Implementing NB-IoT: Communication with a Load Cell

Johannes Almroth

5th August 2020

Acknowledgements

Many thanks to Vetek, who funded the purchase of all the hardware equipment used in the project. I am grateful to the Communication Research Lab at Uppsala university for providing me with space and tools to conduct my work.

I would like to thank, especially, Laura Feeney PhD, for your outstanding patience, words of encouragement as well as your time spent in guiding me. In addition, I would like to thank professor Per Gunningberg.

Heartfelt thanks to my partner, Sabina Smedsrud, for always cheering me up and supporting me throughout this journey.

Abstract

IoT technology has been proclaimed as a new technological prowess that will change our economy, our cities and our way of living. Despite these bold statements, IoT is far from being implemented by ordinary tech companies not directly working with any of the enabling technologies, such as telecom. For IoT to live up to the excitement and predictions surrounding it, the hardware and the technical know-how needs to be accessible in the form of reasonable price and complexity as well as increased availability in the form of infrastructure and networks. Narrowband-IoT, a new radio protocol focusing on wide area coverage and low power consumption, is being heralded by the 3GPP as one of the key technologies necessary to push society into the age of IoT. Narrowband-IoT networks are still extremely new in a lot of countries, and while the SIM-cards necessary to use these networks can be readily purchased from telecom companies, the lack of implemented projects might scare the companies looking to implementing IoT within their business. The purpose of this paper is to provide an example of how an IoT device can be implemented in practicality, with the focus being on using a load cell as a sensor.

Using a microcontroller, a hardware connection with a load cell is implemented. Due to time constraint, the data transmission tests are done with a virtual dataset using Wi-fi.

Contents

List of Figures

Chapter 1

Introduction

Internet of Things (IoT) is a broad, diverse and growing field within the IT sector. Many organizations predict that it will come to impact large areas of our daily life, and many telecom companies are experimenting with different kinds of real world applications that can benefit from this change. The basic idea is the same for all devices, which is to communicate via the internet and/or with other device nodes in a network, without the supervision or interaction of a human. Examples range from simple toasters to complex self-driving cars.[14] The focus is to enable communication between devices without the need for a human middleman, thus optimizing whatever application is being implemented. A simple example is a building equipped with multiple IoT-enabled thermostats, which are controlled by a central heating system. Given effective software, heating can be regulated in an energy efficient manner while still keeping visitors adequately warm throughout the day. Another example might be a parking meter, which can forward the availability of its parking spot to some central system which in turn forwards the closest available spot to an end-user. The potential applications are numerous, but factors such as energy consumption and security have proven to be roadblocks that pose considerable challenges to most IoT projects.

Vetek is a Swedish scale supplier located near Väddö island, situated approx. 100 kilometers north of Stockholm. Vetek constructs their own scales and weighing systems, as well as reselling products from other manufacturers.[18]

Vetek aims to improve their services, and as such are interested in the possible use cases of IoT technology, and ultimately see how that can be applied to their own products. With something as simple as an IoT-enabled scale, they can offer customers products that can be placed in remote areas without the need for constant checkups, enabling long term monitoring and making it easier to analyze the data. An example might be monitoring road salt depots, to enable smarter refill routes during winter time, or a fodder station to map the behaviors of local wildlife.

The largest challenges for this type of device lies in energy consumption and broadcast range. Low energy consumption is needed so that any maintainer does not need to make constant check-ups to switch batteries all the time. This poses a limitation on the type of scale that can be used, which in extension affects parameters such as scale accuracy and capacity. A wide broadcast range is needed so that the device is not limited to being close to a base station. This puts restraints on what type of communication protocols can be used, as traditional ones such as Wi-Fi and Bluetooth will not work in the aforementioned examples.

With the advent of IoT, the 3:rd Generation Partnership Program, a standardiz-

ation organization for telecom (3GPP) has developed new wireless communication protocols intended to be used by these devices.[1] One of these, the Narrowband-IoT (NB-IoT) protocol is particularly suitable for the challenges mentioned above, as its focus lies (among else) in wide area coverage and long battery life. A microcontroller (a small computer) is needed to handle the data polled from the scale, as well as sending it via some wireless communication protocol. The microcontroller chosen for this project is a FiPy, as it has the capability to handle multiple wireless technologies, one of them being NB-IoT. Some other essential pieces of hardware needed is some form of power supply, as well as an Analog-to-Digital Converter (ADC) that connects the microcontroller and the scale.

In this paper, an attempt is made to implement a NB-IoT enabled load cell (the term load cell is interchangeable with scale for all intents and purposes). A functioning connection between the load cell and microcontroller could be established near the end of the project, though no real data transmission of this functional data was made due to time constraints. Instead, some fictional data was sent from the microcontroller itself to test its transmission capabilities.

In the following chapter, some general background information regarding the IoT technology and industry as a whole will be presented, as well as the predictions surrounding it. The purpose of NB-IoT and its market situation in Sweden will also be discussed. In order to emulate some behavior needed in the context of a real-world application, a few requirements have been placed on the device in the way that it handles the polling and transmission of data. The desired behavior is of an all-purpose IoT scale, with the requirements being set by Vetek. These requirements will be presented in chapter 3. The hardware and software implementation of the device will be presented in chapter 4. The outcome of the implementation will be presented in chapter 5 and conclusions as well as a discussion of possible future work are brought up in the final chapter.

Chapter 2

Background

2.1 General Background

IoT has been lauded as a world-changing technology that will significantly affect our economy as well as our way of living. In a report by the GSMA, the total number of IoT devices is estimated to triple by 2025, bringing it to $25.2 *billion*.[9] Meanwhile, the global IoT revenue will fourfold from 2018, increasing it to $4.4 *billion*.[9] While there undeniably is a lot of excitement and potential economic impact associated with IoT, currently, many consumers just associate the term with connecting a common toaster or coffee machine to a Wi-Fi network. While this technically fits the definition for an IoT device,[14] the significant use cases will probably be implemented with different sensors, such as scales, thermometers, etc. that will further improve automatization and optimization processes. As an example, the key categories within the predicted growth are smart homes (*e.g.,* security devices) and smart buildings (*e.g.,* energy consumption sensors).[9] For the predicted growth to happen, businesses need to take a chance and work on projects that implement different IoT technologies, and to enable this, the 3GPP has developed some Low-Power Wide-Area Network (LPWAN) protocols that focus on different key aspects that make IoT possible. Some of these aspect include long battery life, high connection density, indoor coverage and geo-tracking capabilities. Aside from security, one of the biggest challenges regarding IoT devices relate to limitations arising from energy infrastructure. As mentioned earlier, one of the core issues NB-IoT aims to achieve is to be a low-power technology, thus decreasing the maintenance needed for battery-powered devices. A claim often paraded with NB-IoT is that it enables a battery-time of up to 10-years,[8] though it is worth mentioning that over such a period of time the underlying IoT technology (in the form of microcontrollers/sensors) will probably require more frequent maintenance than the batteries themselves.

Chapter 3

Requirements

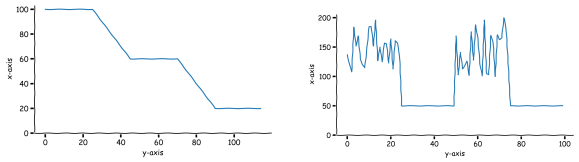
In this section we will discuss some the software requirements imposed on the IoT-enabled scale device that will be implemented, and what behavior we wish to see from the device in the case of different scenarios. These areas were chosen because of their relative simple nature, as well as their relevance to other similar devices. The purpose of these requirements is to emulate the constraints placed on devices in the real world, specifically an all-purpose IoT-scale that Vetek could use in a pilot project for evaluation and future iterations.

There are no specific requirements set for the hardware implementation. Any additional requirements in terms of energy consumption, sizing, etc. would be too vast for the scope of this paper.

3.1 Data Polling Rate

In most IoT devices the relevant data is provided by some form of sensor, whether it be a scale, thermometer or something entirely different.[14] The type of sensor being used has a huge impact on the IoT device, especially when considering that they have to be powered by the same energy source. The simplest way of deciding when to poll data from a sensor is to let it do so at a fixed and constant rate, often enough to be relevant, and seldom enough as to not waste precious energy. However, if within the context of the application we can conclude that no data needs to be polled (for a while), then subsequently no data will need to be sent, and thus we save energy on both ends of the system. For some applications there might even be longer periods of downtime where it is not relevant to conduct monitoring on the given sensor, *e.g.*,during nighttime, closing hours, etc. Another interesting angle is modifying the polling rate depending on the data itself. A simple example of a behavior would be to have a slower polling rate at stable values, and increase it when experiencing large enough changes. Given the conditions of an IoT device powered by batteries, it is not unreasonable to assume that readings might not always be accurate at times. Depending on the sensor, spikes and drops of false values might occur, and not taking these scenarios into account would be prudent.

In this paper, we want the device to change its polling rate depending on the data values, such that the rate increases at periods of activity and change, and lower the rate when the data stabilizes around a given value. The device should also try to take false data values into account.



(a) (b)

Figure 3.1: Graph a) shows plateauing development of values, while graph b) displays spikes in the readings

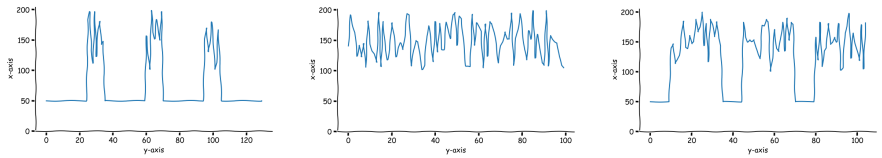
In figure a we see the sensor outputting stable data around a fixed value, to then experiencing a decrease two times. We want the readings to increase during the data changes, and decrease during the stable plateaus.

In figure b we see the sensor outputting some data values with sudden spikes in them. For the sake of simplicity, we will assume that our scale has the maximum capacity of 100 kg, and thus can easily discern that any value above this is a false reading. In other contexts, false readings might be harder to detect and handle.

3.2 Sensor Failure

In this paper, we define sensor failure as a sensor giving too many unreliable or false data values to be considered functional. The goal of identifying such a state in an IoT device is to prevent unstable data from being interpreted as valid, which in turn can save the end user from unwanted consequences. Depending on the longevity and purpose of the device, the threshold of when to declare a sensor as failing may differ, especially as this state can be quite fluid. A functional sensor means different things for different devices and applications. A simple way might be to conclude that if x% of data is considered invalid during the last 24hrs, an alarm should be raised to the device administrator. Complications arise when failures need to be reported quickly, or estimated more thoroughly. It is also possible that the sensor can be temporarily unreliable due to external circumstances, and given enough time, these circumstances might pass. On one extreme you can have a device that reports failures too frequently and bogs down whatever dashboard is handling its status report. On the other, you can have a device taking too long to determine a sensor failure so that false data is believed to be valid in the meantime.

The requirements set on our device state that it has to have some form of self-regulation of when to send these error signals, allowing the sensor some time to recover. If the sensor does not recover within a given timeframe, or outputs too much erroneous data, it will raise an error. The reason for still raising an error even though the device recovered is that the sensor might be in need of maintenance if the % of invalid data is too high. This all depends on the filtering of invalid data, as well as at what data threshold the device loses effectivity.



(a) Sensor recovers after some invalid reads

(b) Sensor does not recover after invalid reads

(c) Sensor produces too much invalid data

Figure 3.2

In figure a the sensor outputs some invalid data, yet it recovers. No errors should be raised.

In figure b the sensor does not recover from reading invalid data, and an error should be raised.

In figure c the sensor does recover, but still outputs enough invalid data that an error should be raised.

3.3 Sensor Disconnect

We define a sensor disconnect as when no credible data is being produced at all. If the sensor does not recover, immediate maintenance is needed for any continued functionality. In this device we know that a sensor disconnect results in the value 0.0 being returned at every poll by the microcontroller to the sensor. While this might be a valid read in some scenarios, in a real world application we would probably never get such a stable value at exactly 0.0. Rather, it would fuzz around at maybe between 1.5 and 0.0 (for example). Disregarding this, we can also know that sensor has disconnect if the data values drop to 0.0 at a too quick pace to be a real-world measurement.

The requirements on this device states that it should raise a disconnect error if the sensor does not recover within a given timeframe.

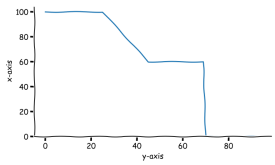


Figure 3.3: The sensor disconnects

In figure 3.3 the data values drop from 60 to 0 in the span of one data point, which would indicate a sensor disconnect.

Chapter 4

Design & Implementation

4.1 Hardware Implementation

A similar paper conducted at the Royal Institute of Technology (KTH) earlier this year served as the main baseline for how the hardware was to be setup.[11] The main components are the microcontroller, the scale as well as the ADC (Analog-to-Digital Converter). Apart from this, an adequate power source is needed to provide energy to all components.

4.1.1 Scale

The scale used for this paper is a Tedea Huntleigh - Model 1022. It is a small and simple model, and the specific device used in this paper had a maximum capacity of 50 kg.[7] In figure 4.1 we can see the labels of the four wires needed to hook up the load cell. The *Input+* and *Input-* signify the voltage input and ground. [17] *Output+* and *Output-* will output a positive respectively a negative charge of about 1.5 voltage. During weighing, the internal resistance in the load cell will change ever so slightly, and the two outputs will have a small difference in the millivoltage range. This difference represents the weight measurement, and can be translated to a corresponding kg/lb value. In general, the more voltage the scale is supplied with, the greater this millivoltage range can be, which (in theory) means larger accuracy when weighing. No two load cells are the same, and they need to be calibrated to output the correct kg/lb value.

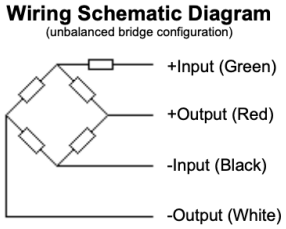


Figure 4.1: The wiring schematic for the load cell.

4.1.2 ADC

To convert the millivoltage output from the load cell into a digital signal, an ADC is needed. The device used in this paper is an HX711, and apart from being a converter,

it also serves as an amplifier for the load cell signal. The front of the ADC is visible in the top of figure 4.2, with the backside below, where the pinout of all the wires from the load cell should be connected. The color coding of the wiring is not the same for all load cells, and the backside should be checked so that the connections follow the correct wiring schematic.

The ADC outputs data via two of its pins, the DAT and the CLK. The CLK pin will output 0 if it is ready to send data, and 1 if it is not ready. When it is ready, the DAT pin will send a series of 0s and 1s that can be converted from binary to a decimal value, which will then represent the output of the load scale.[15] Multiple code libraries have been written to handle this for the user, and which only require a specification of which pins are being used by CLK and DAT respectively. For this paper, a library written for micropython was used.[6]

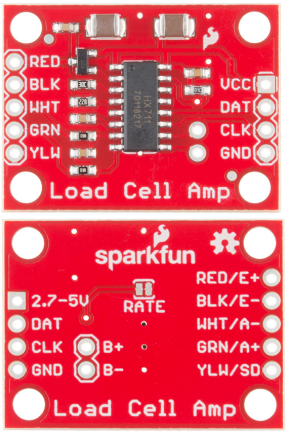


Figure 4.2: The front and back of the HX711.

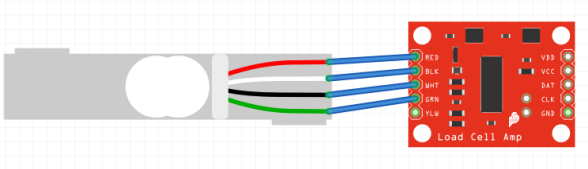
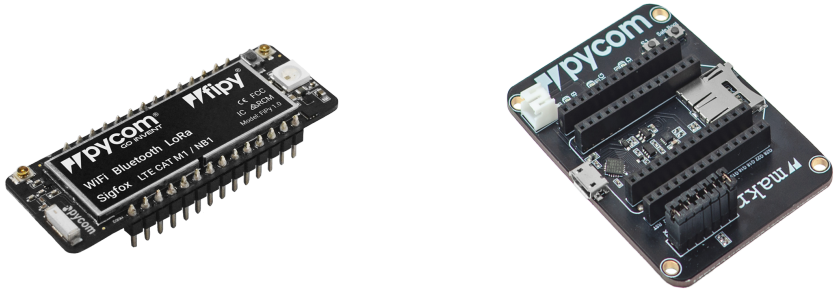


Figure 4.3: The load cell wired to the ADC. The color of the wiring schema corresponds to the schematic seen in figure. 4.1

4.1.3 Microcontroller

The microcontroller used for this paper is the FiPy development board from PyCom. It boasts a wide range of capabilities when it comes to communication protocols, NB-IoT being one of the five available.[12] With the supplied expansion board, connections via pinout is possible. It runs on micropython, which is an implementation of Python 3 optimized to run on microcontrollers.[5] The FiPy can be seen on figure 4.4a and its expansion board on figure 4.4b



(a) (b)

Figure 4.4: The Fipy and the expansion board 3.0

4.1.4 Powering the Project

In the early stages of the project, the hardware was powered via USB cable from a computer. Since the USB was of type micro, the voltage output was 5V. Since this did not yield a functional behavior from the components, another setup was tried.

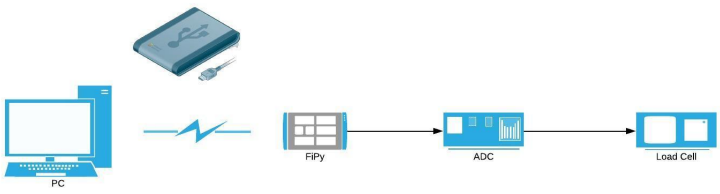


Figure 4.5: The first wiring used[4]

The second approach used two different power, where the ADC was powered by a wall outlet at a higher voltage, while the FiPy kept the USB. Once again, this did not produce a valid result.

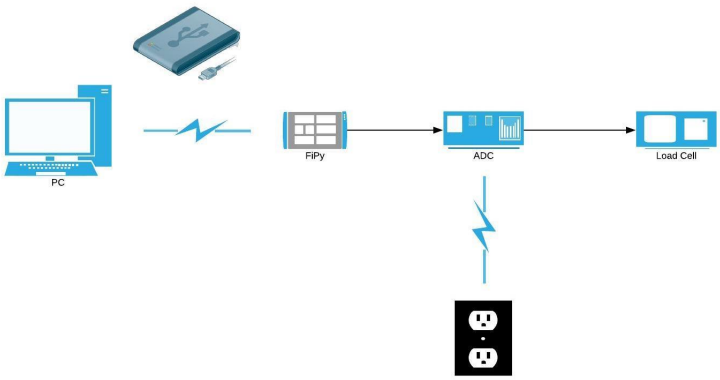


Figure 4.6: The second wiring used[4]

The final wiring involved an Otii battery toolbox, which is an advanced piece of hardware used to profile and emulate batteries.[2] With this piece of equipment, a common ground was provided to all components, as well as adequate voltage. It was capable of supplying 5V to both the FiPy and the ADC at the same time, which in turn enabled stable output of data values.

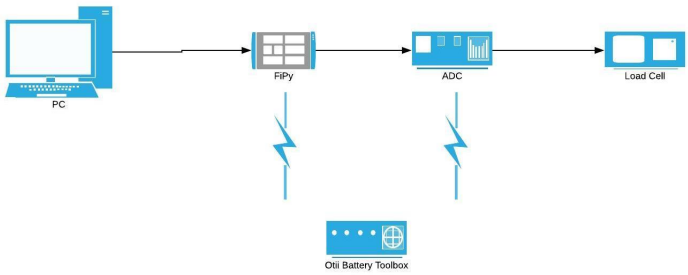


Figure 4.7: The third wiring used[4]

All data values produced by the load cell and ADC during these tests were raw values, as a calibration for kg/lb values for our intents and purposes would not bring any significant enhancements to the project.

4.2 Software Implementation

The FiPy code runs on MicroPython, an implementation of Python 3 optimized for microcontrollers. MicroPython only executes two files on its system's root folder, the boot.py and the main.py files. Any remaining code must be placed in the /lib/ folder. The boot.py runs first, and is intended to contain low-level code that is meant to configure the hardware. The main.py file contains the main program loop, and imports auxiliary files from the /lib/ folder. The two files used for the implementation in the /lib/ folder are reader.py and error_tracker.py. A test file has been written for each module, to ensure that the software did not regress during development. All python classes and their respective tests can be found in the appendix.

The software tests were built using a micropython library called unittest, and the tests were constructed so that they would mirror the requirements set in the Requirements chapter. The files used for testing in the /lib/ folder are the unittest.py, test_reader_poll_rate.py and test_error_tracker.py.

Chapter 5

Related Work

A similar project done at KTH in 2019 served as the main inspiration for this paper.[11] The goal of that project was to monitor the battery levels in defibrillators via an IoT-enabled scale, and thereby optimize the battery-swapping routine. This project proved successful in it's implementation, but faced a slew of challenges that prevented it from ever being integrated into an actual user environment, due to security concerns related to the use of the hospital Wi-Fi. This shows the importance of seeing the bigger picture of where and how the device will be implemented by the end-user, and having that in mind when making technology and design trade-offs throughout the development process.

One of the implementation details in this project concerned the data polling rate, which could potentially affect battery life of a sensor. The ways to regulate data polling is as varied as there are systems and devices that implement it, and apart from looking at the the energy consumption between the polling unit and the sensor, the following two articles look at the aspect of data polling in the context of a system, which is particularly interesting in the perspective of IoT since multiple devices will often share the same network and need to optimize the use of shared resources.

In a 2018 study by Siddiqui *et al.*[16] a new protocol was developed for the MAC layer of a wireless sensor network which adjusted its polling interval depending on the rate of incoming traffic, and given certain types of traffic, was able to optimize energy and delay performance compared to another MAC protocol. In a work by Yu *et al.*[19] concerning electromagnetic-based wireless nano sensor networks, a polling scheme is proposed that adjusts itself according to network conditions on the IoT backhaul portion of the system, which improved bandwidth efficiency and lessened energy consumption.

Relating to error detection in this paper, a simple and naive check in the form of value outliers was implemented. In an article by Abedjan *et al.*this would fall in under the *quantitive error detection* category [3], and suitable next step would be to include one of the other four categories of data cleaning tools and algorithms. The article provides a good overview of what to keep in mind when ensuring data quality, such as running multiple tools, the order in which they run and ensuring that end-user effort is kept to a minimum.

Chapter 6

Results

6.1 Hardware results

The configuration and setup between the load cell and ADC took some time to get right, mostly due a faulty load cell being used during the first half of the project. It was also assumed that the color of the wires was standardized between load cells, so the correct wiring in figure 4.3 was not implemented right away. Even though no calibrations were made to the load cell during this paper, an increase in the voltage provided to the load cell did correlate to an increased stability in the values produced.

The second big challenge involving the hardware came from powering the project in a sufficient manner. The first power setup as seen in figure 4.5, was a simple and naive approach that lacked enough voltage to efficiently power the components. The benefit of this setup was a simple wiring schema where each component powered the next in line. The drawback was that the FiPy could only supply the ADC with 3V, which in turn affected the ADC's ability to read data from the load cell. During testing, the output rate of the raw data would be infrequent and erratic, sometimes taking several seconds to produce a single value. The values themselves did not correspond to increases and decreases in force being applied to the load cell, and would seemingly spike and crash at random. These problems were largely in part due to the insufficient voltage being supplied to the ADC and load cell as later setups would reveal.

The second wiring setup was the one seen in figure 4.6, and resulted in an electrical interference throughout the system, because of two different grounds being present in the circuit. This electrical interference rendered the output of the system nonsensical at time, though output rate of the data values had improved to a bit more stable rhythm than previously, and the raw data values were a bit more responsive to the force applied to the load cell..

The third and final power setup for the project involved the Otii battery toolbox, as seen in figure 4.7. This setup, though a bit too advanced for a device intended to be small and simple, did produce a functioning connection between the load cell and the microcontroller. When force was applied to the load cell, the data values responded accordingly with an increase or decrease in value, and no irregularities in form of disconnects or spikes were seen in the tests. Due to time constraints and hardware availability, the setup could not be used for real network transmissions of the load cell data.

6.2 Software results

6.2.1 reader.py

The reader.py file contains the Reader class, which is responsible for processing the data polled from the load cell and passing it on to being transmitted. To instantiate a functional reader object, it needs to be passed a poller function as well as a transmitter function. The purpose of having these functions passed to the instance of the class instead of being hardcoded into the class is to follow the separation of concern design principle.[10] The type signatures for the two functions are shown below. The poller function should accept no argument and is expected to return the current data value produced by the load cell when called. The transmitter function can accept a argument of any type, and should not return any value.

```
def poller_function() -> value: int
def transmitter_function(value: Any): -> None
```

The main method of the reader instance is the run() method. When called, the run() method performs a cycle consisting of data polling, a check for false values, adjustment of the polling rate as well as a possible transmission.

Error Check

The polled data value is subsequently checked for validity in the form of out-of-bounds values or extreme delta changes. A separate class called Error_tracker monitors the interval and frequency of these occurrences, and the purpose of the class is to raise an exception when the error rate is deemed too high, and some form of remedial action needs to be taken. The internal workings of the Error_tracker will be explained in a separate section.

Disconnect Check

In the case of a disconnect, the sensor outputs 0.0, and if the preceding values in the data buffer are not anywhere close to 0 we should raise a disconnect error as per the requirements. To do this, we set an arbitrary cutoff limit at half the permitted max value. If the polled data is 0.0 while the data buffer still contains values above the specified limit, it is considered a disconnect since the drop in values is considered to high to be a natural application behavior.

Adjustment of polling rate

If the value is deemed valid, it is added to a First-in, First-out (FIFO) buffer of the most recent values. The contents of the buffer are then summed into a total delta value, which is used to adjust the polling rate. If the total delta surpasses a pre-defined threshold, the polling rate is increased, whereas if it is lower it might be maintained or decreased. The total delta value is the the delta values between two points added to the delta of the next two points, as follows:

$$\sum_{n=0}^{10} x_n - x_{n+1}$$

6.2.2 error_tracker.py

The purpose of the Error_tracker class is to keep track of the error occurrence and frequency. The intended usage is to instantiate an instance of the class, and call the error_occurred () method when an invalid read has occurred. This method increases an internal counter, which will then raise an exception if it passes a pre-defined threshold.

The Grace Period

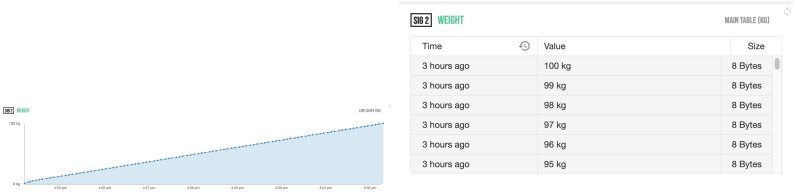
When invalid reads occur due to some temporary circumstance, it might not be beneficial to count all errors within a given timeframe towards raising an exception. It is not useful sending an alert when short periods of errors occur (given that very high uptime is not of importance), since this might risk producing a multitude of needless error messages. Instead, it is at the long-lasting periods of polling errors an exception should be raised. To account for short bursts of error, a constant called GRACE_PERIOD is used. This constant measures the time window (in seconds) after the error_occurred () method has been called, during which sequential calls will not count toward the exception threshold.

The Cooldown Period

If errors rack up towards an exception over a longer period of time, it would lose meaning in regards to the status of the device in the short term, and would only be an indicator of long-term performance. To avoid this, the internal error counter needs to be reset or decreased periodically. To achieve this, a constant similar to the GRACE_PERIOD is used, called the COOLDOWN constant. This constant indicates the time window (in seconds) after the grace period, where if no calls are made to the error_occurred () method, the internal error counter is decreased by one. However, if a call to the error_occurred () method is made during the cooldown period, the internal error counter increases, and a new grace period starts. This way of self-regulation ensures that only a long-lasting and consistent frequency of errors raise an exception.

6.2.3 Data Transmissions

Data transmission tests were done separately from the load cell using fictional data. Pycom, the parent company behind the microcontroller used in this project also operate a cloud-based device management platform. [13] Via the pybytes platform, configuration of the network settings of the device can be managed via a firmware updater. In figure 6.1a and figure 6.1b we can see data being transmitted via Wifi on a home network. The data is then displayed via graphs on the pybytes platform.



(a) Data transmitted via Wifi (b) Values transmitted via Wifi

Figure 6.1

Regrettably, due to time limitations in the project, no actual tests of the transmission of the data via the NB-IoT protocol were performed.

6.3 Discussion

Despite hardships and complications when implementing the hardware, the results suggest that building and programming an NB-IoT enabled device connected to a load cell is possible with fairly simple consumer available hardware devices. It is a reasonable assumption that the hardware hindrances encountered in this paper can be bypassed with adequate planning and experience in assembling electric hardware.

These results show a small starting point of implementing a IoT device using a load cell as its sensor. With the rise of 5G and IoT platform services there exists a real commercial interest to enable more and more actors to innovate and create products for consumers and companies alike. This project can serve as a starting point for what design and requirement considerations regarding hardware and software that need to be considered when planning and designing such a device.

6.3.1 Limitations

The most glaring issue with the way the hardware and software were tested during this project pertains to the time duration. Any potential end-use application would require the device to be left unattended for period of at least a couple of days. This would have tested the ability of both the network and the device to communicate in sub-optimal conditions, as well as recovering from any severed connection. This project was done on a small scale, and extensive research in multiple different areas need to be conducted to even get a IoT enabled scale close to being produced for functional use, commercial or private. Furthermore, these results cannot account for whether the software implementation was close to emulating the desired behavior of a device from a practical standpoint, since no potential end-users were involved in the requirement specification. Since the testing was only done indoors in a clean and controlled environment, unknown variables present in the real world could very well produce a multitude of challenges that change the way that the device works. Another interesting angle is how different locations would interact with the connection to the cellular network the NB-IoT SIM-card relies on for communication. The NB-IoT technology makes huge promises, but at the end of the day it is up to the local telecom company to fulfill the underlying conditions that make those claims possible, which would be Telia in our case.

Expanding on this, since the NB-IoT technology is fairly new in a lot of countries, there is bound to be extremely different implementation experiences from region to region depending on the network provider. In fact, NB-IoT devices could be rendered obsolete in entire regions depending on the network provider.

Chapter 7

Conclusions

With the help of the Otii battery toolbox, a functioning load cell-to-microcontroller setup was devised, though no end-to-end data transfer via network was made with the real data. Separate tests with the FiPy showed capabilities to send data via WiFi in conjunction with the PyCom platform, PyBytes.The device also fulfilled the software behavior requirements imposed to emulate real world situations and usage, though it is worth repeating that these requirements were not based on tests and research.

7.1 Future Work

The most focal point regarding future work to be done on this IoT scale is to involve an end-user early on. Ideally, a UX-designer would lead some form of market research study to more accurately specify what needs and requirements this kind of device would need to satisfy. This would be used to guide any further modifications and limitations put on the device. A more proper analyze of the needed hardware components needs to be done. Optimization in the form of hardware space, energy consumption and economical costs are essential if any more than a handful of devices are to be produced. A more UX and design related question would pertain how standardized the software of the device needs to be constructed. Depending on how the use cases and market needs, can one solution fit all, or is there a way to tailor to needs in an effective way?

While the predictions and promises for the IoT industry remain hopeful and grandeur, no new devices and products will appear from thin air and propel IoT into being an integral pillar to our modern society just like that. The technology available today enables smarter products that have the potential to vastly improve our lives so long as we keep testing, trying, failing and improving. This paper was one, albeit small, test of these new technologies and many exciting projects are sure to come.

Chapter 8

Appendix

8.1 main.py

```
# pylint: disable=undefined-variable
# pylint: disable=import-error
RUN_TESTS = True
RUN_PROGRAM = False

def t(value):
        print("Now sending value", value)
        pybytes.send_signal(2, value)

if RUN_TESTS:
        from test_reader_poll_rate import *
        from test_error_tracker import *
        unittest.main()
if RUN_PROGRAM:
        import reader as re
        r = re.Reader(transmitter=t)
        while True:
                r.run()
```

8.2 reader.py

```
# pylint: disable=import-error
from utime import sleep

def generator(start = 0, inc = 1):
        x = start
        while True:
                yield x
                x += inc

gen = generator()

def default_poller():
        return next(gen)
```

```
def default_transmitter(*args):
        return

class DisconnectErrorException(Exception):
        pass

class Reader:
        def __init__(self,
        testing = False,
        poller = default_poller,
        transmitter = default_transmitter,

        buffer_len = 10,

        unit_value_max = 100,
        unit_value_min = 0,

        polling_delay_init = 3,
        polling_delay_max = 5,
        polling_delay_min = 1,
        polling_delay_inc = 0.5,

        delta_treshold = 10,
        dc_treshold = 10,

        debug = False
        ):
                self.DEBUG = debug
                self.TESTING = testing
                self.poller = poller
                self.transmitter = transmitter
                self.data_buffer = FIFO(buffer_len)
                self.DELTA_THRESHOLD = delta_treshold
                self.DC_THRESHOLD = dc_treshold
                # Defines the range of units that can be
                    polled
                # while still considered normal within
                   the
                # context of the application
                self.MAX_UNIT_VALUE = unit_value_max
                self.MIN_UNIT_VALUE = unit_value_min

                # Defines the range of values the delay
                # between pollings can have and the
                # increment when increasing/decreasing
                   it
                self.polling_delay = polling_delay_init
                self.POLLING_DELAY_MAX =
                   polling_delay_max
```

```
        self.POLLING_DELAY_MIN =
           polling_delay_min
        self.POLLING_DELAY_INC =
           polling_delay_inc

        self.polling_delay_setup_check()

        self.iterations = 0

def polling_delay_setup_check(self):
        if (self.polling_delay <= 0):
              raise ValueError("Initial␣
                 polling␣delay␣cannot␣be␣zero␣
                 or␣less")

        if (self.POLLING_DELAY_MAX < self.
           polling_delay):
              raise ValueError("Initial␣max␣
                 value␣for␣polling␣delay␣
                 cannot␣be␣less␣than␣the␣
                 initial␣delay␣value")

        if (self.POLLING_DELAY_MIN > self.
           polling_delay):
              raise ValueError("Initial␣
                 minimum␣value␣for␣polling␣
                 delay␣cannot␣be␣more␣than␣the
                 ␣initial␣delay␣value")

        if (self.POLLING_DELAY_MIN < 0):
              raise ValueError("Initial␣
                 minimum␣value␣for␣polling␣
                 delay␣cannot␣be␣less␣than␣
                 zero")

        if (self.POLLING_DELAY_INC < 0):
              raise ValueError("Initial␣
                 increment␣value␣for␣polling␣
                 delay␣cannot␣be␣less␣than␣
                 zero")

        diff = self.POLLING_DELAY_MAX - self.
           POLLING_DELAY_MIN
        if (diff % self.POLLING_DELAY_INC != 0):
              raise ValueError("Initial␣
                 increment␣value␣for␣polling␣
                 delay␣must␣be␣divisible␣by␣
                 the␣difference␣between␣the␣
                 max␣and␣min␣values␣of␣the␣
                 polling␣delay")
```

```
            diff = self.polling_delay - self.
               POLLING_DELAY_MIN
            if (diff % self.POLLING_DELAY_INC != 0):
                  raise ValueError("Distance␣
                     between␣initial␣polling␣delay
                     ␣and␣mininimum␣polling␣delay␣
                     must␣be␣divisible␣by␣polling␣
                     delay␣increment")

            diff = self.POLLING_DELAY_MAX - self.
               polling_delay
            if (diff % self.POLLING_DELAY_INC != 0):
                  raise ValueError("Distance␣
                     between␣initial␣polling␣delay
                     ␣and␣maximum␣polling␣delay␣
                     must␣be␣divisible␣by␣polling␣
                     delay␣increment")

      def run(self, iterations = 1):
            for _ in range(iterations):
                  if(not self.TESTING): sleep(self
                     .polling_delay)

                  self.iterations += 1
                  value = self.poller()
                  if(self.verify_data(value)):
                        self.data_buffer.add(
                           value)

                        if self.DEBUG: print("
                           Queue␣is", self.
                           data_buffer.queue)

                        self.dc_check()
                        self.adjust_polling_rate
                           ()
                        self.transmitter(value)

      def dc_check(self):
            if (self.data_buffer.maxlen != len(self.
               data_buffer.queue)): return

            x = list(filter((lambda y: y > 50), self
               .data_buffer.queue[:3]))
            z = list(filter((lambda y: y == 0), self
               .data_buffer.queue[4:]))

            if self.DEBUG: print("x␣is", list(x), "z
               ␣is", list(z))
```

```
        if (any(x) and (len(z) != 0)):
                raise DisconnectErrorException


def verify_data(self, value):
        return (value <= self.MAX_UNIT_VALUE)
           and (value >= self.MIN_UNIT_VALUE)

def adjust_polling_rate(self):
        delta = self.current_delta()
        if abs(delta) > self.DELTA_THRESHOLD:
                self.decrease_polling_delay()
        else:
                self.increase_polling_delay()

def current_delta(self):
        if self.DEBUG: print("Queue␣is", self.
           data_buffer.queue)
        prev = self.data_buffer.queue[0]
        total_delta = 0
        for x in self.data_buffer.queue[1:]:
                new_delta = prev - x
                total_delta += new_delta
                prev = x

        if self.DEBUG: print("Current␣delta␣is",
            total_delta)

        return total_delta

def increase_polling_delay(self):
        if self.polling_delay < self.
           POLLING_DELAY_MAX:
                new_polling_delay = self.
                   polling_delay + self.
                   POLLING_DELAY_INC

                if self.DEBUG: print("Increased␣
                   the␣delay␣from␣", self.
                   polling_delay, "to",
                   new_polling_delay)

                self.polling_delay =
                   new_polling_delay
        else:
                if self.DEBUG: print("Already␣at
                   ␣max␣polling␣delay")


def decrease_polling_delay(self):
```

```
                if self.polling_delay > self.
                    POLLING_DELAY_MIN:
                        new_polling_delay = self.
                            polling_delay - self.
                            POLLING_DELAY_INC

                        if self.DEBUG: print("Decreased␣
                            the␣delay␣from␣", self.
                            polling_delay, "to",
                            new_polling_delay)

                        self.polling_delay =
                            new_polling_delay
                else:
                        if self.DEBUG: print("Already␣at
                            ␣min␣polling␣delay")

class FIFO:
    def __init__(self, maxlen):
        self.maxlen = maxlen
        self.queue = []

    def add(self, value):
        self.queue.append(value)
        if len(self.queue) > self.maxlen:
            self.queue = self.queue[-self.maxlen:]
```

8.3 test_reader_poll_rate.py

```
# pylint: disable=import-error
# pylint: disable=undefined-variable
# pylint: disable=relative-beyond-top-level
import unittest
import reader as re

class TestReaderPollRate(unittest.TestCase):

    def test_run_function_works(self):
        # Arrange
        r = re.Reader(testing=True)

        # Act
        r.run(10)

        # Assert
        self.assertTrue(r.iterations == 10)

    def test_polling_delay_increment_mismatch(self):
        with self.assertRaises(ValueError):
            re.Reader(testing=True, polling_delay_inc
```

```
            =0.5, polling_delay_init=0.6)


def test_polling_delay_increment_correct_input(self)
    :
    try:
        re.Reader(testing=True, polling_delay_max=1
            ,polling_delay_min=0, polling_delay_init
            =1, polling_delay_inc=0.25)
    except:
        self.fail()

def test_polling_delay_decreases(self):
    # Arrange
    l = [i for i in range(0, 100, 5)]
    p = iter(l)
    r = re.Reader(testing=True, poller=lambda: next(
        p))
    init_delay = r.polling_delay

    # Act
    r.run(20)

    # Assert
    self.assertTrue(init_delay > r.polling_delay)
    self.assertTrue(r.iterations == 20)

def test_polling_delay_increases(self):
    # Arrange
    l = [50 for i in range(20)]
    p = iter(l)
    r = re.Reader(testing=True, poller=lambda: next(
        p), delta_treshold=5)
    init_delay = r.polling_delay

    # Act
    r.run(20)

    # Assert
    self.assertTrue(r.polling_delay > init_delay)
    self.assertTrue(r.iterations == 20)

def test_polling_delay_fluctuates(self):
    # Arrange
    l = [i for i in range(0, 100, 5)]
    l.extend([100 for i in range(20)])
    l.extend([i for i in range(100, 0, -5)])
    l.extend([0 for i in range(100)])
    p = iter(l)
    r = re.Reader(testing=True, poller=lambda:next(p
```

```
            ), delta_treshold=5)
        prev_delay = r.polling_delay

        # Act / Assert
        r.run(20)
        self.assertTrue(prev_delay > r.polling_delay)
        prev_delay = r.polling_delay
        r.run(20)
        self.assertTrue(r.polling_delay > prev_delay)
        prev_delay = r.polling_delay
        r.run(20)
        self.assertTrue(prev_delay > r.polling_delay)

        self.assertTrue(r.iterations == 60)

    def test_false_values_dont_affect(self):
        l = [75 for i in range(10)]
        l.extend([25 for i in range(10)])
        l.extend([100 for i in range(10)])
        l.extend([150 for i in range(10)])
        p = iter(l)
        r = re.Reader(testing=True, poller=lambda:next(p
            ), unit_value_min=50, unit_value_max=100)

        # Act / Assert
        r.run(10)

        prev_delay = r.polling_delay
        r.run(10)
        self.assertTrue(prev_delay == r.polling_delay)

        prev_delay = r.polling_delay
        r.run(10)
        self.assertTrue(prev_delay != r.polling_delay)

        prev_delay = r.polling_delay
        r.run(10)
        self.assertTrue(prev_delay == r.polling_delay)

        self.assertTrue(r.iterations == 40)

    def test_poller_disconnect(self):
        l = [100 for i in range(10)]
        l.extend([75 for i in range(10)])
        l.extend([0 for i in range(10)])
        p = iter(l)

        r = re.Reader(testing=True, poller=lambda:next(p
            ))
```

```
        with self.assertRaises(re.
            DisconnectErrorException):
            r.run(30)

if __name__ == '__main__':
    unittest.main()
```

8.4 error_tracker.py

```
# pylint: disable=import-error
from utime import sleep
import _thread

class MaxErrorException(Exception):
        pass

class Error_tracker:
        def __init__(self,
        grace_period = 1,
        cooldown = 1,
        max_allowed_errors = 10,

        debug = False):
                self.GRACE_PERIOD = grace_period
                self.COOLDOWN = cooldown
                self.MAX_ALLOWED_ERRORS =
                    max_allowed_errors
                self.error_count = 0
                self.DEBUG = debug

                self.grace_period_is_active = False
                self.cooldown_is_active = False
                self.cooldown_id = None

        def error_occurred(self):
                if(self.grace_period_is_active == False)
                    :
                        if(self.error_count >= self.
                            MAX_ALLOWED_ERRORS): raise
                            MaxErrorException

                        self.grace_period_is_active =
                            True
                        self.cooldown_is_active = False
                        self.error_count += 1

                        _thread.start_new_thread(self.
                            grace_period_timer, (1,))

                if(self.DEBUG): print("error_count␣is␣
```

```
                now", self.error_count)


        def grace_period_timer(self, args):
                sleep(self.GRACE_PERIOD)

                if(self.DEBUG): print("Exiting grace
                    period")

                self.grace_period_is_active = False
                self.cooldown_timer()

        def cooldown_timer(self):
                self.cooldown_id = _thread.get_ident()
                self.cooldown_is_active = True
                sleep(self.COOLDOWN)

                if(self.cooldown_is_active and
                not self.grace_period_is_active and
                self.cooldown_id == _thread.get_ident())
                    :

                        self.error_count -= 1
                        if(self.DEBUG): print("Cooldown
                            successful")
                        self.cooldown_is_active = False
```

8.5 test_error_tracker.py

```
# pylint: disable=import-error
# pylint: disable=undefined-variable
# pylint: disable=relative-beyond-top-level
import unittest
import error_tracker as f
from utime import sleep
import _thread

class TestErrorTracker(unittest.TestCase):

        def test_runs(self):
                ft = f.Error_tracker()
                ft.error_occurred()
                self.assertTrue(ft.error_count == 1)

        def test_grace_period_prevents_strikes(self):
                ft = f.Error_tracker(grace_period=1)
                ft.error_occurred()
                ft.error_occurred()
                self.assertTrue(ft.error_count == 1)
```

```
def
   test_grace_period_start_allowing_strikes_again
   (self):
        ft = f.Error_tracker(grace_period=0.5)
        ft.error_occurred()
        sleep(1)
        ft.error_occurred()
        self.assertTrue(ft.error_count == 2)

def test_cooldown_removes_strikes(self):
        ft = f.Error_tracker(grace_period=0.1,
           cooldown=0.1)
        ft.error_occurred()
        self.assertTrue(ft.error_count == 1)
        sleep(0.3)
        self.assertTrue(ft.error_count == 0)

def test_cooldown_allows_strikes(self):
        ft = f.Error_tracker(grace_period=0.1,
           cooldown=0.1)
        ft.error_occurred()
        self.assertTrue(ft.error_count == 1)
        sleep(0.3)
        ft.error_occurred()
        self.assertTrue(ft.error_count == 1)

def test_cooldown_allows_fluctuating_strikes(
   self):
        ft = f.Error_tracker(grace_period=0.1,
           cooldown=0.5)
        ft.error_occurred()
        self.assertTrue(ft.error_count == 1)
        sleep(0.2)
        ft.error_occurred()
        self.assertTrue(ft.error_count == 2)
        sleep(3)
        self.assertTrue(ft.error_count == 1)

def test_max_strikes_error_assertion(self):
        ft = f.Error_tracker(max_allowed_errors
           =10, grace_period=0.1, cooldown=1)
        with self.assertRaises(f.
           MaxErrorException):
                for i in range(11):
                        sleep(0.2)
                        ft.error_occurred()
```

```
if __name__ == '__main__':
    unittest.main()
```

8.6 unittest.py

```
# Code taken from https://github.com/micropython/
    micropython-lib/tree/master/unittest

import sys


class SkipTest(Exception):
    pass


class AssertRaisesContext:

    def __init__(self, exc):
        self.expected = exc

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, tb):
        if exc_type is None:
            assert False, "%r not raised" % self.
                expected
        if issubclass(exc_type, self.expected):
            return True
        return False


class TestCase:

    def fail(self, msg=''):
        assert False, msg

    def assertEqual(self, x, y, msg=''):
        if not msg:
            msg = "%r vs (expected) %r" % (x, y)
        assert x == y, msg

    def assertNotEqual(self, x, y, msg=''):
        if not msg:
            msg = "%r not expected to be equal %r" % (x,
                y)
        assert x != y, msg

    def assertAlmostEqual(self, x, y, places=None, msg='
        ', delta=None):
```

```
    if x == y:
        return
    if delta is not None and places is not None:
        raise TypeError("specify␣delta␣or␣places␣not
            ␣both")

    if delta is not None:
        if abs(x - y) <= delta:
            return
        if not msg:
            msg = '%r␣!=␣%r␣within␣%r␣delta' % (x, y
                , delta)
    else:
        if places is None:
            places = 7
        if round(abs(y-x), places) == 0:
            return
        if not msg:
            msg = '%r␣!=␣%r␣within␣%r␣places' % (x,
                y, places)

    assert False, msg

def assertNotAlmostEqual(self, x, y, places=None,
    msg='', delta=None):
    if delta is not None and places is not None:
        raise TypeError("specify␣delta␣or␣places␣not
            ␣both")

    if delta is not None:
        if not (x == y) and abs(x - y) > delta:
            return
        if not msg:
            msg = '%r␣==␣%r␣within␣%r␣delta' % (x, y
                , delta)
    else:
        if places is None:
            places = 7
        if not (x == y) and round(abs(y-x), places)
            != 0:
            return
        if not msg:
            msg = '%r␣==␣%r␣within␣%r␣places' % (x,
                y, places)

    assert False, msg

def assertIs(self, x, y, msg=''):
    if not msg:
        msg = "%r␣is␣not␣%r" % (x, y)
```

```
    assert x is y, msg

def assertIsNot(self, x, y, msg=''):
    if not msg:
        msg = "%r is %r" % (x, y)
    assert x is not y, msg

def assertIsNone(self, x, msg=''):
    if not msg:
        msg = "%r is not None" % x
    assert x is None, msg

def assertIsNotNone(self, x, msg=''):
    if not msg:
        msg = "%r is None" % x
    assert x is not None, msg

def assertTrue(self, x, msg=''):
    if not msg:
        msg = "Expected %r to be True" % x
    assert x, msg

def assertFalse(self, x, msg=''):
    if not msg:
        msg = "Expected %r to be False" % x
    assert not x, msg

def assertIn(self, x, y, msg=''):
    if not msg:
        msg = "Expected %r to be in %r" % (x, y)
    assert x in y, msg

def assertIsInstance(self, x, y, msg=''):
    assert isinstance(x, y), msg

def assertRaises(self, exc, func=None, *args, **
   kwargs):
    if func is None:
        return AssertRaisesContext(exc)

    try:
        func(*args, **kwargs)
        assert False, "%r not raised" % exc
    except Exception as e:
        if isinstance(e, exc):
            return
        raise
```

```
def skip(msg):
    def _decor(fun):
        # We just replace original fun with _inner
        def _inner(self):
            raise SkipTest(msg)
        return _inner
    return _decor

def skipIf(cond, msg):
    if not cond:
        return lambda x: x
    return skip(msg)

def skipUnless(cond, msg):
    if cond:
        return lambda x: x
    return skip(msg)


class TestSuite:
    def __init__(self):
        self.tests = []
    def addTest(self, cls):
        self.tests.append(cls)

class TestRunner:
    def run(self, suite):
        res = TestResult()
        for c in suite.tests:
            run_class(c, res)

        print("Ran %d tests\n" % res.testsRun)
        if res.failuresNum > 0 or res.errorsNum > 0:
            print("FAILED (failures=%d, errors=%d)" % (
                res.failuresNum, res.errorsNum))
        else:
            msg = "OK"
            if res.skippedNum > 0:
                msg += " (%d skipped)" % res.skippedNum
            print(msg)

        return res

class TestResult:
    def __init__(self):
        self.errorsNum = 0
        self.failuresNum = 0
        self.skippedNum = 0
        self.testsRun = 0
```

```
        def wasSuccessful(self):
            return self.errorsNum == 0 and self.failuresNum
                == 0

    def run_class(c, test_result):
        o = c()
        set_up = getattr(o, "setUp", lambda: None)
        tear_down = getattr(o, "tearDown", lambda: None)
        for name in dir(o):
            if name.startswith("test"):
                print("%s (%s) ..." % (name, c.__qualname__)
                    , end="")
                m = getattr(o, name)
                set_up()
                try:
                    test_result.testsRun += 1
                    m()
                    print(" ok")
                except SkipTest as e:
                    print(" skipped:", e.args[0])
                    test_result.skippedNum += 1
                except:
                    print(" FAIL")
                    test_result.failuresNum += 1
                    # Uncomment to investigate failure in
                        detail
                    #raise
                    continue
                finally:
                    tear_down()


    def main(module="__main__"):
        def test_cases(m):
            for tn in dir(m):
                c = getattr(m, tn)
                if isinstance(c, object) and isinstance(c,
                    type) and issubclass(c, TestCase):
                    yield c

        m = __import__(module)
        suite = TestSuite()
        for c in test_cases(m):
            suite.addTest(c)
        runner = TestRunner()
        result = runner.run(suite)
        # Terminate with non zero return code in case of
            failures
        sys.exit(result.failuresNum > 0)
```

Bibliography

[1] 3GPP. 3gpp, 2019. URL https://www.3gpp.org.

[2] Qoitech AB. Otii battery toolbox. URL https://www.qoitech.com/products/battery-toolbox.

[3] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, M. R. Stonebraker, and Nan Tang. Detecting data errors: where are we and what needs to be done? *Proceedings of the VLDB Endowment*, 9(12):993–1004, 2016. doi: 10.14778/2994509.2994518.

[4] Lucid Chart. Lucid chart, 2020. URL https://www.lucidchart.com/.

[5] Damien George. Micropython. URL https://micropython.org/.

[6] David Gerber. hx711-lopy. https://github.com/geda/hx711-lopy, 2019.

[7] Vishay Precision Group. Model 1022, 2017. URL http://www.scalesnet.com/files/users/PDF/TEDEA/1022.pdf.

[8] GSMA. Narrowband – internet of things (nb-iot), 2019. URL https://www.gsma.com/iot/narrow-band-internet-of-things-nb-iot/.

[9] GSMA Intelligence. The mobile economy. GSMA's Mobile Economy report series https://www.gsma.com/r/mobileeconomy/, 2019.

[10] Philip A. Laplante. *What Every Engineer Should Know About Software Engineering*. CRC Press, Taylor & Francis Group, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742, 2007. URL https://books.google.se/books?id=pFHYk0KWAEgC&pg=PA85&dq=%22separation+of+concerns%22&hl=en&sa=X&ei=WQ_aUNn5DYjNiwLS54GADQ&redir_esc=y#v=onepage&q=%22separation%20of%20concerns%22&f=false.

[11] Lisa Mach and Maneejun Kadepisarn. Internetuppkopplade vågar till södersjukhuset. Bachelor's Thesis, June 2019.

[12] PyCom. Fipy, 2019. URL https://docs.pycom.io/datasheets/development/fipy/.

[13] Pycom. Pybytes, 2020. URL https://pycom.io/solutions/software/pybytes/.

[14] Steve Ranger. What is the iot? everything you need to know about the internet of things right now, 2018. URL https://www.zdnet.com/article/what-is-the-internet-of-things-everything-you-need-to-know-about-the-iot-right-now/.

[15] AVIA Semiconductor. 24-bit analog-to-digital converter (adc) for weigh scales. URL https://cdn.sparkfun.com/assets/b/f/5/a/e/hx711F_EN.pdf.

[16] Shama Siddiqui, Sayeed Ghani, and Anwar Ahmed Khan. Adp-mac: An adaptive and dynamic polling-based mac protocol for wireless sensor networks. *IEEE Sensors Journal*, 18(2):860–874, 2018. doi: 10.1109/JSEN.2017.2771397.

[17] Dara Trent. Understanding load cell specifications, 2019. URL https://www.800loadcel.com/white-papers/377.html.

[18] Vetek. Om vetek, 2019. URL https://www.vetek.se/om-vetek/content.

[19] Hang Yu, Bryan Ng, and Winston K. G. Seah. On-demand probabilistic polling for nanonetworks under dynamic iot backhaul network conditions. *IEEE Internet of Things Journal*, 4(6):2217–2227, 2017. doi: 10.1109/JIOT.2017.2751524.