Network Embedded Systems [VU]
Semster Project - Protocol

Vienna University of Technology
May 30, 2017

Helmut Bergmann, Matr.Nr. 0325535

0325535@student.tuwien.ac.at

Bernhard Fritz, Matr.Nr. 0828317

0828317@student.tuwien.ac.at

Marko Stanisic, Matr.Nr. 0325230

0325230@student.tuwien.ac.at

Everyday life brings many repetitive tasks. As technology increases also more complicated tasks can be automatised, relieving humans from repetitive tasks. In our project we want to develop a system that handles the watering of plants in order to decrease the number of routine tasks done in a household and also to reliably continue watering even if no person is at home for a longer period of time.

1 Requirements

The system should substitute the need for human interaction. Therefore it has to sense as much information relevant to the task of watering as possible and it needs the possibility of adding water to the flower pots. Furthermore it should be reliable, so it doesn't have to be supervised.

- **All data containing information vital for the watering task has to be tracked.** This contains ambient temperature, ambient humidity, soil moisture and brightness. The data has to be logged in a database.
- **Reliable flower watering through pumps or valves.** Based on the sensor data the system has to make decision when to water the plants and for how long. It is critical that this part is fail safe. If there are a failures of any components of the system, the water supply has to be automatically cut off to avoid damage.
- **Any influence to the flower or surrounding through the system has to be avoided.** This in particular concerns the choice of sensors, pipes, pumps and water tanks to not pollute the soil, keep the noise level low, being an efficient solution and not disturbing other electronic devices nearby.
- **The hardware solution should be visually appealing.** Since the system should be usable in living areas, it has to be as unobtrusive as possible. As a result the data communication should be wireless and the sensor nodes run on battery.
- **Battery powered low power sensor nodes.** The sensor nodes should be powered by batteries, not the power grid. However the batteries also should last long time to decrease the maintenance work, what means the power supply is not trivial and needs to be paid attention to while developing the sensor node.

2 Design

2.1 Sensor Node

The sensor node provides data about the current state of the flower environment to the controller. It has been developed as a low power device which guarantees long battery duration. Arduino has been chosen as platform for this node because of its powerful libraries and support. Also the microcontroller has to offer sufficient ADCs and GPIO pins to access all required modules such as communication chip, real time clock and sensors. Other platforms such as XBee could not be used, as they do not directly support protocols such as TWI. Also they are expensive. In particular we chose the Arduino Mini 3.3V 8MHz board, as it is the most basic board and therefore offering the lowest power consumption. Further details about minimizing the power consumption are explained in 2.1.1.

The sensors used are:

- **DHT11 - Temperature and humidity sensor** The datasheet claims this sensor reaches a repeatability of $\pm 0.2C$ for the temperature and an accuracy of $\pm 5\%$ for humidity. It seems to be a widely used sensor for arduino projects, however in operation its humidity accuracy was far worse than promised by the manufacturer and a cumbersome calibration of the sensor is needed. Further research showed that it is not always a reliable humidity sensor and probably should be replaced by the more expensive DHT22 or other alternatives. We kept the sensor in the setup, however no decision has been based on its data.

- **Moisture sensor** We tried two moisture sensors. One resistance based sensor and one capacitive sensor. The electrodes used for the resistance sensors interact with the water in the soil if a current flows. This pollutes the soil and destroys the sensor over time. Making just sporadic measurements postpones this effect, however they cannot be avoided. Capacitive sensors do not interact with the soil as they measure the humidity through the capacity of the soil, not its resistance. So no current flows through the soil and there are no metallic or corrosive electrodes.

  The resistance based sensor consists of two parts, the resistance probe (YL-69) and a control board (YL-38). The probe will be put into the measured soil, while the board should stay dry. It provides an analog signal. The ATMega328 has a 10 bit ADC, so the result read from the ADC is an integer between 0 and 1023.

  The test showed that in a dry state the sensor shows 1023. Connecting the electrodes with a finger leads to a value around 1000.
  Also in dry soil the readout is 1023, but as we pour water into the pot, it continuously decreases down to a few hundred. (I did not want to drown the flower, but I

assume it goes to 0 if we put the probe into water...)

We might have to calibrate the moisture levels for different flower or soil types, but in general the sensor seems to do the job very well and also is usable with a XBee board because of its analog output. The only downside is that it's written in forums that after some weeks this type of sensor will be corroded, so we cleaned it as well as possible after the test. For developing and testing it is fine, in a real environment one might want to use another sensor like a capacitive one or with other electrode materials. Also turning off the sensor when not using it is a good approach to avoid this problem.

The capacitive sensor used is the SEN0193. It is handled the same way as the YL-69 sensor.

- The last sensor to test was a photoresistor which is a semiconductor that reacts to light. It was also provided in the Arduino kit. The manual suggested the range of the photoresistor is between 500Ω (bright) and 50$k\Omega$ (dark). Further it suggested to use a $1k\Omega$ resistor for the voltage divider. An arduino tutorial [**?**] suggested 10 $k\Omega$. So we decided to measure the actual resistor values and use the formula from the lecture to calculate the matching resistor of the voltage divider.

  The measurements showed that the actual photoresistor range is 500Ω to $1.7M\Omega$.

$$R_1 = \sqrt{R_{max} * R_{max}} = \sqrt{500 * 1700000} = \sqrt{850000000} = 29k\Omega \qquad (1)$$

  The closest resistor available to us was $26k\Omega$.
  These values differ to those suggested by the tutorials, but since also the photoresistor has different values, our choice seems feasible. Also we are convinced a higher resistor will not harm the components in contrast to a low resistor.
  So we connected the chosen pulldown resistor between ground and the A1 pin of the Arduino Nano. To complete the voltage divider the photoresistor was connected between VCC and the A1 pin.

  Sketches again are provided by online tutorials and the Arduino kit producer. Although it is very similar to the one for the moisture sensor, just another pin is used.

  The test worked very well, close to a light bulb we get values over 950 and if we cover the photoresistor it drops to under 30. That confirmed our choice of the resistor.

  We can conclude that this sensor is easy to work with and compatible with XBee, as no special protocols are required.

2.1.1 Low Power Capability

As mentioned before, the Arduino Mini 3.3V 8MHz is the basis of our sensor node platform. However the Arduino uses about 6 mA in standby. With this current consumption a set of batteries (assuming a capacity of 2000mAh) would be empty within 14 days. In a network consisting of 7 sensor nodes, every second day batteries would be needed to be changed. This motivates to look into further power saving techniques for the Arduino.

2.1.2 Power supply

One of the obvious questions is the supply of the operating voltage. It has been looked into the following options:

- **Using the Arduino voltage regulator** The Arduino board can be powered directly from the battery and generate its own supply voltage. However this regulator is very inefficient.
- **Using an external voltage regulator** A more efficient external regulator can be used instead of the onboard regulator. The original regulator will still consume a small current, even if not used. Therefore it should be removed in this case. The external regulator will require a supply voltage of typically 100-200mV above the output voltage. The regulator tested by us, MCP1700-3302E, needs 3.5V, so in case of three batteries this is 1.17V per battery.
  During the discharge process of a battery its output voltage drops. Once the voltage drops under the required supply voltage of a voltage regulator, the output voltage of the regulator will not match 3.3V anymore.
  Figure 1 illustrates the battery voltage of a AA NiMH battery over different loads. It is important to notice that with a low load the ampere hours obtained from the battery increases. So decreasing the current used by the system by half results in more than just doubled battery life. In our application the load is less than 1mA with some 15mA pulses during data transmission. This is even far less than the uppermost line indicates. It can be concluded that the energy remaining in the battery once its voltage drops below 1.17V is less than 400mA.
- **Using a buck-boost converter** A conventional voltage regulator needs a supply voltage of at least little over 3.3V to provide a 3.3V output. A buck-boost converter like the Pololu reg 12a which we tested, can convert voltages above 0.5V to constant 3.3V.
  This is beneficial because of the dropping battery voltage during the discharge process. A buck-boost converter has a lower required supply voltage (0.5V) and therefore gets nearly all energy out of the battery. However it turned out that for very low

currents, like in our sensor node, the power consumed by the Pololu is much higher than for the linear voltage regulator. This diminishes the advantage of the Pololu and even turns it into the worse choice in terms of energy efficiency. Regarding costs the linear power regulator beats the pololu by an order of magnitude.

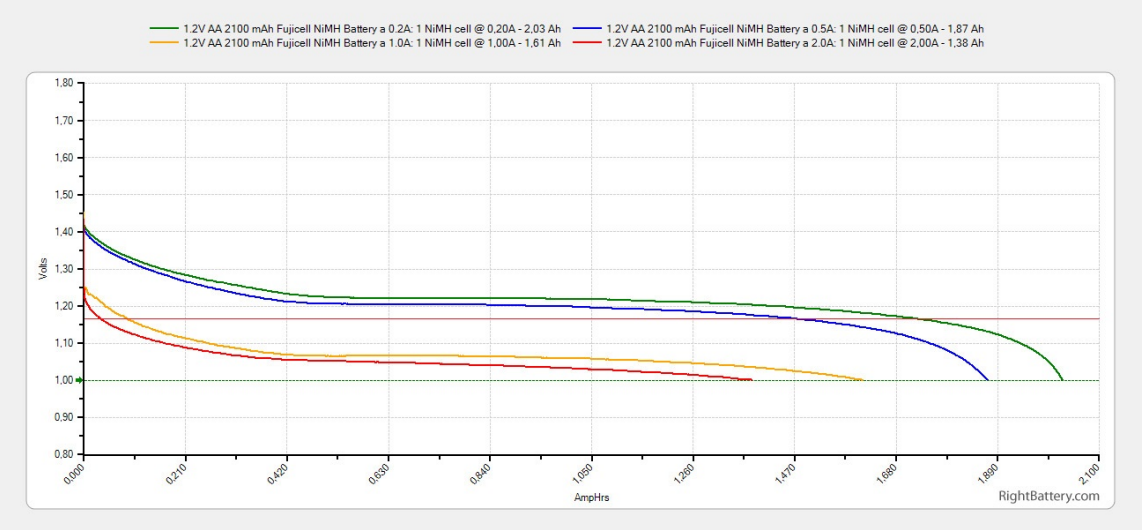

Figure 1: Discharge curves over different currents

We tried all solutions in our tests, however for the final solution we recommend the linear voltage regulator with adequate batteries.

2.1.3 Power consumption optimizations

To make the batteries last as long as possible, the node should use as little power as possible.
The biggest power safer is to put the microcontroller and all peripheral devices into sleep mode as much as possible.
This reduces the power of the system to around 1.5mA and gives 8 weeks of battery life per node. This is a big improvement, however far away from a convenient solution and the technical limits.
One further optimization is to remove the power LED used to indicate the Arduino is running, as there is no need for it and the LED uses a lot of power. The LED can be removed by adding tin to connect both legs, heating them up uniformly and pulling away the LED with tweezers. This decreases the consumption to about 0.26 mA (45 weeks).
Disabling TWI before sleep saved another 170µA.
Removing the parasiting onboard voltage regulator saved further 75µA. However desoldering of medium sized SMD components requires special equipment, skills or patience.

The resulting consumption for the design including all final components and using the external voltage regulator is 15µA. (4.5V ==> 67.5µW) Using the Pololu the same design needs 90µA (4.5V ==> 405µW). It can be concluded that in this situation an appropriate voltage regulator is more efficient than a Pololu. Also the voltage divider is a far cheaper solution.
It must be mentioned that three 1.5V batteries are assumed in the calculations, resulting in 4.5V total. Using rechargeable batteries with 1.2V each, the resulting total 3.6V are just slightly above the threshold for the voltage regulator to operate. In this case the Pololu would have the better performance. For operation with the voltage regulator 3x1.5V batteries or 4x1.2V rechargeable batteries are suggested.

2.2 Controller

The controller keeps track of all nodes and handles their registration. It executes the watering policy that the user has selected for each flower. For this task it wirelessly communicates with the sensor node situated in the flower pot and the pump node situated next to the water tank. The sensor node provides the controller with the most recent measurement values, whereas the pump node executes the pump commands received from the controller.

2.2.1 State diagram

States:

- Check pending tasks

  This is the begin of the loop() Function. The controller will always come back to this state to check which tasks are pending. The following actions are possible:

  - A NRF24 message is arrived, process this message.
  - Also if an interactive command has arrived via the IOT framework, this command has to be processed. (Typically sending an instruction to a Motor Node)
  - If, according to the watering schedule, a pump node should take an action now, the controller goes to the "Send Watering Command to Motor Node" state
  - If a Motor Node did not respond as expected, an error handling sequence should be started (writing to log and if needed respond to the IOT framework)

- Send Watering Command to Motor Node Whenever the controller decides a pump node should start to pump, this function is called. It prepares and sends a message to concerned the pump node. It also writes the message content to a log file.

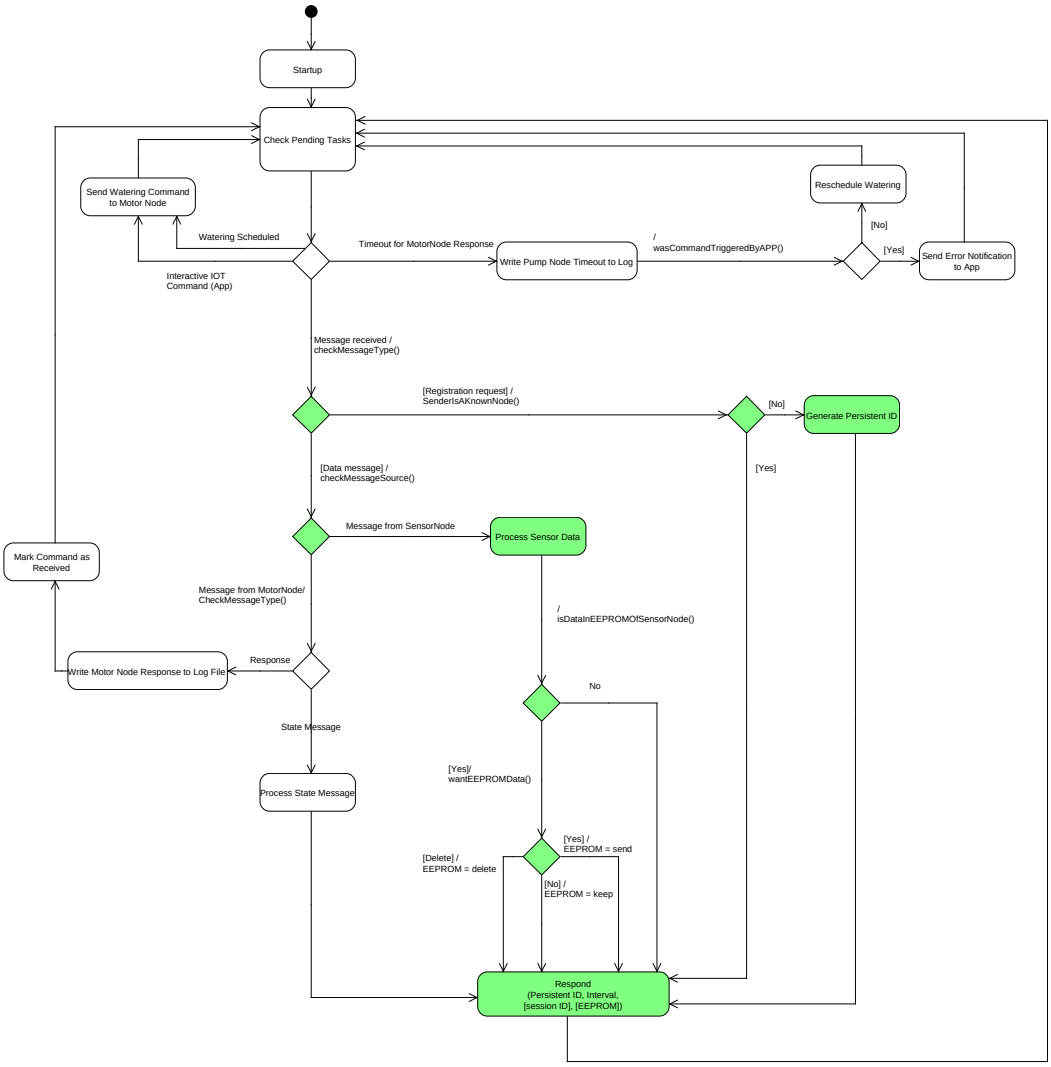

Figure 2: Overview of the chosen setup.

- Write Pump Node Timeout to Log If a pump node does not reply within the specified timeout duration, an entry into the log file has to be written. In case the command of the concerned message has been triggered by the IOT framework, the controller will move to the “Send Error Notification to App” state. Otherwise it enters the “Reschedule Watering” state.

- Send Error Notification to App If a IOT command could not be successfully executed, the IOT framework has to be notified about the problem. Afterwards the controller goes back to checking for pending tasks.

- Reschedule Watering If a scheduled watering task failed, the controller needs to reschedule this watering task. There are many possible algorithms for this such as:

  - Skip this watering task and proceed as usual
  - Retry the watering task a certain number of times. If it fails, proceed as usual
  - Reschedule the watering of this pump node in a given fashion.

  For a first version it is sufficient to skip the watering, if it failed.

- Write Motor Node Response to Log File If a motor node responds to a control command, this response has to be logged to a log file. Afterwards the controller processes the response.
- Mark Command as Read The watering command is marked in the controller internal command list as "read by the node". Otherwise a timeout would occur.
- Process State Message As the pump node finished to pump, it will send a state message to the controller indicating the end of the pump sequence. This message can be responded to by the controller, but it doesn't have to be.
- Process Sensor Data When the controller receives data from a sensor node, it stores the data. Later this data will be used for the watering algorithm and statistical analysis. Afterwards it will check whether the sensor node has data stored in its EEPROM. This will be the case if the controller was unavailable to the sensor node for some time. If there is EEPROM data the controller can choose whether it wants the data to be transmitted now, later or not at all. If the latter is the case, the node will delete its sensor data. If the controller has currently time to receive the EEPROM data, it should request it. Otherwise the data might get lost. If there is a lot of traffic scheduled in the next slots, the controller can postpone the EEPROM data transmission. In this case the next sensor node transmission should be scheduled in a slot that is followed by empty slots. This way it is guaranteed that there is enough time to transmit all EEPROM data.
- Respond In the respond state a response is prepared and sent to the node from which the last message has been received. Its content is based on the previous states.

2.3 Pump Node

2.3.1 Fall-back Mode

To guarantee a dependable solution resistant to external factors such as a failing internet connection or wireless jamming attacks, a fall-back mode has been implemented to water the flowers reliably, even if the control node is unavailable.

For this purpose the pump node uses all incoming messages, also those addressed to other nodes, to detect the availability of the controller. If the pump node does not

receive any messages from the controller for a specified timeout_duration, it assumes the controller is offline and goes into a fall-back mode.

The parameters for the fall-back mode are:

- timeout_duration
- enable
- time
- watering_duration

These parameters can be set at the IOT front-end-interface and get transmitted to the pump nodes by the controller at the pump node registration and with every pump request. (so in case they get changed the pump node stays up to date) In case the fall-back is disabled, no action takes place if the controller is unavailable. In case the fall-back is enabled and the controller is unavailable, daily at the specified time the pump node pumps for the specified watering_duration.

This mechanism does not guarantee a perfect watering, however it prevents the plants from drying.

To make the fall-back visible to the monitor node, a notification message is broadcasted via the wireless network containing the timestamp, pump duration and last timestamp from a controller message.

2.4 Protocol

For registering nodes and perform data exchanges a protocol has been developed. The developed library provides access to the protocol registers to use the functionality of the protocol.

Status register in the node –> controller message:

Bit	name	description	0 - meaning	1 - meaning
0	RTC_RUNNING_BIT	RTC is paired on the node	no RTC	RTC paired
1	MSG_TYPE_BIT	type of message	registration request	data
2	NEW_NODE_BIT	new or known node	known node	new node
3	EEPROM_DATA_AVAILABLE	is data in the EEPROM	no data	data available
4	EEPROM_DATA_PACKED	kind of data	live data	EEPROM data
5	EEPROM_DATA_LAST	more EEPROM data pending	more data	not more data
6	NODE_TYPE	type of node	sensor node	pump node

Bit	name	description	0 - meaning	1 - meaning
0	REGISTER_ACK_BIT	registration acknowledge	no reg	registration
1	FETCH_EEPROM_DATA1			
2	FETCH_EEPROM_DATA2			
3	ID_INEXISTENT	id invalid	id valid	id invalid

Status register in the controller –> node message:

Further the combination “00” in the FETCH_EEPROM_DATA1/2 bits indicates the EEPROM data shouldn’t be transmitted now, “01” means the sensor node should send the data now and “10” means the node should delete the stored EEPROM data.