Amsterdam University of applied sciences

Platform, Sensors and Communication

Clock with weather forecast

Author:
Daniel Gross

Student Nr.:
500792752

April 8, 2018

Hogeschool van Amsterdam

Contents

1 Introduction

Part of the course Platform, Sensors and Communication was to build a small Internet of Things (IoT) device in about seven weeks. This report covers the process of building the IoT device, all the things that were learned and problems faced. This report should make it possible for anyone to recreate the project.

2 The Idea

Getting an idea for this small project was kind of a hard part. I spent many hours on figuring out what I probably can build. As I just moved here I started to think about what is missing in my room that I have in my room in Austria and use frequently. Thinking of it I realised that I have no clock next to my bed. I started thing of how to make a clock a IoT device. Most of the clocks you have to set on your own. So when you change your location you manually have to change the time on the clock. I wanted to make that better in the way that the clock adjusts it self by using GPS to get the location and time. I thought it would also be nice to see the weather forecast for the current day and as I already have the position this should not be a big problem. I also wanted it to connect to the internet over the "The Things Network"[1] so there is no need for a WiFi network.

3 Sketch of setup

3.1 Initial setup

Figure 1 shows a sketch of how I imagined that the whole setup could look like. I wanted to use the "TTGO LORA32 868 / 915MHz SX1276 ESP32"[8] as the main module as it could connect to a LoRa network and also has display on it. To get the current position I wanted to take the "GY NEO-6M V2 GPS" GPS module. So the Arduino module connects to The Things Network[1] and sends the GPS coordinates to my web-service over the HTTP integration[2] function of The Things Network. The web-service would then get the weather information from the Open Weather Map[3] and sends the information back to the arduino.

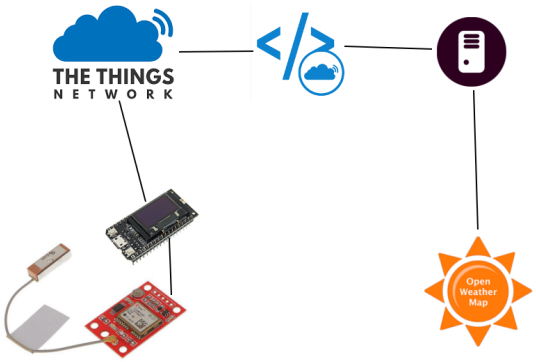

Figure 1: Initial setup of the idea

3.2 Final setup

Figure 2 shows the final setup I came up with. The main module of the device is the "Maker Hawk ESP32 LoRa" which can connect to the LoRa network. It has attached a display and a GPS module. The arduino sends data over The Things Network to my web-service. The data send contains the longitude and latitude. By this information the web-service gets the weather forecast from weatherbit[5] and the time offset from timezonedb[4]. This information is than send back to the arduino.

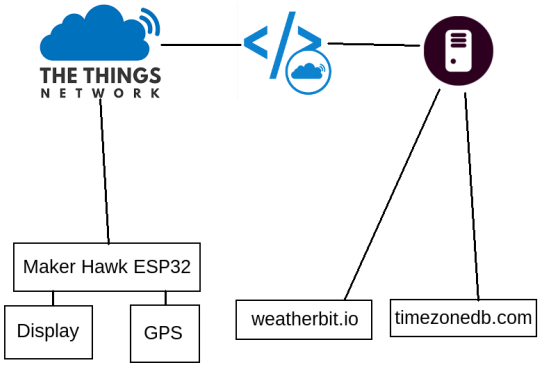

Figure 2: Final setup of the idea

4 Hardware

4.1 Arduino

The arduino I have chosen is the Maker Hawk ESP32[6]. It has a build in LoRa module, build in 128x64 OLED display and the possibility to be run with a 3.7V LiPo battery. I chose this arduino because of its delivery time as I needed it very fast to start building. It also has a small form factor which can be very good if you want to build a case for the whole device.

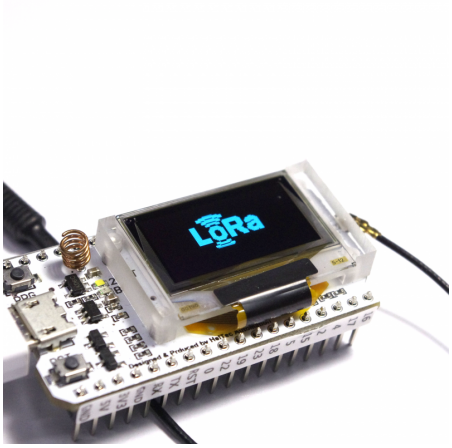

Figure 3: Maker Hawk ESP32 with LoRa

An other option would have been the "TTGO LoRa". It is available with a OLED display[8] and without[7]. It also has a build in LoRa module like the "Maker Hawk" and also the option for running it with a LiPo battery. But I did not chose it because of the delivery time. But this project still should be possible to realise with it if you want to use this arduino.

4.2 GPS Module

To determine the current location I used the GY-NEO6MV2[9] GPS module. It works with a supply voltage from 3V to 5V. The communication with the module is done by serial communication. I found out that all other GPS modules use the same chip so I did not look for any other module.

Figure 4: NEO-6M V2 module

4.3 OLED Display

As the build in display of the MakerHawk broke at one point I had to get a replacement for it. I chose a 128x64 OLED display[10] with I2C communication because the switch from the internal display to the external display could be done by just connecting the two pins that were internal routed to the display to the external display.

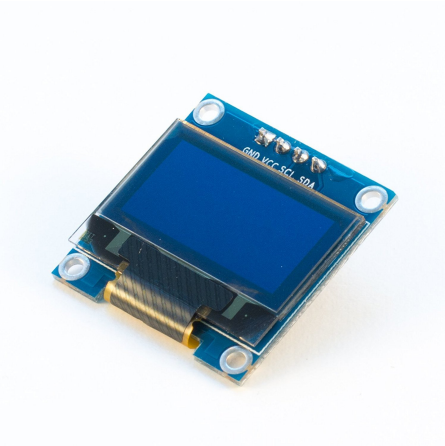

Figure 5: 128x64 OLED display

5 Connecting the hardware

Figure 6 shows the wiring I have chosen. The connection between OLED and Arduino is SCL(OLED) to SCL(Arduino - PIN 15) and SDA(OLED) to SDA(Arduino - PIN 4). For the GPS module the connected PINs are RX(GPS) to TX(Arduino - PIN 17) and RX(Arduino - PIN 16) to TX(GPS). The GPS module uses the 5V power-out of the Arduino and the OLED display the 3.3V power-out of the Arduino.

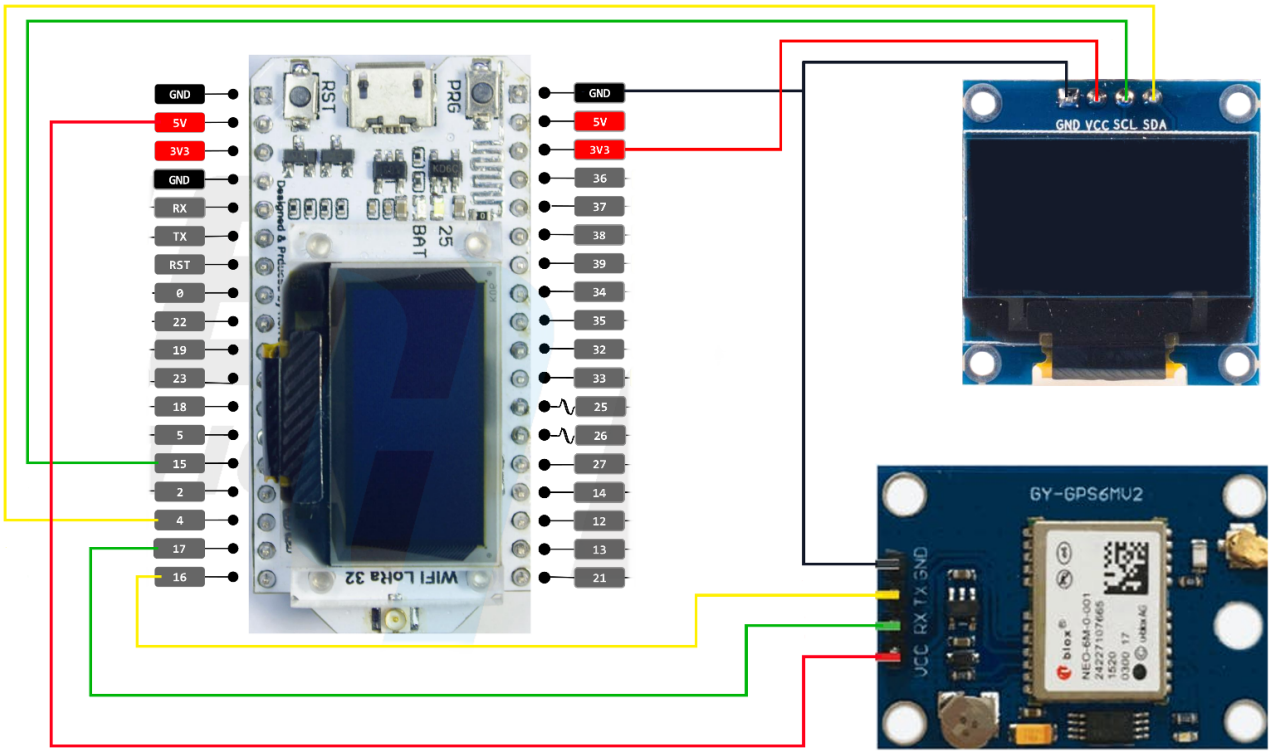

Figure 6: Wiring diagram

I used a breadboard to connect everything together. My setup is shown in figure 7.

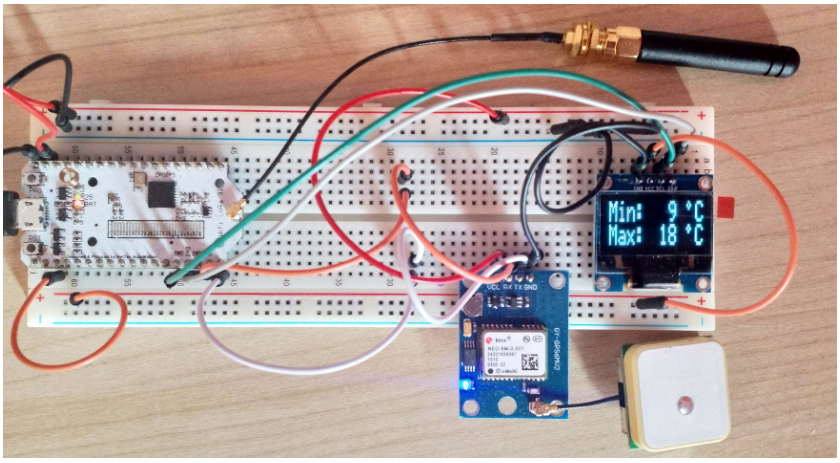

Figure 7: Development setup on a breadboard

6 Libraries

6.1 Maker Hawk ESP32 LoRa

For the Maker Hawk I use the library arduino-esp32[11] from the company espressif[13]. On the github repository is a manual how to install the library and by following these steps the setup was done very quickly.

6.2 LoRaWAN

To connect to the "The Things Network"[1] I used the library Arduino-LMIC[16]. On the github repository is a very good explanation how to install the library. The library also comes with example codes which makes it easy to understand how to use it.

6.3 NEO-6M V2

For the GPS module I use the TinyGPS++[14] library. An other library that I had a look at was NeoGPS[15]. I chose the TinyGPS++ library because the usage of it is much easier. Adding the library is done in the arduino studio in the menu "Sketch→Include Library→.ZIP Library".

6.4 128x62 OLED display

To the control the display I used the library "OLED"[12]. The library was written by Stefan Frings and consists of two classes. The library is very easy to use and also allows to draw images on to the display which was very important for me as I want to show weather icons. I had to modify the library but this will be covered in the section 7.3.2 on the page 15.

7 Arduino Software

The source code is available on the git repository[26].

7.1 Clock

The clock is initialized with the information from the GPS. The GPS not only contains position data, it also contains the UTC time. I use this information to set the initial time of the clock. From the web-service I get, beside the weather information, the offset to the UTC time. This offset is then added to the clock so it shows the correct time. To determine if a minute has passed the *millis()* function is used. This function returns the milliseconds passed since the program has started. Source Code 1 shows how the function is implemented. The global variable *timeSynchronised* is set to *false* when the GPS data changes and helps to keep the time synchronous.

```
void updateTime(){
  int currentTime = millis();
  //Try to keep seconds as synchronised as possible
  if (gps.location.isValid() && !timeSynchronised){
    int currentSecond = (currentTime - lastTime);
    if(currentSecond < (timeSecond *1000)){
    //move the last time check backwards so the time
    //will be updated earlier
      lastTime -= (timeSecond *1000) - currentSecond;
    }
    timeSynchronised = true;
  }

  if((currentTime - lastTime) >= 60000){
    //Check if we are currently in minute 59 of the hour
    if(timeMinute == 59){
      //If we are in hour 23 then set it back to 0 otherwise
      //increase
      if(timeHour == 23){
        timeHour = 0;
      } else {
        timeHour += 1;
      }
      timeMinute = 0;
    } else {
      timeMinute += 1;
    }
    lastTime = currentTime;
  }
}
```

Source Code 1: Function to update time

7.2 Switching screens

As I have three screens to show I had to implement something that switches between the screen. I also wanted the screens to be shown a certain time. Therefore I first used delays which caused a asynchronous clock. After some research I figured out that I have to use the *millis()* function and implement the switching with it. The current version now uses a sort of state-machine to switch the screens which is shown in Source Code 2. Every second a step is called and the step then checks how long it has already been shown and if necessary switches the value of *currentStep* to the next step. This all is done in the main loop of the arduino. The three screen states are shown in figure 8, 9 and 10.

```
//Switch to the different screens
    int currentTime = millis();
    if((currentTime - lastTimeLoop) >= 1000){
      lastTimeLoop = currentTime;
      switch(currentStep){
        case 0:
          timeStep();
          break;
        case 1:
          currentWeatherStep();
          break;
        case 2:
          currentWeatherMinMaxStep();
          break;
      }
    }
```

Source Code 2: Code to switch between the different screens

Figure 8: Screen with clock

Figure 9: Screen with weather icon, temperature and weather description

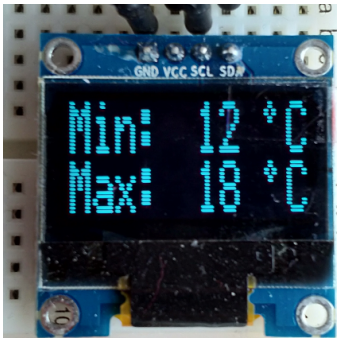

Figure 10: Screen with min and max temperature

7.3 Display

7.3.1 Communication

The communication with the display is done by I2C which is all handled by the library. I2C is a Bus system which was developed by Philips in the early 80s. It uses to lines to communicate. These lines are SDA(Serial Data) and SCL(Serial Clock). Figure 11 shows how the communication on I2C works. The start of a communication is initialised by the master by setting SDA from high to low while SCL is high. After that the SCL is switching between high and low and data is put on the SDA line. During the communication SDA is only allowed to change when SCL is low. On i2c-bus.org[19] a good explanation for the I2C protocol can be found.

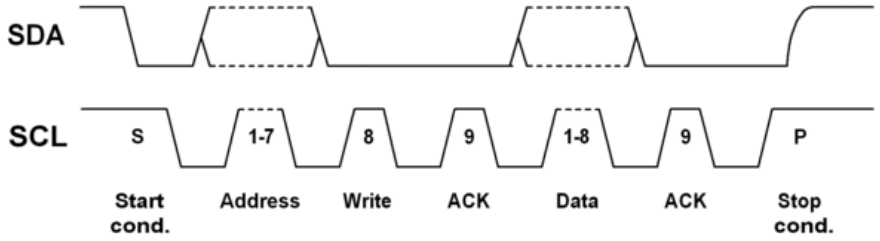

Figure 11: Communication sequence on I2C

To control the display it was only necessary to set the PINs correct and the address. The address of the display needs to be changed in the constructor for the class *OLED* in the file *oled.h* as shown in Source Code 3. The constructor in the header file also suggests to different addresses that should be tried. Setting the PINs is done in the initialisation as shown in Source Code 4.

```
/**
     * Constructor of the OLED class.
     * @param sda_pin Pin number of the SDA line
      (can be any Arduino I/O pin)
     * @param sda_pin Pin number of the SCL line
     (can be any Arduino I/O pin)
     * @param reset_pin Pin number for the /RST line,
     or use NO_RESET_PIN if the reset
     * signal is generated somewhere else
     * @param i2c_address The IC address is usually 0x3c or
     0x3D
     * @param width With of the display in pixels: 96 or 128
     * @param height Height of the display in pixels: 16, 32 or
     64
     * @param isSH1106 Must be true=SH1106 chip, false=SSD1306
     chip
     */
    OLED(uint8_t sda_pin, uint8_t scl_pin,
    uint8_t reset_pin=NO_RESET_PIN,
    uint8_t i2c_address=0x3C, uint_fast8_t width=128,
    uint_fast8_t height=32, bool isSH1106=false);
```

Source Code 3: Constructor of OLED class

```
//Configure the display
#define OLED_SDA 4
#define OLED_SCL 15
#define OLED_RST 23
OLED display=OLED(OLED_SDA,OLED_SCL);
```

Source Code 4: Initialisation of the display

7.3.2 Displaying weather icons

The library for the display has a function to draw images. The images has to be passed as a byte array. Therefore I downloaded all the weather icons from weatherbit, converted them to black and white icons and generated the byte array with the help of the website dot2pic[17]. For some reason the icons were not shown correct. As the byte array of the icons are a bit to read I created a simple sin-wave with dot2pic and let the arduino draw it to analyse it. As shown in figure 12 the sin-wave is not correct shown.

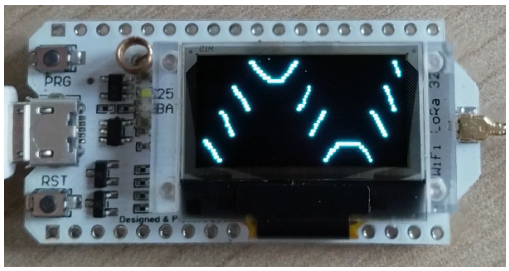

Figure 12: Image drawing problem

I figured out that the bit order for the pixels of one byte was wrong. To fix the problem I added some code to the function *draw_bytes* in the file *oled.cpp*. The part which fixed the problem is shown in Source Code 5.

```
            if(reverse){
              //Reverse the order
                uint8_t byteNew = 0x00;
                if((b & 0x01) == 0x01){
                    byteNew |= 0x80;
                }
                if((b & 0x02) == 0x02){
                    byteNew |= 0x40;
                }
                if((b & 0x04) == 0x04){
                    byteNew |= 0x20;
                }
                if((b & 0x08) == 0x08){
                    byteNew |= 0x10;
                }
                if((b & 0x10) == 0x10){
                    byteNew |= 0x08;
                }
                if((b & 0x20) == 0x20){
                    byteNew |= 0x04;
                }
                if((b & 0x40) == 0x40){
                    byteNew |= 0x02;
                }
                if((b & 0x80) == 0x80){
                    byteNew |= 0x01;
                }
                b = byteNew;
            }
```

Source Code 5: Code added to draw_bytes to fix image drawing issue

7.4 GPS

7.4.1 Communication

The communication with the GPS module is done with a serial connection. The GPS module sends single bytes and every set ends with a "new line" character. Figure 13 shows the data that is received from the GPS module. The data has the NMEA Message Structure which was developed by National Marine Electronics Association[18].

```
$GPRMC,133225.00,A,5220.99147,N,00455.23160,E,0.740,,070418,,,A*78
$GPVTG,,T,,M,0.740,N,1.371,K,A*24
$GPGGA,133225.00,5220.99147,N,00455.23160,E,1,04,8.53,27.9,M,45.9,M,,*63
$GPGSA,A,3,12,19,17,06,,,,,,,,,11.52,8.53,7.74*34
$GPGSV,3,1,10,02,22,108,20,03,03,358,18,06,23,067,26,12,77,058,33*78
$GPGSV,3,2,10,14,30,308,,17,06,037,30,19,20,043,27,22,03,338,19*78
$GPGSV,3,3,10,24,44,137,,25,56,262,17*7F
$GPGLL,5220.99147,N,00455.23160,E,133225.00,A,A*68
```

Figure 13: Raw data received from the GPS module

The arduino buffers the received data and the received data can than simply passed on to the TinyGPS++ library byte by byte. The library processes the information and makes it easy accessible by some functions.

7.4.2 Reading the data

Source Code 6 is a sample code to read the data from the serial communication. It is important to first check if data is available. If data is available it will be passed to the TinyGPS++ library to encode it. The library has functions to check if the data is valid and if it was update. The data received from the GPS module should only be used if both return *true*.

```
//Check if serial communication is functional
 while (Serial2.available() > 0) {
   chr = Serial2.read();

   gps.encode(chr);
   // check if location data is updated
   if (gps.location.isValid() && gps.location.isUpdated())
   {
     //gps.location.lat() - returns latitude
     //gps.location.lng() - returns longitude
     //gps.time.hour() - UTC hour
     //gps.time.minute() - UTC minute
   }
 }
```

Source Code 6: Code sample to read the data from the GPS module

7.5 LoRa

In the file *LoRaCom.h* are the static variables for APPEUI, DEVEUI and APPKEY located. This variables need to be set with the information from The Things Network. Note that the order of the bytes is in reverse compared to the array from The Things Network. In this file there is also the variable which sets the interval for sending payloads. The variable for the interval is *TX_INTERVAL*.

7.6 Payload

7.6.1 Data structure

As communication over the LoRa network is only possible with byte data I developed my on small and simple structure for the payload of the uplink(sending data) and downlink(receiving data).

Figure 14 shows the structure of the uplink data. The first byte is the action that has to be done with the data. This byte is *0x00* when the rest of the bytes

can be ignored and *0x01* when the data needs to be processed. After sending data we expect to get a downlink message but for this sending data is needed. In this case the first byte is set to *0x00*. To send the longitude and the latitude both float values are converted to integer values. This is done by dropping everything after the second decimal place and multiply by 100. So for example if we have a *latitude=4.501* this will result as *450* in the payload. Bit 15 is used to indicate if the value is negative or positive.

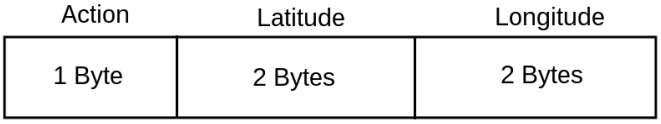

Figure 14: Payload format for the uplink

Figure 15 shows the structure of the downlink data. All the values are transmitted as integer values. The temperatures are originally float values and are converted the same way as in the uplink. Also the indication if the value is negative is handled the same way.

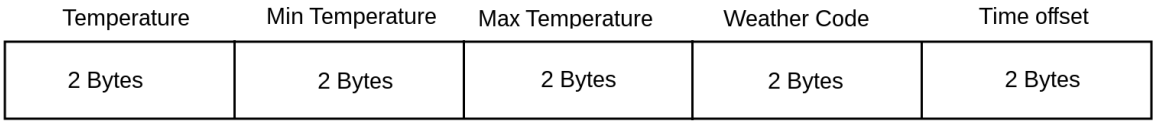

Figure 15: Payload format for the downlink

7.6.2 Encode and decode

Encoding and decoding is handled by four functions. The function shown in Source Code 7 in combination with function of Source Code 8 generate the an array of bytes that can be sent over the LoRa network. The array generated complies with the structure show in figure 14.

```
//Generates the payload for the request to
//the web service and sends the array
void sendRequestToWebService(){
  //Set to action byte
  //to 0x01 so it will be processed
  loraPayload[0] = 0x01;

  //Convert the latitued value to a byte array
  //and store it in the payload array
  uint8_t latitudeBytes[2];
  toBytes(latitude, latitudeBytes);
  loraPayload[1] = latitudeBytes[0];
  loraPayload[2] = latitudeBytes[1];

  //Convert the longitued value to a byte array
  //and store it in the payload array
  uint8_t longitudeBytes[2];
  toBytes(longitude, longitudeBytes);
  loraPayload[3] = *longitudeBytes;
  loraPayload[4] = *(longitudeBytes + 1);

  do_send(&sendjob);
}
```

Source Code 7: Code to encode the payload

```
void toBytes(int value, uint8_t bytes[]){
  bool negativeValue = false;

  //Check if value is negative and make it positive
  if(value < 0){
    negativeValue = true;
    value *= (-1);
  }
  //If the value is greater than 255 make a shift
  //bits to the right and store the bit 8 to 15 in byte[0]
  if(value > 255){
    bytes[0] = (value >> 8);
  }
  //Make all bits after bit 7 to 0 and connect it by
  //using the bit & operation
  bytes[1] = (((~(value >> 8)) << 8) | 255) & value;

  if(negativeValue){
    bytes[0] = bytes[0] | 128;
  }
}
```

Source Code 8: Function that helps converting to byte

The function shown in Source Code 9 in combination with function of Source Code 10 decode the received payload so the information can be further processed.

```
void processReceivedData(){
  currentTemp = (int)(byteToInteger(loraReceived, 0, 1) / 10);
  minTemp = (int)(byteToInteger(loraReceived, 2, 3) / 10);
  maxTemp = (int)(byteToInteger(loraReceived, 4, 5) / 10);
  int weatherCode = byteToInteger(loraReceived, 6, 7);
  int timeOffsetNew = byteToInteger(loraReceived, 8, 9);

  //Check if time needs to be set new
  if(timeOffset != timeOffsetNew){
    timeOffset = timeOffsetNew;
    setTime = true;
  }

  setWeatherIcon(weatherCode);
  setWeatherDescription(weatherCode);
  weatherAvailable = true;
}
```

Source Code 9: Code to decode the payload

```
int byteToInteger(uint8_t data[], int byte0, int byte1){
    //Check if value is negative --> Bit 8 is set if negative
    bool valueNegative = (data[byte0] >> 7) == 1;

    //Set Bit 4 of byte 0 to 0
    data[byte0] = ((data[byte0] << 1) & 255) >> 1;

    //Calculate float value
    int theValue = ((data[byte0] << 8) | data[byte1]);

    //Set algebraic sign
    if(valueNegative){
      theValue=theValue*(-1);
    }
    return theValue;
}
```

Source Code 10: Function that helps with converting from byte value to integer value

8 The Things Network

8.1 What is The Things Network

The Things Network is a free to use LoRa network. LoRa uses lower radio frequencies to transmit data which have a longer range. LoRa has a range from about 5 to 15km this depends on if the next gateway is *Line of Sight(LOS)* or *Non Line of Sight(NLOS)*. LOS means that there are no obstacles between the device and the gateway. NLOS is the case if there are buildings ore other obstacles between the device and the gateway which reduce the signal strength.

The LoRa network uses four different frequency ranges around the globe:

- EU: 863-870 MHz

- US 902-928 MHz
- Australia: 915-928 MHz
- China: 779-787 MHz and 470-510 MHz

A module only supports one frequency range which means that for example a module for Europe will not work in the US, Australia or China.

The data send is called *payload*. There is a page of The Things Network where they explain the limitations[20] of it. This should be kept in mind when developing a application that uses LoRa. On the forum[21] of The Things Network a user describes how to calculate the used bandwidth. He also says that the data that application packet size varies between 51 bytes for the slowest data rate, and 222 bytes for faster rates. A bigger data package means that you can sent less often.

8.2 HTTP integration

The Things Network has the option to integrate different services of them to a application. I used the HTTP integration[2] which makes it possible to forward information to a web-service. This is done by defining the URL and the HTTP method that should be used. I used the payload formats functionality explained in section 8.3 to convert the data before sending it to the web-service.

8.3 Payload formats

If you want to sent JSON data to the web-service there is a functionality called Payload formats which gives the opportunity to convert the byte data in a more readable JSON object. There is a very good video on youtube[22] which explains the use of it. I used the encode and decode functionality to convert from byte to JSON and back. Source Code 11 shows the decoding of the received payload to a JSON object.

```
function Decoder(bytes, port) {
  if(bytes[0]== 0x01){
    //Check if latetitude is negative --> Bit 8 is set if
    //negative
    var latNegative = (bytes[1] >>> 7) == 1;
    //Set Bit 4 of byte 0 to 0
    bytes[1] = ((bytes[1] << 1) & 255) >>> 1;
    //Calculate float value
    var lat = ((bytes[1] << 8) | bytes[2])/100;
    //Set algebraic sign
    if(latNegative){
      lat=lat*(-1);
    }
    //Check if Longitude is negative --> Bit 8 is set if
    //negative
    var lonNegative = (bytes[3] >>> 7) == 1;
    //Set Bit 4 of byte 2 to 0
    bytes[3] = ((bytes[3] << 1) & 255) >>> 1;
    //Calculate float value
    var lon = ((bytes[3] << 8) | bytes[4])/100;
    //Set algebraic sign
    if(lonNegative){
      lon=lon*(-1);
    }
    return {
      lat:lat,
      lon:lon
    };
  } else {
    return {};
  }
}
```

Source Code 11: Decode function to convert the byte data to a JSON object.

Source Code 12 shows the encoding from the JSON object to the byte data. This function uses a second function *toBytes* to convert the values, which is shown in Source Code 13.

```
function Encoder(object, port) {
    var bytes = [];
    var temp = object.temp * 10;
    var minTemp = object.minTemp * 10;
    var maxTemp = object.maxTemp * 10;
    var code = Number(object.weatherCode);
    var offset = Number(object.offset);

    var tempBytes = toBytes(temp);
    bytes[0] = tempBytes[0];
    bytes[1] = tempBytes[1];

    var minTempBytes = toBytes(minTemp);
    bytes[2] = minTempBytes[0];
    bytes[3] = minTempBytes[1];

    var maxTempBytes = toBytes(maxTemp);
    bytes[4] = maxTempBytes[0];
    bytes[5] = maxTempBytes[1];

    var codeBytes = toBytes(code);
    bytes[6] = codeBytes[0];
    bytes[7] = codeBytes[1];

    var offsetBytes = toBytes(offset);
    bytes[8] = offsetBytes[0];
    bytes[9] = offsetBytes[1];

    return bytes;
}
```

Source Code 12: Encode function to convert the JSON object to byte data for the downlink

```
//Converts integer value in a byte[]. Bit 8 of byte[0]
//is used as algebraic sign. Max value = 32767
function toBytes(value){
    var bytes = [0,0];
    var valueNegative = false;

    //Check if value is negative and make it positive
    if(value < 0){
        valueNegative = true;
        value *= (-1);
    }

    //If the value is greater than 255 make a shift
    //bits to the right and store the bit 8 to 15 in byte[0]
    if(value > 255){
        bytes[0] = (value >>> 8);
    }

    //Make all bits after bit 7 to 0 and connect it by
    //using the bit & operation
    bytes[1] = ((~(value >>> 8)) << 8) | 255) & value;

    //Marke the value as negative if needed
    if(valueNegative){
        bytes[0] = bytes[0] | 128;
    }

    return bytes;
}
```

Source Code 13: Function that helps converting during the decoding process

9 Web-service

To get the information from weatherbit and timezonedb I implemented a small web-service in Java. The web-service features a REST-API to communicate with it. The source code is available on the git repository[26].

9.1 How it works

The web-service has a REST-API method called *getWeatherArduino* in the class *RestAPI.java*. This method expects a JSON object in the Uplink format of the HTTP-integration[2] of The Things Network. In this JSON object is a field called *payload_fields* which the method is looking for. The format shown of the object in the field called *payload_fields* is shown in Source Code 14. Latitude and longitude are expected as a float value.

```
{
  "lat":40.0,
  "lon":9.01
}
```

Source Code 14: JSON object example for the REST-API

With this information then the method *getWeather* and *getUTCOffset* are called. The method *getWeather* gets the weather information from the weatherbit[5] API which is a free service. From timezonedb[4] the web-service gets the offset to the UTC time and is also a free service. Both need a registration and a API-key which needs to be added to the code. After all the information is collected a JSON object is created, Source Code 15 shows an example JSON object.

```
{
  "maxTemp": 18.3,
  "minTemp": 9.8,
  "offset": 2,
  "temp": 14.7,
  "weatherCode": "804",
  "weatherDescription": "Overcast clouds"
}
```

Source Code 15: Example of the JSON object with the weather information and time offset

To send this JSON object back to the arduino a call of the integration service has to be done. This is also explained on the HTTP-integration[2] website in the section "Downlink". All the information needed to create the downlink JSON object is in the received uplink JSON object contained.
I used the Java SDK[24] from The Things Network to send the data back. This library is very easy to use and the quick start guide[25] explains how to use it.

9.2 Running it

To run the web-service I used the IBM Cloud. The cloud is free to use with some limitations on the application size of the Java web-application and that it will be shout down if there is no development on the application. The service for the Java application is called *Liberty for Java*[23]. The documentation explains how it works and how the application can be pushed and run.

10 Problems

During the building and programming process I had some small problems. One I mentioned in the section 7.3.2 with the displaying of the images. But as there is access to the source code of the library it was easy to solve the problem. An other problem was that I sometimes found code examples that are not working due to missing information which libraries are needed. An other issue is the pore

reception of the GPS module. The GPS module only finds satellites when it is very close to a window but according to the internet this is related to the stock antenna and can be fixed with a better antenna.

11 Personal experience and what I have learned

This small project was a combination of brushing up knowledge from my previous school, learning new things and using knowledge from the previous years at my university in Austria. My personal biggest challenge was to get the system as whole to work. Controlling the display of the arduino or reading the GPS data was simple due to the help of the libraries. Implementing the web-service was simple as this was a part of my last years project at the university. Working with the LoRa network was something completely new. I had to come up with a way to transmit the data between arduino an web-service efficient and therefore understand how the LoRa works. I struggled a bit with the programming of the arduino as it has been a while since I programmed in C but after a few days of working on it I felt comfortable again.

References

[1] The Things Network
https://www.thethingsnetwork.org/

[2] The Things Network - HTTP integration
https://www.thethingsnetwork.org/docs/applications/http/

[3] Open Weather Map
https://openweathermap.org/

[4] Timezonedb
https://timezonedb.com/

[5] Weatherbit
https://www.weatherbit.io/

[6] MakerHawk ESP32 LoRa
http://heltec.diytrade.com/sdp/2044581/4/pd-6785993/12013554-0/
ESP32_Development_board_Lora_Transceiver_SX1278_43.html

[7] TTGO ESP32 LoRa
https://www.banggood.com/
Wemos-TTGO-ESP32-SX1276-LoRa-868-915MHz-Bluetooth-WIFI-Lora-Internet-Antenna-I
html?rmmds=search&cur_warehouse=CN

[8] TTGO ESP32 LoRa OLED
https://www.banggood.com/
Wemos-TTGO-LORA32-868915Mhz-SX1276-ESP32-Oled-display-Bluetooth-WIFI-Lora-p-12
html?rmmds=search&cur_warehouse=CN

[9] GY-NEO6MV2
https://www.aliexpress.com/item/
GY-NEO6MV2-block-new-flight-control-GPS-module-with-EEPROM-MWC-APM2-5-flight-c
1000001413649.html?ws_ab_test=searchweb0_0,searchweb201602_1_
10152_10151_10065_10344_10068_10342_10325_10343_10546_10340_
5722611_10548_10341_10697_10696_5722911_5722811_5722711_10084_
10083_10618_10304_10307_10059_10534_308_100031_10103_441_10624_

10623_10622_10621_10620_5722511,searchweb201603_25,ppcSwitch_5&
algo_expid=cffba55b-9d9e-4301-90bd-614476b39323-1&algo_pvid=
cffba55b-9d9e-4301-90bd-614476b39323&transAbTest=ae803_2&
priceBeautifyAB=0

[10] 128x64 OLED display
https://www.aliexpress.com/item/
1pcs-0-96-blue-0-96-inch-OLED-module-New-128X64-OLED-LCD-LED-Display-Module/
32640627772.html?ws_ab_test=searchweb0_0,searchweb201602_1_
10152_10151_10065_10344_10068_10342_10325_10343_10546_10340_
5722611_10548_10341_10697_10696_5722911_5722811_5722711_10084_
10083_10618_10304_10307_10059_10534_308_100031_10103_441_10624_
10623_10622_10621_10620_5722511,searchweb201603_25,ppcSwitch_5&
algo_expid=e386410e-cb07-484e-9c17-6784e45162cd-3&algo_pvid=
e386410e-cb07-484e-9c17-6784e45162cd&transAbTest=ae803_2&
priceBeautifyAB=0

[11] MakerHawk ESP32 LoRa library
https://github.com/espressif/arduino-esp32

[12] OLED display library
http://stefanfrings.de/esp8266/WIFI-Kit-8-Test2.zip

[13] Espressif
https://www.espressif.com/

[14] TinyGPS++
https://github.com/mikalhart/TinyGPSPlus

[15] NeoGPS library
https://github.com/SlashDevin/NeoGPS

[16] Arduino-LMIC library
https://github.com/matthijskooijman/arduino-lmic

[17] dot2pic converter
http://dot2pic.com/

[18] National Marine Electronics Association (NMEA)
www.nmea.org

[19] I2C protocol explanation
www.i2c-bus.org

[20] The Things Network - limitations
https://www.thethingsnetwork.org/docs/lorawan/limitations.html

[21] Explanation of data rate, packet size, uplink and downlink
https://www.thethingsnetwork.org/forum/t/
limitations-data-rate-packet-size-30-seconds-uplink-and-10-messages-downlink-p
1300

[22] Video that shows use of payload function of The Things Network
https://youtu.be/nT2FnwCoP7w

[23] IBM-Cloud liberty for Java documentation
https://console.bluemix.net/docs/runtimes/liberty/
getting-started.html#getting-started-tutorial

[24] Java SDK for The Things Network
https://www.thethingsnetwork.org/docs/applications/java/

[25] Java SDK for The Things Network Quick start guide
https://www.thethingsnetwork.org/docs/applications/java/
quick-start.html

[26] Git repository with source code of arduino and web-service
https://github.com/dgr1992/Clock-with-weather-forecaste