet for the 6P6C pcb connector, Jun. 2021. [Online]. Available: https://www.we-online.com/components/products/datasheet/615006138521.pdf.

[40] *Add-on boards and hats*, official guide for designing new hardware. [Online]. Available: https://github.com/raspberrypi/hats.

[41] (), [Online]. Available: https://www.instructables.com/USBASP-Installation-in-Windows-10/.

[42] Wikipedia contributors, *Led circuit*, 2023. [Online]. Available: https://en.wikipedia.org/wiki/LED_circuit.

[43] (), [Online]. Available: https://www.sparkfun.com/products/9592.

[44] (), [Online]. Available: https://forum.arduino.cc/t/pwm-didnt-work-on-atmega8/248564.