Smart IoTs Based Urine Measurement System

Bournemouth University

RUSHAN ARSHAD

A thesis submitted in partial fulfilment of the
Requirements of Bournemouth University
For the degree of Master by Research

November 2017

Department of Computing
Faculty of Science and Technology, Bournemouth University, UK

Copyright Statement

Abstract

Smart IoTs Based Urine Measurement System

Urine Measurement is one of the most important processes for diagnosis in the hospitals nowadays. Acute Kidney Injury (AKI) is usually diagnosed by taking patient’s urine samples for a specific period of time. It has been suggested that the average Urine Output of a patient depends upon his weight. As we are all aware that currently the means to monitor the major vital signs of the human body in the ICU (Intensive Care Unit) or various clinical settings such as Heart Rate, Blood Pressure, Central Pressure etc. is done by the means of a continuous recording of impulses and its digital display. It is utmost necessary to record and continuously monitor a patients’ fluid input, administered mostly by electronic devices (e.g. Syringe infusion pumps). At the same time, it is also important to monitor patients’ fluid outputs, in which, urine volume is one of the major components. Currently, it is obtained intermittently (per hour) from urine meters and urine collection bags, and a visual assessment is made and recorded manually relying heavily on the nurse's capability and skills. Therefore, even after so much technological advancement the measurement of urine output is literally the only critical parameter constantly recorded and monitored non-electronically by the medical staff.

The references from Medical Professionals at Royal Bournemouth Hospital clearly indicate a need for automated Urine Measurement System for efficient diagnosis process. There are automated devices for urine measurement, that are discussed in the Literature Review section, but none of them is available commercially. Some have cost issues whereas others are too complex to implement. We have found approx. 15 systems which have been patented by the inventors but none of them made it to the market. Cost-efficiency, complexity, and reliability are the issues we need to address, and we have tried to address in our project.

In this project, an integrated prototype based on IoT, that measures urine volume in real time for both high and low flow is developed. The system

measures the urine coming from the patient through two different sensors, Photo Interrupter Module and Hall-Effect based liquid sensor, and transmits that data to a cloud-based application via WiFi. The Arduino Yun micro-controller was used because of its built-in WiFi chip and more robust performance as compared to other options. The measurement of both high and low flow of liquid makes our system unique from the existing systems.

The application at Cloud analyze the data from the sensors for visualizations as mentioned by the doctors. MATLAB analytics facilities will be used because it provides extended options for multiple real-time visualizations. The data is sent in real-time, every 20 seconds and visualizations are updated accordingly. The data is also available to view on an Android App. The real-time stream of data on cloud and ease of data accessibility distinguishes our system to those described in the literature.

Series of experimentation was carried out for the prototype. Firstly, due to a problem in Photo Interrupter sensor for drop by drop measurement, the error was huge. Then, we developed an algorithm that solved the problem of object detection and then the error came to below 10% for both the high and low-flow measurement combined. This algorithm can be used to improve the working of photo interrupter sensor in other scenarios and it is one of the contributions of our project.

This system decreases the workload of the nursing staff as well as that of the doctors. The human-error is minimized. The Data Analytics application enables the doctors to have an in-depth understanding of the condition of a patient at several different intervals of time. Hence, our system is expected to benefit the medical industry and especially the staff at the hospitals. Lastly, we have also found our concept to be helpful in process industry also where the liquid measurement is used and we presented this concept at EPSRC conference in Glasgow.

Contents

List of Figures

Acknowledgments

First and Foremost, I’d like to thank Almighty God, the source of limitless love and countless blessings. I would also like to extend my gratitude to my Supervisor, Prof. Hongnian Yu for his support and mentorship throughout my project. I believe, under his supervision, I have learned a lot more than the insightfulness and drive to be what I want to be and to achieve what I want to achieve. Also, I would like to thank Dr. Simon McLaughlin for his insight on the possible challenges and requirements of the project. I would like to thank my friends in Bournemouth, I really had a great time with all of you and learned a lot from you. Dr. Shuang Cang also helped me during my project with insights and arranging versatile workshops to facilitate the research work and I want to thank her for that. I also want to thank my parents and siblings for always being there for me and support me through thick and thin. Last but not the least, I am thankful to be given the opportunity of studying on a project funded by FUSION.

CHAPTER 1

INTRODUCTION

Electronic and automated monitoring of different aspects of the human body is becoming very popular in the medical industry for the last decade. The rise in popularity and efficiency of Information Technology has had its impact on the medical industry (Sultan 2014). Although, in some situations, costs of treatment have increased substantially,, the increase in efficiency has neutralized this effect. Instead of spending hours of labor on manual readings, possibly reducing the accuracy in the process, doctors and medical professionals are adopting and developing partnerships with Information Technology (IT) industry (Devaraj et al. 2013).

Patients in Intensive Care Unit (ICU) need precise treatment and diagnosis should also be done very precisely and accurately. Measurements like Heart Rate are usually monitored in ICU to diagnose different diseases and to keep a close eye on patient's progress. The severity and complexity of the condition of the patient required the drastic change in the Automated Systems for Heart Rate Monitoring. So, technology was introduced and many systems have been developed for different situations (Jess et al. 2016). Another system developed by Jess et al. (2016) monitored heart rate using a non-contact device attached to chair where the patient's heart rate was continuously measured and sent to a remote location even when the patient was idle.

Problem Statement

Urine Measurement is one of the most important processes for diagnosis in the hospitals nowadays. Acute Kidney Injury (AKI) is usually diagnosed by taking patient's urine samples for a specific period of time (Parikh, 2005). It has been suggested by (De Melo Bezerra, Vaz Cunha, and Libório, 2013) that the average Urine Output of a patient depends upon his weight. Currently, the means to monitor the major vital signs of the human body in the ICU (Intensive Care Unit) or various clinical settings such as Heart Rate, Blood Pressure, Central Pressure etc. is done by the means of a continuous recording of impulses and its digital display (Hersch, Einav and Izbicki, 2009). It is utmost necessary to record and continuously monitor a patients' fluid input, administered mostly by electronic devices (e.g. Syringe infusion pumps). At the same time, it is also important to monitor patients' fluid outputs, in which, urine volume is one of the major components. Currently, it is obtained intermittently (per hour) from urine meters and urine collection bags, and a visual assessment is made and recorded manually relying heavily on the nurse's capability and skills (Hersch, Einav and Izbicki, 2009). Therefore, even after so much technological advancement the measurement of urine output is literally the only critical parameter constantly recorded and monitored non-electronically by the medical staff (Otero et al, 2009).

According to (Hersch, Einav and Izbicki, 2009), if we consider a situation where data is collected electronically then there would be an immediate, accurate and efficient assessment of the patients' medical condition. There are also several additional benefits such as efficient storage of information and access to that information later. The manual devices and technique which are used to measure urine output today only provides a rough estimate. Moreover, such a method requires constant nursing attention, management, and handling difficulties. This whole process is very time-consuming as well as laboring for the medical staff who are required to be in constant contact with the devices such as recording the readings etc. Programmed and persistent volume-flow recording of urine output may decrease human error, spare costly medical staff time, and give an early alarm to approaching kidney or organ failure in an intensive complex care condition (Grover and Barney, 2003).

A group of specialists in Kidney disorders recognized the importance of urinary output in the detection of a renal failure. The severity grade of kidney dysfunction depends upon the level of renal injury which is measured by serum creatinine or urinary output or both (Bellomo *et al.*, 2004). Now, the interesting thing to note is that serum creatinine levels are seen only when at least 50 percent kidney function is impaired. But if this parameter is combined with urine output then there is a chance to identify patients earlier who are at risk of renal failure (Galley, 2000). This may save many lives and increase the efficiency of early detection of kidney failure.

Kidney dysfunction is seen in 9-40 percent in patients with sepsis, among which the mortality rate is >50 percent. There are various clinical manifestations commonly seen in sepsis such as hypotension, hypovolemia etc. But our focus for this thesis are conditions relying on monitoring of urinary output. Thus, we see that renal hypo perfusion is a leading cause of Acute Renal Failure (ARF). A rise in a number of cases of Sepsis or Septic shock is seen in the last few decades. Factors assigned to this phenomenon could be an increasing number of immunocompromised patients, diabetes, alcoholism, malignancies and Chronic Kidney Disease. After a profound volume loss, the patient suffers from hypotension and start developing oliguria and peripheral cyanosis. It is crucial at this stage to monitor urine output efficiently so that prevention towards progression to MODS (Multi-Organ Dysfunction Syndrome) can at least be attempted (Molitoris, 2015).

Even the mortality rates reported for critically ill children is 35-73% requiring dialysis. It has been suggested that both accurate measurements of fluid input and output may lead to improved outcomes (Ronco *et al.*, 2010). Similarly, in postsurgical cases of Acute Kidney Injury (AKI), a higher urinary output has to be maintained using diuretics which needs to be continuously monitored (Ronco *et al.*, 2010). If a person undergoes Renal Replacement Therapy (RRT), it is important to consider the urinary output which could be 1) Oliguria (urine output <200 ml/12 hours) or 2) Anuria (urine output <50 ml/12 hours) (Murray, Brady and Hall, 2001). Similarly, for the cessation of RRT in Intensive Care Unit, it is important to note that Urine output averages 1 ml/kg/h over a 24-hour period (Murray, Brady and Hall, 2001). It is important to emphasize here that on both occasions an electronic urine output monitoring device could revolutionize the treatment protocol.

Urine output has a prognostic value in patients with AKI. The general method to monitor urine output as stated above by visual readings of the amount of urine accumulated is often inaccurate and provides limited data (MacEdo *et al.*, 2011). Identification of decreased urine output with the help of a real-time integrated monitoring tool could greatly improve management of critically ill patients (MacEdo *et al.*, 2011). In an ICU, the fluid gains or losses in a postoperative setting along with total parenteral nutrition need to be recorded and quantified. The main focus is hemodynamic changes, urinary output, volume assessment of fluid balance and medications administered (Molitoris, 2015).

We visited Royal Bournemouth Hospital, our industry partner, to further investigate the problems faced by medical staff related to the urine measurement systems. Dr. Simon McLaughlin briefed us about the challenges they are facing when it comes to noting down the readings from a scale of the urine container or from a small LCD screen that shows the measurements of the patient's urine output. After discussions with Dr Simon and the nursing staff, we found out that apart from the extra burden on the nursing staff, there is always a probability of human error in the readings. Dr. Simon also pointed out that they need to measure the urine flow as low as 15-20ml/hour. We also need to tackle the high flow which could be 100-150ml/hour or higher.

The above references from Medical Professionals clearly indicate a need for automated Urine Measurement System for efficient diagnosis process. There are automated devices for urine measurement, which are discussed in the next chapter, but none of them is available commercially. Some have cost issues whereas others are too complex to implement. We have found approx. 15 systems which have been patented by the inventors but none of them made it to the market. Cost-efficiency, complexity, and reliability are the issues we need to address and we will try to address in our project.

Aims and Objectives

The aim of this study is to investigate and propose an efficient solution for measuring the urine-filled status by minimizing the cost of the hardware and improving the data analysis for equipping the doctors with real-time information of the patient's condition. Another thing which we will try to investigate is the implementation of our model in an industrial environment for liquid monitoring such as process industry. For this purpose, we will add the high flow of liquid to our measurements as well. To achieve our aims, we have outlined the following objectives:

OB1: Study the current work (Patents, Papers, Commercial work): We have studied the current literature corresponding to our goal. We have studied the devices that have been claimed in the form of patents for measuring the urine output.

Furthermore, we have also highlighted the problems in each device and the solutions to overcome those problems in our scenario.

OB2: Design of Prototype: Initial Design of the prototype will be developed on the computer to simulate the working of our actual prototype. This will help us in selecting the hardware we will need, provide us the blueprint of the device the efficient integration of that hardware for developing the actual physical prototype.

OB3: Select hardware and software: After the design phase, we will select and purchase the hardware that we will use in developing the prototype. Moreover, software architecture will also be developed which will be the basis of the application that will be used in collaboration with the hardware to develop and efficient system.

OB4: Develop a prototype for Real-Time Urine measurement based on IoT: In this stage, we will actually develop the prototype device and the software application will also be developed.

OB5: Conduct validation and verification: We will perform experimentation in different scenarios firstly at our end before going into the actual real-world scenario and if needed, improvements will be made in the system.

OB6: Facilitate the doctors with interactive data visualizations: Data Analytics like Graphs and data visualization will be added to the system to expand the horizons of the system.

Research Methodology

Usually, there are two types of Research Methodologies used by researchers to complete a specific research objective, namely Qualitative (Silverman, 2016) and Quantitative (Hoy and Adams, 2015). There is also a hybrid method called a mixed method of Research in which a researcher collects the data, analyses it, visualizes it using the combination of Qualitative and Quantitative methods. Specifically, the mixed method approach refers to the collection of data from the literature called literature review. The challenges in the literature are highlighted and the solutions to overcome the challenges are proposed. After that, the proposed solution is tested with the help of experiments and then the results are compared with the literature to validate our proposed solution. We will use mixed methods approach in our research.

After analyzing the challenges, aims, and objectives, we have learned that our research poses the following research questions:

- What are the requirements of the user related to Urine Measurement System?
- What are the current practices in this environment?
- How are we going to design the prototype to minimize the complexity and to improve Human-Computer Interaction?
- What is the approach for testing the prototype?

Exploring Research Database

For studying the existing Literature (described in detail in the 2nd chapter), there are different methods used by researchers. Some people use Snowballing method (Badampudi, Wohlin and Petersen, 2015) for systematic reviews. We used the traditional method of literature review in which we elaborate the features and the possible weak points of a study or system We used search string method (Stol, Ralph and Fitzgerald, 2016) and the famous Journals like ACM, ScienceDirect etc were searched. The main search tool used was Google Scholar. We will describe the search string criteria below:

Search String The search string developed for searching Google Scholar Database is: ((“Urine OR Liquid OR Fluid) AND (Monitoring OR Measuring OR Output) AND (System OR Device”)). Based on this string, we collected relevant papers for the literature review.

Inclusion and Exclusion Criteria After the search string, we were given a lot of papers by Google Scholars. Now, to find papers that were closely related to our project, we developed inclusion and exclusion criterion. For including a paper to our own database, we followed the following criterion*:*

- The paper is explicitly about the system designed to monitor urine output in hospitals.
- The paper proposes a system to measure liquid output in real time.
- The paper proposes an integrated system for any fluid monitoring and analysis.

We also developed an exclusion criterion to filter out the irrelevant papers, which are described below:

- The paper was published before 2000.
- The paper was not relevant to liquid measurement.

The structure of our thesis is as follows:

Chapter 2 describes the extensive literature review of the technologies, sensors, and systems that have been developed an used to measure urine output in real-time. We have highlighted the pros and cons of all the systems and model and we have also considered the commercial implications of the systems that have been proposed in the current literature.

Chapter 3, depicts the prototype design, both hardware, and software, for development of the integrated system for urine measurement. We have described in detail the challenges we had to face in hardware selection and interfacing hardware and software.

Chapter 4 has the details of the development of the prototype in detail as well as the software design for data analysis.

Chapter 5 depicts the results of our prototype development and the improvements that we have made to tackle some problems that we have faced during the testing of the prototype.

Chapter 6 concludes our work and provides further suggestions to improve the system and also provide the industrial implications of our work.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we will critically evaluate the current systems and literature proposed for Urine Measurement Systems. We evaluate the models and systems based on their performance in real time. Furthermore, we also indicate whether the hardware used in the system is commercially available or not. In this way, we can find out if there are commercial and market potential in the model or not. It is worth noting here that none of the proposed models recently for urine measurement system is available as a commercial device. Finally, we categorize the models into different categories based on the techniques and hardware used. Detailed Taxonomy is depicted in Figure 1.

Anti-reflux Based Systems

A urine measurement system developed by (Chelsey Fontaine, Stephen Tully, 2013) used an anti-reflux mechanism (see Figure 1) to measure the urine output. It used a urine bag and a urine meter connected to a catheter or a urine input mechanism, protected by a shield in the middle which prevents the flow of excessive urine from the urine bag to the urine meter. However, this system measures the urine correctly to some extent, it is still time-consuming and complex in nature. It also still requires manual measurements to be taken.

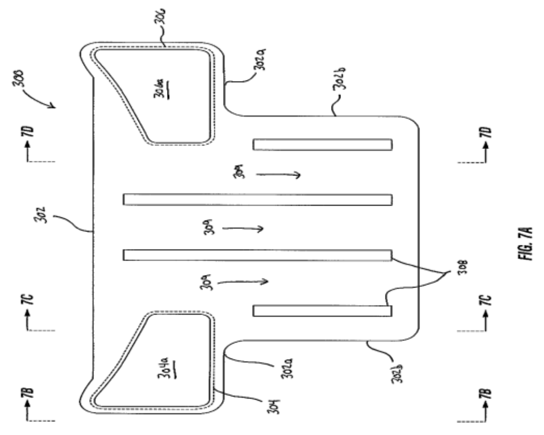


Figure 1 Anti-Reflux Based Mechanism((Chelsey Fontaine, Stephen Tully, 2013)

A manual urine measuring device invented by (Sippel, Collin and Voges, 2000) used a holder which is fixed to the bed of the patient. The holder has a measuring container and a urine bag attached to it. The liquid comes from the catheter to the liquid container and when it is filled completely or up to the desired level, it is then held at a specific axis so that the container is emptied into the urine bag. However, it is comparatively easy to handle but it requires more operation by the hospital staff apart from noting down the readings, which could result in inefficient measurements.

(Voges, Sippel and Collin, 2002) developed a device to measure urine flow, consisting of a dripping chamber and a measuring chamber. A tube, glued to the dripping chamber assists in pouring the urine from the catheter into the dripping chamber. To prevent solid material from entering the measuring chamber, an anti-reflux filter was used. Although this system for measuring the urine output is inexpensive to build and maintain, it cannot be used with modern micro-controller approach and does not ease the workload of the staff.

Ultrasonic Based Systems

Benign Prostatic Hyperplasia (BPH) is a disease in which the patient experiences symptoms like Low-Urine Frequency, weak stream or prolonged periods of no urine excretion (Chughtai *et al.*, 2016). To diagnose this disease and for treatment, Urine Output Analysis is necessary. An integrated system to measure and analyze the flow of urine was invented by (PAULSEN *et al.*, 2016) with three different parts named as Recording Unit, Analysing Unit and a Transmission Unit (see Figure 2). The recording unit consists of sensors (Acoustic and Camera) connected to an incoming urine catheter whereas analyzing unit consists of hardware and a software that helps in analyzing the recorded data on different parameters. Finally, the transmission unit may consist of wireless or wired communication to transmit the recorded data to a remote PDA. Although, this is an integrated system that can increase efficiency use of acoustic sensor and an optical device might affect the recorded data under certain conditions of the noisy area.

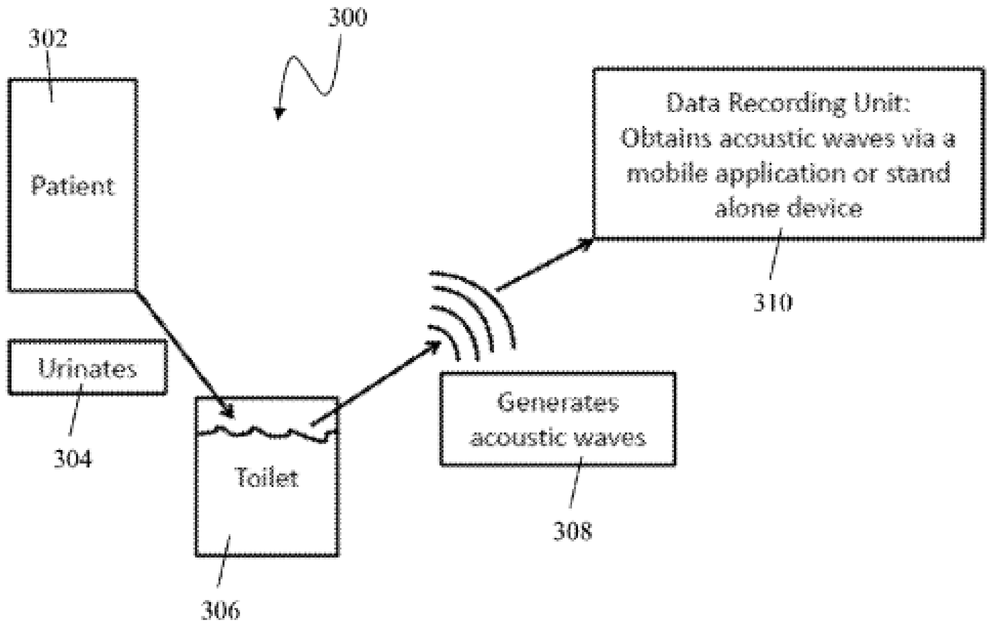

300
302
Patient
Urinates
304
Toilet
306
Generates acoustic waves
308
Data Recording Unit: Obtains acoustic waves via a mobile application or stand alone device
310

Figure 2 Ultrasonic Based System ((PAULSEN *et al.*, 2016)

A device to measure low-flow of urine was presented by (Ilan Paz, Gush EtZion (IL); Harold Jacob, Cederhust, 2003) with the mechanism of drop by drop counting for accurate measurement. Moreover, their system also had a mechanism of correction if the drop size varies during the measurement. This correction made their system more reliable. However, their device was standalone and could not be moved to another location if the patient wants to move. Furthermore, there was no solution to centralized data access which makes this system not suitable for the fully integrated environment.

Modern Micro-Controller Based Systems

Otero et al. ((Otero, A.Panigrahi B, Palacios F, Akhinfiev, T, R, 2009) developed a fully integrated Urine Measurement System (see Figure 3) which consists two containers, a small one of 15ml for precise measurement and a larger one of 60ml long-term measurements. Each container was equipped with a float sensor which was controlled by a Micro-Controller (Atmel AT89S52). With the help of Physicians, hourly goals were set for a specific weighted patient and alarms were generated if goals were not met. The software, developed in Java, controlled the data and generated relevant alarms. This model was one of the very first which had practical implications suitable for the medical environment but due to some legal reasons, this device never went into the manufacturing industry.

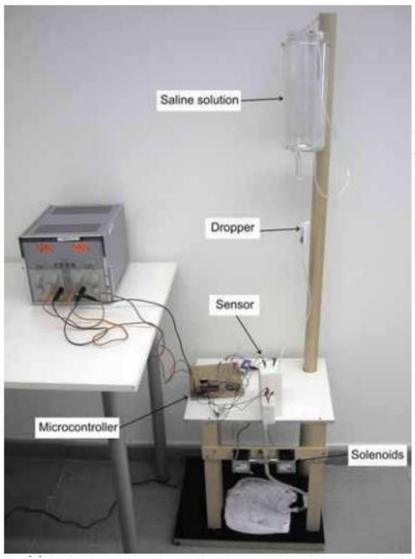


Figure 3 Modern Urine Measurement System (*Otero et al. (*(Otero, A.Panigrahi B, Palacios F, Akhinfiev, T, R, 2009)

The Siphon Principle is used to transfer the water or any liquid from a high-pressure area to a low-pressure area (Garrett, 1991). Using the Siphon Principle, Otero et al. (Otero *et al.*, 2010) developed another device that consists of two containers. To empty the container, the siphon principle was used. Reed switches were attached to the walls of the containers. The measurement was recorded when the container was being emptied and the liquid started to flow through the siphon mechanism and till the time the container was completely empty. Like their previous prototype, a micro-controller controlled the liquid sensor and the reed switches and sent the data via Bluetooth to a remote PC where it was analyzed using the java based software. Although the system worked fine, the complexity issue of the container design, the output tube size and the connection between the two containers in some conditions were found to be problematic. Due to these problems, manufacturing the device using the siphon principle became almost impossible.

A fully automated, Internet of Things (IoT) based urine measurement system was developed in (*Atigorn*, 2016) in which they have used two different approaches to measure the Urine Output. Firstly, they used three different sensors (Ultrasonic, Photo Interrupter, and Microphone based sensor) for measuring the urine from the patient and they deduced that Photo Interrupter sensor produced the most accurate measurements. Furthermore, they used only Photo Interrupter in the second embodiment for measuring the urine in the drop by drop fashion. It is worth noting that unlike most of the previous literature we have discussed, this system used modern micro-controller Arduino and used the wireless medium to send the data to a remote server where it was analyzed for the Physicians to understand.

A real-time urine measurement system developed in (Greenwald, 2012) measured both high and low flow of urine simultaneously. This system had a processor to store and transmit the data. Moreover, the inventors used the Radio Frequency (RF) based sensor for low-flow measurement. However, due to non-availability of RF-based sensors for low-flow measurement commercially, the commercial implications are quite low for this device. Furthermore, the external conditions like sound waves may hinder or change the measurements due to RF-based sensors.

Capacitance Based Systems

Capacitance-based measurement methods (see Figure 4) for urine were presented in (Ramos, O’Grady and Chen, 2016) in which they claimed various systems to measure the urine flow in real time. According to their claims, they used a liquid container with which a capacitance sensor has the inter-digital electrode structure. The data from the capacitance sensor transmitted through a wireless medium via micro-controller reached the software where it was analyzed remotely. The complexity of design is the only disadvantage of this system and may hinder the manufacturing of this device on a commercial scale.

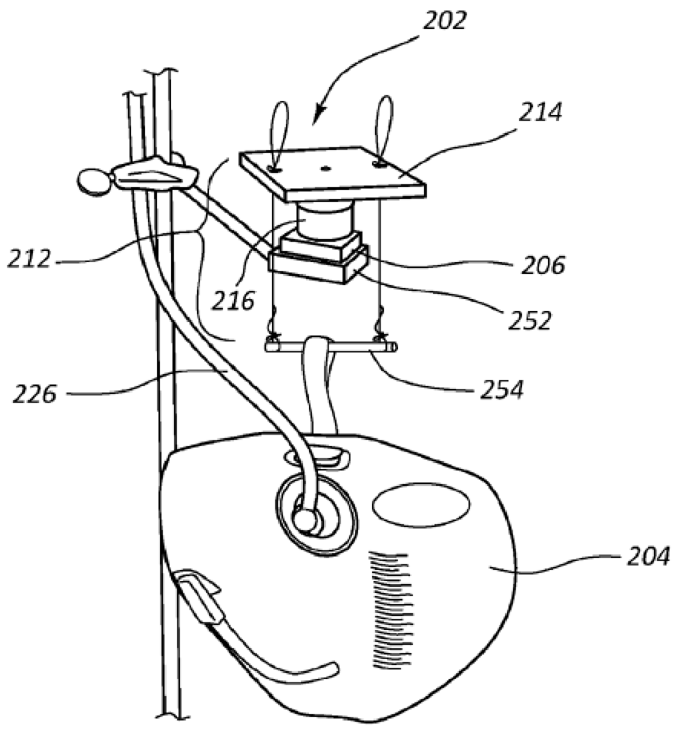


Figure 4 Capacitance Based System ((Ramos, O’Grady and Chen, 2016)

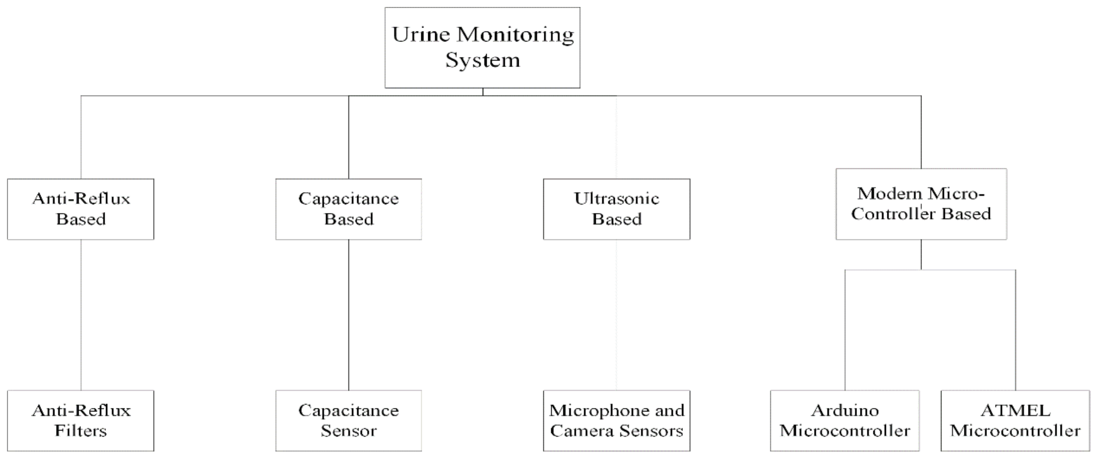


Figure 5 Taxonomy of Urine Measurement Systems

The main problems in the systems described in the literature are as follows:

- A standalone device that restricts data collection during movement of the patient for tests within the hospital.
- Complex hardware that is difficult to commercialize.

- Inappropriate hardware that might pose health risks to patients
- In-accurate readings that may hinder the diagnosis process.
- Sophisticated hardware that is not commercially available.
- Poor Data analysis application that makes it difficult for nursing staff and physicians to understand.

Challenges in Developing a Urine Measurement System

From the above literature, we have identified some challenges that are listed below. Some of them are explicitly discussed in the literature and the others have been identified by us.

- *Data Privacy*: As highlighted earlier, most modern systems for Urine Flow Measurement will be based on IoT, so, data privacy is one of the biggest challenges. The data, in this case, is critical and the security measures for Data Security are more important than ever. However, when addressing this concern, there is a possibility that you must compromise in the robustness of the whole system which could result in a substantial difference in measurements of the liquid.
- *Hardware Selection*: Selecting a hardware that is simple and effective is the most challenging part of developing the automatic and integrated urine monitoring system. There are many sensors that can do the job of collecting the data from the liquid input and send it to the remote computer for processing but keeping in consideration the conditions and surroundings of the hospital or medical center, selecting the most feasible and accurate is the utmost challenge. For example, an ultrasonic sensor can be used to measure the urine

output but, in the hospital, where the readings could be affected by the surroundings, it is neither feasible nor accurate whereas the Photo Interrupter Sensor is found to be more accurate in these conditions.

- *Software Design*: Software Design is also an integral part of the Urine Measurement System. The staff who are going to use your system must be consulted before developing the software. The Programming Language or platform must be chosen with which the staff is comfortable with and it does not require any extra skills to operate. The system should be simple and accurate.
- *Ethical Challenges*: To develop a System for a hospital, the stakeholders include patients, doctors and nursing staff, so, the data is both sensitive and critical. This poses the challenge of managing the data accurately and securely. Sending and storing the data using secure medium is one of the challenges that are faced, both ethically and technically. It should be made sure that the networking strategies used in the devices where this data will be stored should also be secured.
- *Health and Safety Issues*: The primary goal of the automation of the urine measurement system is to make it is easier for the Hospital Staff to measure the urine output. But, in this case, patient's health is at stake. It should be ensured that the data that is recorded and sent is accurate so the decisions made by a doctor for the treatment of the patient may not be false because it can be dangerous for the patient. Moreover, the system that is going to be built should not have any contact with patient's body so that any harmful substances in the hardware do not affect the patient's condition. These are the health and safety

challenges that one may have to deal when they are developing this kind of systems.

Reference	Method Used	Technique Used	Novelty	Issues
(Chelsey Fontaine, Stephen Tully, 2013)	A Shield is used to prevent excessive flow to the measurement container.	Anti-Reflux Mechanism	Prevention Mechanism for Liquid Overflow.	Complex Hardware Orientation, Laborious.
(Sippel, Collin and Voges, 2000)	Liquid Container fills up to the desired level and it needs to be held at an axis to empty it.	Anti-Reflux Mechanism	A measurement after a desired level of liquid is met.	Staff must note the readings and empty the container.
(Voges, Sippel and Collin, 2002)	A tube is used to move liquid to the dripping chamber and an anti-reflux filter is used to prevent solids from entering the chamber.	Anti-Reflux Mechanism	Separate containers for liquid input and measurement.	Inability to be used with modern micro-controllers, laborious for staff.
(PAULSEN *et al.*, 2016)	Hardware is divided into 3 units named recording, analyzing and transmission.	Ultrasonic Mechanism	Separate Units for Recording, Measuring and Analysing the Data, Modern Micro-Controllers and PDAs are used.	Due to Ultrasonic and Acoustic Sensors, measurements might be affected under noisy conditions.
Otero et al. ((Otero, A.Panigrahi B, Palacios F,	Modern Micro-controller used to control float sensors	Float Sensors Mechanism	Modern Micro-Controllers along with modern programming language	Due to some legal issues listed in the paper, the

Akhinfiev, T, R, 2009)	attached to the containers.		was used, Staff workload was minimized.	device never was commercialized.
Otero et al. (Otero *et al.*, 2010)	The liquid was measured when the container was being emptied using siphon principle. Analysed using Java language.	Siphon Principle	The liquid was measured by modern sensors using siphon principle and modern Bluetooth technology was used to transmit the data to the remote computer.	The complex design of containers and integration of Reed Switches made the design impossible to be commercialized.
Atigorn (Otero *et al.*, 2010)	The liquid was measured drop by drop and the total liquid was calculated by the number of drops. Average Drop volume was calculated.	Modern Micro-Controller Mechanism	The modern approach of Arduino Micro-controller along with commercially available sensors was used. Proper requirements from Physicians were taken into consideration and it was tested in a real scenario	Software glitches and hardware orientation were notable issues which would alter the measurements in critical conditions.
(Greenwald, 2012)	Radio Frequency based sensors were used and drop counting mechanism was used.	Low-Flow Measurement	Drop-by-drop measurement, correction mechanism if the size of drop varies.	Standalone device, not portable, no centralized access to data.

Table 1 Critical Review of Urine Measurement Systems

The above table shows the extensive findings of the literature review in a brief form. It is worth noting here that none of the systems in the literature is available commercially. They are either complex, expensive or have healthcare risks attached to it. The main challenge is to simplify the process of urine measurement by using cost-effective and felicitous technologies which we will try to address in this project.

Internet of Things (IoT) is a concept that envisages the connectivity between daily life things by using different types of sensors like Radio-frequency identification (RFID) (Sun, 2012), actuators that work collaboratively to sense, collect and transmit important information from their surroundings on to the Internet. IoT is a term that envisions connectivity between physical and digital world by using felicitous technologies (Miorandi *et al.*, 2012). IoT has been one of the hot topics in the technology domain for the last few years and it is expected to revolutionize the world similar to that the Internet itself did (Chase, 2013). Frost & Sullivan (2011) projected the increase in RFID sales over the years and it is going to increase exponentially in the next few years. If the predictions are even closely accurate, then the energy consumption concerns are going to arise because Active RFIDs (Zhang *et al.*, 2014) need battery powered energy and to handle this issue we need to make the IoT technology Green by implementing various strategies. Some of them are discussed in the later sections. It could be observed that the number of Internet-connected devices is growing at a very fast pace. The mechanism of IoT consists of several elements such as Identification, sensing, communication, computation, services, and semantics. Identification is the most important one as it ensures that the required data or service reaches to the correct address. Sensing deals with the collection of the information from different resources and this information is then sent to data centers. This data is then analyzed using different conditions and parameters for the purpose of various services. The sensors can be used to collect humidity, temperature etc. Communication in IoT performs the combination of heterogeneous objects to offer specific services. Communication is usually performed by using Wi-Fi, Bluetooth etc. Computation is performed by different microcontrollers, microprocessors, Field Programmable gate arrays and many software applications. Services can be related to identity, information aggregation, collaborative or ubiquitous. Lastly, Semantics deals with the intelligent knowledge gathering to make decisions (Zhu *et al.*, 2015).

Now, we are going to discuss the techniques that can be used to make IoT a green technology. This research is made part of this thesis because the use of IoT on a large scale can have a negative impact on the environment. Therefore, introducing Green IoT is the need of the hour. The conceptual model described in the Future Work Section, which can be used on a large scale, needs to integrate Green IoT.

To make the IoT green, there is a need to study more state-of-the-art techniques and strategies that can fulfill the energy hunger of billions of devices. Here, we aim to provide a comprehensive overview of energy saving practices and strategies for the green IoT. We consider a case study of smart phones to show that how different stakeholders can play their roles for the green IoT.

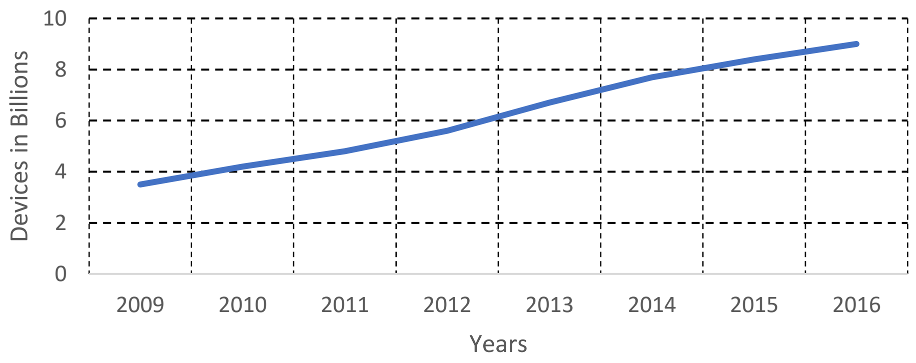


Figure 6 IoT Devices Projections

IoT Trends

The current era is considered to be fully Internet-based. Our dependence on the Internet and the devices is rapidly increasing. How does IoT influence in routine things? This is the main question to be addressed in the subsequent section.

Applications of IoT

IoT is revolutionizing our daily life activities by tracking different scenarios and making intelligent decisions to improve our lifestyle and to protect our environment. There are numerous applications of IoT in daily life. We explore several of them below

- *Smart Homes*: As described by (Li, Xu and Zhao, 2015), by equipping our home or office with the IoT technologies like RFIDs, we can track the activities of in-habitants in the building and can make decisions that can save energy, money and whole environment in the process. For example, a smart fridge can have RFIDs on every item inside it and we can decide when to go shopping and what we need to buy on the basis of information provided by the sensors attached on the items.
- *Food Supply Chains (FSC)*: IoT can have a huge impact on the business industry. Using IoT technologies, vendors can track the production of their products from the farm to the end users. A framework for such an application is proposed by (Pang *et al.*, 2015). It proposes a Business-oriented model of IoT for FSC which can enhance food security and can be used to collect the data related to production processes and that data can be manipulated to make better decisions regarding the business process model.
- *IoT in Mining Industry:* IoT technology can be used to ensure safety for miners and can provide Mining Companies with important information regarding mining process which can help them in enhancing the current practices (Xu, He and Li, 2014). RFIDs, Wi-Fi, and sensors can be deployed to improve communication between miners and their employers. Furthermore, diagnosis of different diseases in miners can be done by collecting symptoms using these sensors.
- *IoT in Transportation:* IoT is revolutionary in the Transportation and Logistics industry. We can track vehicles and products using RFIDs and sensors from source

to destination in real-time. A DNS architecture (Xu, He and Li, 2014) is developed for IoT where large-scale operations enhance the capabilities of IoT in supply chain management.

- *IoT in Garments:* A new type of E-Thread (KIOURTI, Lee and Volakis, 2015) envisions the idea of collecting data from clothes. This can help in collecting real-time data to track the activities of a patient without using any extra device.
- *Smart Cities:* One of the most scintillating and emerging applications for IoT is Smart Cities (Caragliu, Del Bo and Nijkamp, 2011) which has gained popularity in the last few years. A smart city is a combination of different smart domains like Smart Transportation, Smart Energy Saving Mechanism, Smart Security (Zanella *et al.*, 2014) and many more which provide the users with latest technological facilities all under one umbrella.

Challenges of IoT

IoT is at its cutting edge and could prove to be revolutionary in the IT industry, but everything comes at a cost. There are many challenges posed by the IoT technologies like Security and Privacy challenges as described by (Weber, 2010), as one of the key areas that experts need to work on in order to gain the trust of the users (Singh and Jara, 2014). The cited paper described that the RFID tags can follow a person without this consent or information and this could lead to a very serious distrust among the people. However, the most significant challenge that we will face in the implementation of IoT will be energy. It has been predicted by National Intelligence Council of US that by 2025, daily life objects such as food items, pens etc. will be a part of the Internet. This means that there could be billions of devices connected to the Internet.

Paper	**Method**	**Novelty**	**Issues**	Application
(Pang *et al.*, 2015)**.**	Installing RFIDs	Tracking of Food Products	Data Privacy can be violated	Food Supply Chain
(Xu, He and Li, 2014)	Deploying Sensors to track miners and activities	Activity Tracking and reporting	Due to extreme situations, data might be delayed in communication	Mining Industry
(Xu, He and Li, 2014)	RFIDs Installation	Keeping Track of transportation and goods	Possible High energy consumption	Transportation Industry
(KIOURTI, Lee and Volakis, 2015)	RFIDs and other wearable's installation	Keeping track of Garment Production	Some garments might be contaminated by sensors' material	Garments Industry

Table 2 Applications of IoT

According to (Lee, Kim and Kim, 2014), each active RFID needs a small amount of power to operate depending on its functionality and active RFIDs are necessary for the efficient services. Therefore, imagine billions of such devices consuming energy on daily basis and millions of GBs of data transmitted by the sensors needs to best or edited by huge Data Centers thus huge processing and analytics capabilities are needed (Tsai *et al.*, 2014)(Mukherjee *et al.*, 2014), which consume a lot of energy resources, and to further deepen the crisis, we are running short of traditional energy sources. Moreover, emission of CO_2 due to ICT products is increasing rapidly which is damaging our environment (Gelenbe and Caseau, 2015) and it is projected to do so if sufficient measures are not taken to address this concern. To solve these critical problems, the Green IoT is an important topic.

Discussion

In this chapter, we have tried to identify the challenges in the current systems used for Urine Measurement. In Table 1, it is very clear that the systems that have been claimed so far are either very complex in nature, are inaccurate, not suitable for hospital environment or all of these. Furthermore, the challenges that we have listed in this chapter clearly depicts the need of a comprehensive research that should be carried out in determining the suitable hardware, software and the environment in which the systems should be developed. Particularly, if we look into the complexity issue, which is the biggest issue, need to be solved because the hospital staff is not trained and equipped enough with the knowledge of ICT and other electronic devices. Therefore, a team of diverse experts needs to collaborate to solve these issues to develop a modern technologies based urine measurement system.

CHAPTER 3

PROTOTYPE DESIGN

In this chapter, we will describe in detail our System design for the IoTs based Urine Measurement System. Moreover, we will also describe in detail the challenges we faced during the design of the system. It is worth noting here that this system is just used to depict our proof of concept and it is kept simple to avoid any manufacturing challenges. But firstly, we will glance through the basics of Prototyping and how it is beneficial for depicting a proof of concept.

Introduction to Prototyping

Prototyping is a widely used method to depict the concept or a system without having to build the entire system. We chose to adopt the prototyping because of its ease, robustness and fast development to have detailed information about the user's requirements. Another reason why we used prototype is that the user does not know the exact requirements and possible implications of different scenarios in the beginning. So, by prototyping, we can easily test the initial requirements and proceed with the further development based on the initial results.

(Cho, Chung and others, 2007) seconds our argument and further emphasizes that instead of developing the whole system first with high costs and valuable equipment and then test it with possibilities of changing the whole system altogether, we should go for the prototyping which is much cheaper and fast to develop. The problem in hand was developing an automated urine measurement system for which we chose to use the cheap off-the-shelf sensors to develop a prototype and to depict our proof of concept.

System Design

In this section, we propose a model of the system that is used to measure the Urine flow in real-time. Our proposed model is depicted in Figure 7. In the proposed system design, there are four layers; The Physical Layer consists of two sensors, one measures high-flow of liquid and the other measures low-flow. The next layer is Data Layer which deals with the data filtration. Moreover, the data communication from the sensors is done in communication layer and the data analysis and representation is done in the information layer.

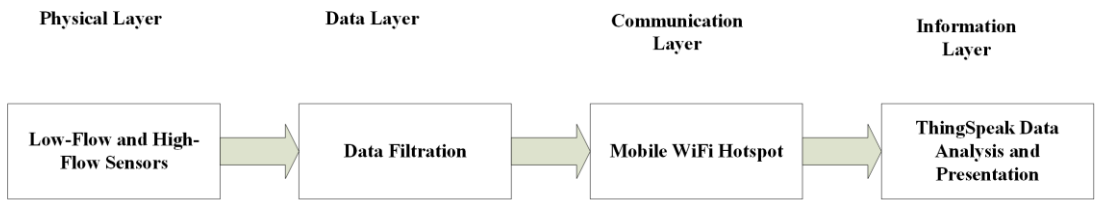


Figure 7 System Design Model

Design of Prototype

The design of the prototype is a part of the physical layer of our system design. To measure the urine output, we first evaluated our design strategy. In that process, we evaluated different sensors that we can use to measure the urine. During our research, we found that there are very few sensors available that could serve our purpose and develop a new sensor was out of the scope of our time-frame. As we have to measure both high and low flow of urine, we opted for the following sensors:

- For high-flow, we selected Hall-effect based sensor, which generates 9.5 pulses per second per ml of liquid.

- For low-flow, we selected Photo Interrupter Module, which will be attached with a dropper to measure each drop passing through it.

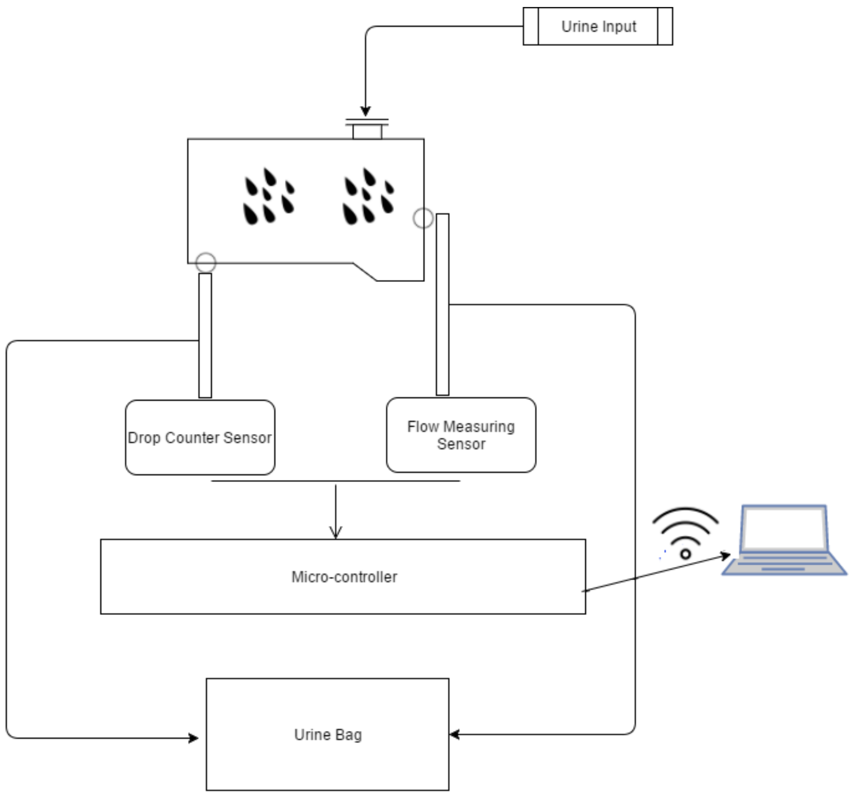


Figure 8 Prototype design

Our prototype design can be seen in Figure 8. The main idea behind the design is that the urine input will come from the patient via a catheter into the urine container (could be a normal bottle). Then, there are two sensors attached. The low-flow sensor is attached with a dropper (attached to the urine container) which measures the drop by drop flow of water and counts the number of drops, through which we can count the amount of liquid passed through it. Another sensor, which measures the high-flow of urine, is attached on the lower side of the bottle. If the liquid is of high-flow then it will reach the high-flow sensor quickly and it will get measured from that and if the flow is slow, drop counter sensor will measure it. In the end, we can combine both the measurements and we will get the total measurement of the urine. This prototype design was chosen to minimize the manufacturing challenges and to depict our proof of concept and novelty in a better way. The fact that our device can measure both high and low-flow of liquid, distinguishes it from the literature.

Once we get the data from the sensors, Arduino Micro-Controller will send that data through Bluetooth or Wi-Fi to a cloud server, where it will get analyzed in the form of graphs. The graphical representation will be hourly, 8 hourly and 24 hourly. The doctor can analyze how much urine a patient has produced in the last day and at any specific hours or a number of hours. In this way, we can eliminate the human error and can relieve the burden of the hospital staff. The complete system architecture can be seen in Figure 9.

The process of data collection and visualization depicted in figure 9 has three main parts: Data Read, Data Transmit, and Data Visualization. The combination of Arduino, Sensors, and Libraries help us in data collection. Then, using ThingSpeak Library for Arduino, the collected data is sent through wifi to the Public Cloud. Finally, the built-in service of ThingSpeak aided by our MATLAB coding analyses the data in the form of graphs.

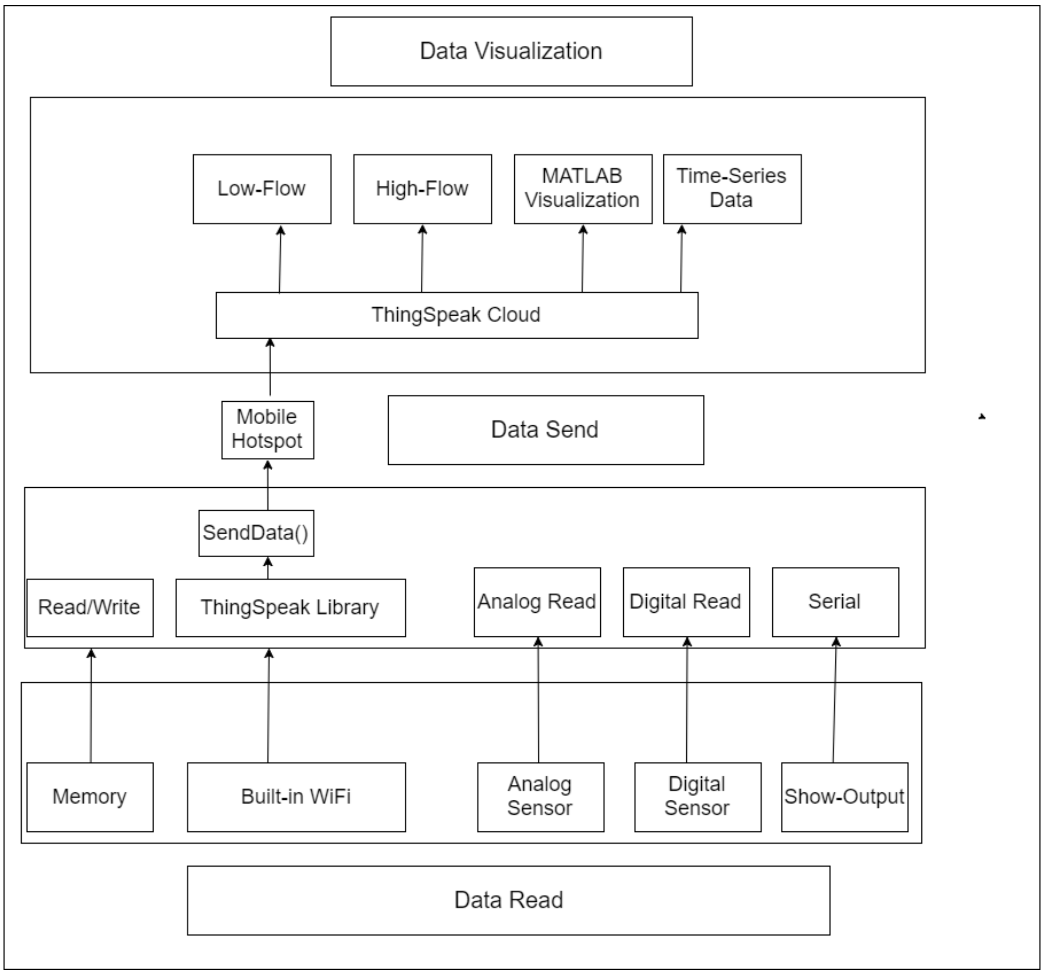


Figure 9 Architecture of The System

Internet of Things and our Prototype Design

Internet of Things (IoT) is defined by Rushan et al. (2017) as the connection of daily life objects to the Internet and the ability of 'things' to talk and listen to and from the digital world. Our prototype design is based on the similar concept. In our case, a urine container has sensors attached to it and these sensors then transmit the data from physical objects to the digital world via a transmission medium such as WIFI. There is another part of the IoT in which the sensors, actuators communicate with each other but in our case, it is not necessary. The future implication of our prototype could use this concept as more than two devices are connected with each other and they can communicate the data between each other.

Micro-Controller

The sensors are connected to a micro-controller which is responsible for transmitting the readings on to the cloud server. Generally, there are two most famous types or brands of Micro-controllers; Raspberry Pi and Arduino. Both are open source and have huge support in form of individuals and volunteers. In our project, we used Arduino Micro-Controller (depicted in Figure 10). There are different models of Arduino available in the market. Arduino Uno, Arduino Genuino, and Arduino Yun are the most famous ones.

Arduino Yun has some advantages over the other models such as built-in Wi-Fi module which lets us connect to any server through a local Wi-Fi network. These are the reasons we preferred Arduino Yun over others.

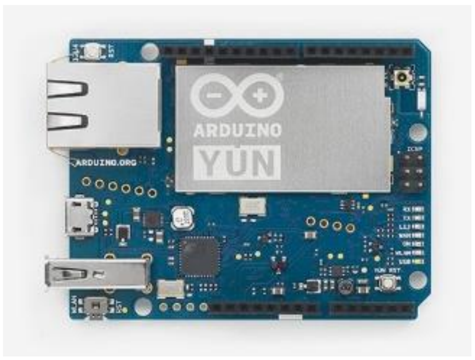


Figure 10 Arduino Yun Micro-Controller

Arduino Yun is based on Atheros processor which supports a Linux distribution known as Lininio OS. The support of Linux makes Arduino Yun a powerful device. Some experts say it's a whole new machine itself that can perform various tasks, even can be used as a server machine. It has built-in WIFI and ethernet support and has several Digital and Analog Pins to connect sensors and devices. It can be powered by either USB or 4-6 AA batteries can also be used. The complete setup guide will be provided later in this chapter.

WIFI Configuration of Arduino:

To transmit the data from sensors to a public cloud via WIFI using Arduino Yun, we first have to configure the internal WIFI module of Arduino using our Mobile Hotspot (Dong *et al.*, 2013). The following are the steps for configuration of Arduino Yun Wi-Fi module:

1. Connect a new Arduino Yun (or Reset Wi-Fi of an older one) to a power supply such as USB or a Battery. After a couple of minutes, a network named as Arduino-XXXXXX will appear on your networks list as shown in Figure 11. Connect to that network.
2. After connecting to that network, go on to the browser and type 192.168.240.1 which is the IP address of the Arduino Yun. After hitting the enter button, a window similar

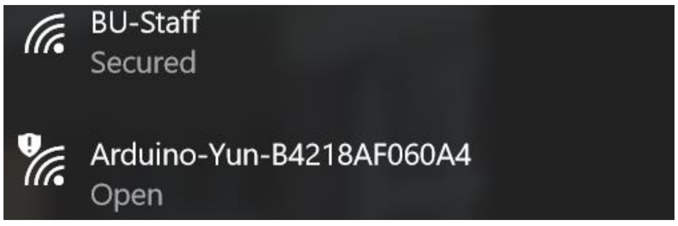


Figure 11 Arduino Yun Network

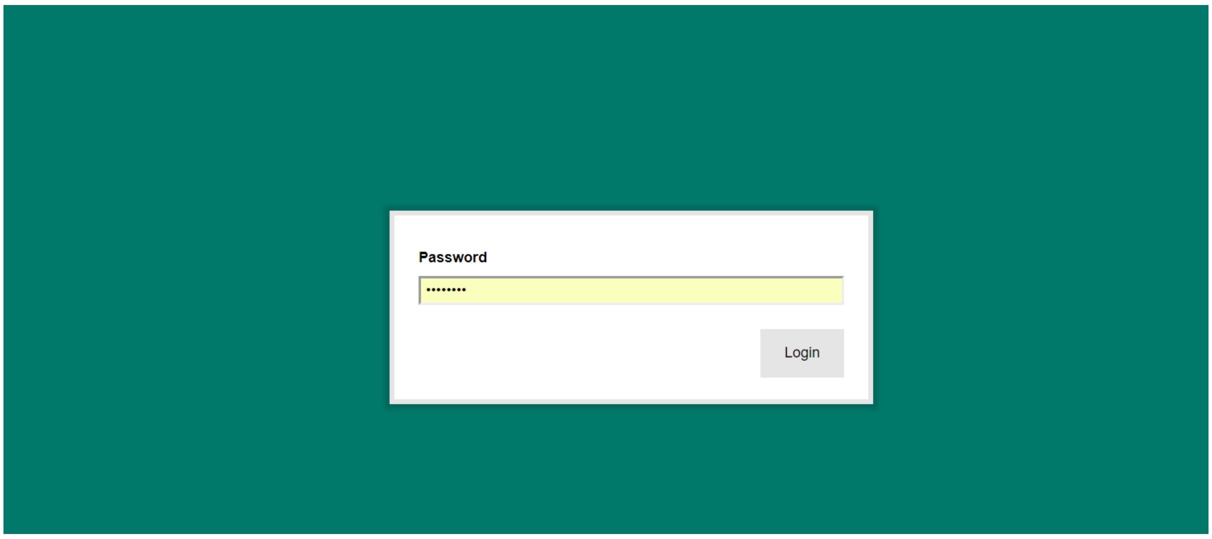


Figure 12 Arduino Yun Login

to Figure 12 will appear. If you are using Arduino Yun for the first time, the default password is *Arduino*. Enter the password and hit enter.

3. After your password is deemed correct by the system, you will be taken into the Arduino Configuration Wizard similar to Figure 13.

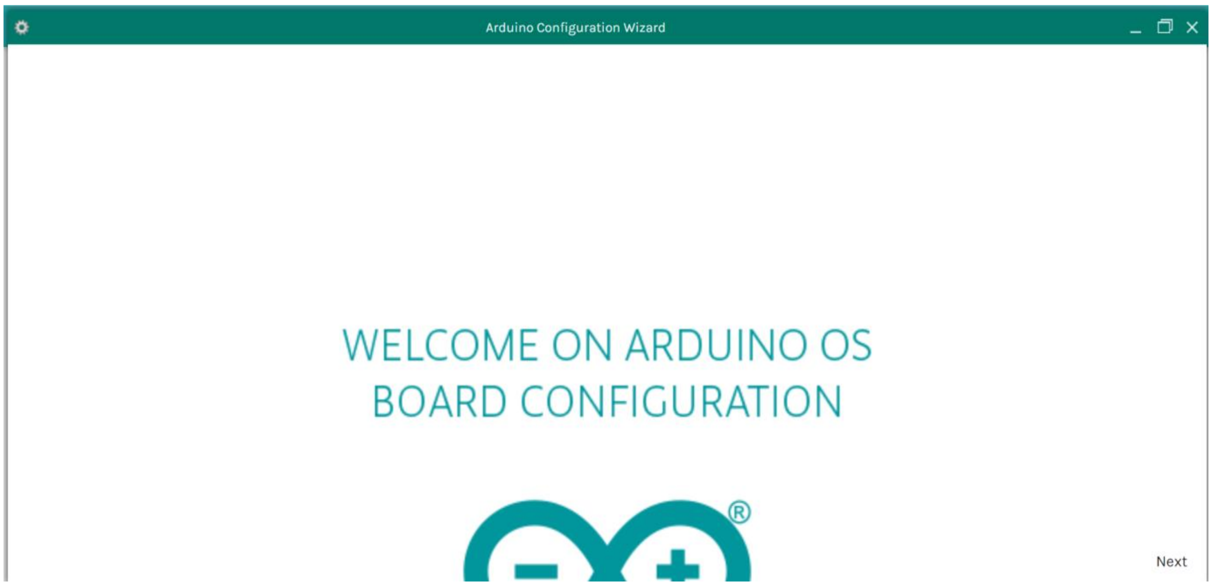
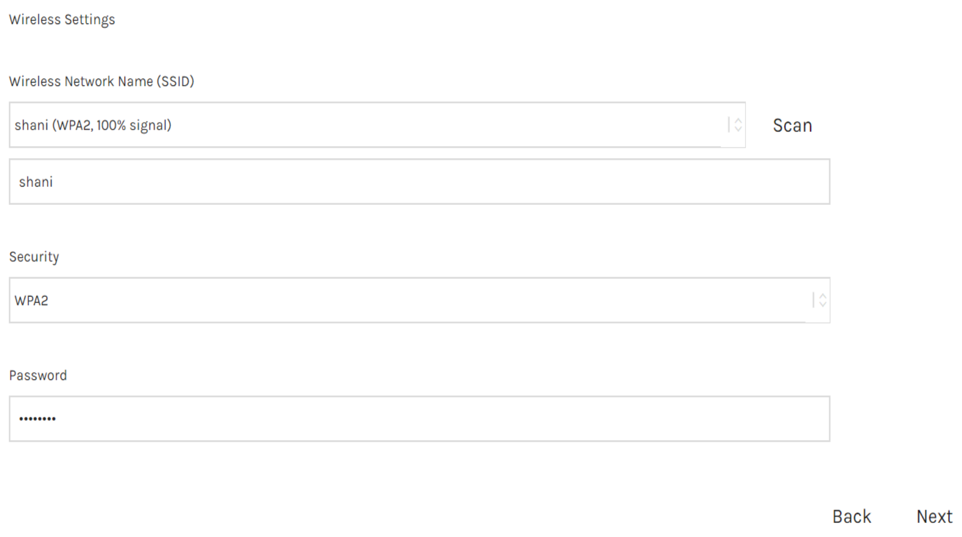


Figure 13 Arduino Installation

4. Press Next Button and you will see a screen depending on the model and version of Arduino Yun you are using.


Wireless Settings

Wireless Network Name (SSID)

shani (WPA2, 100% signal)

shani

Security

WPA2

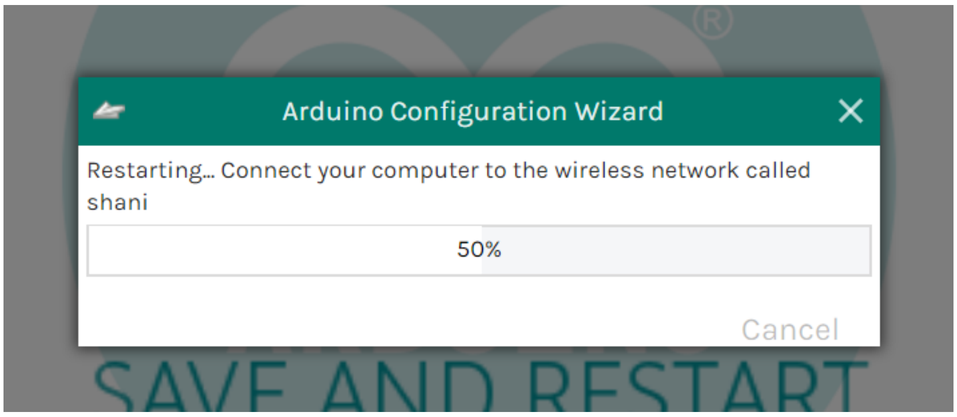
Password

••••••••

Figure 14 Mobile Hotspot Configuration

Here you can change the name of your Arduino Yun to anything you are comfortable with or you can keep the default name Arduino. You can also set the time zone you are in and change the password if you would like. This is very important as some public clouds will not show the data properly if your time zone is not set correctly as discussed in the Arduino Forums.

5. After entering all the details, press Next button and it will take you to the Wireless network settings as depicted in Figure 14. At this step, you need to turn on the mobile hotspot. Select the Wi-Fi network of your mobile hotspot and enter its password and press next button. If your network is not on the list, you should scan and it would appear in the list.



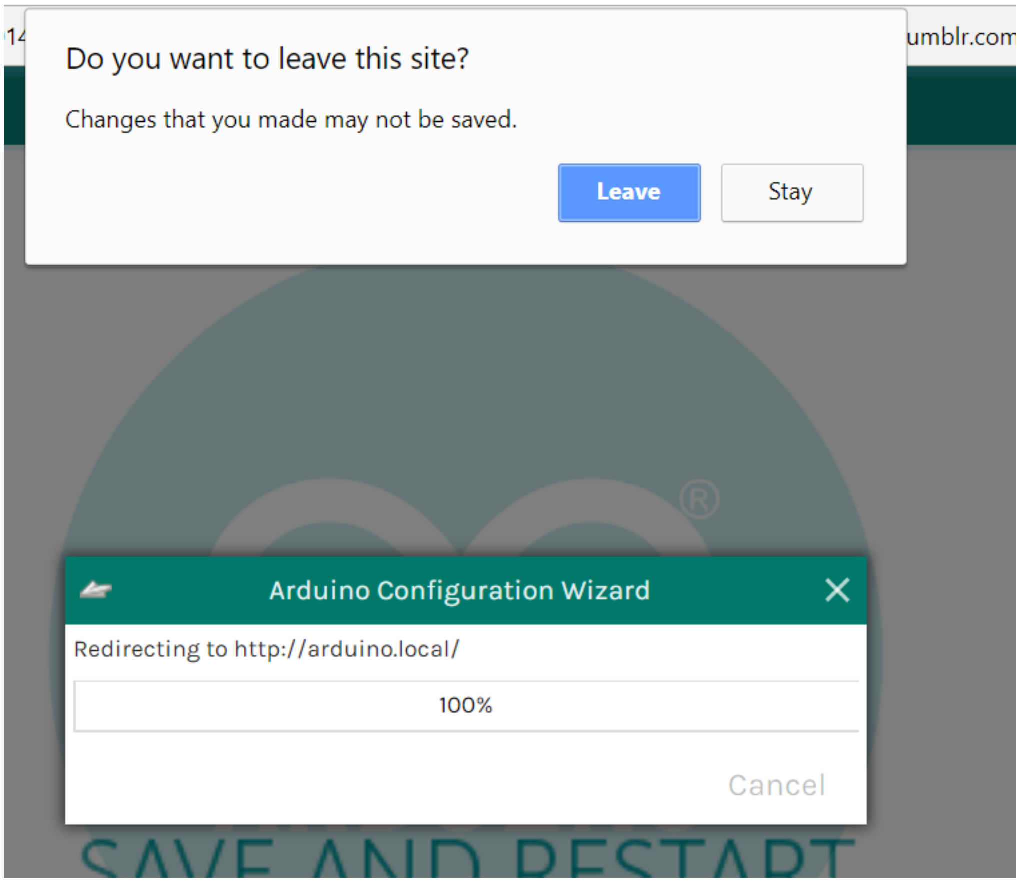
Arduino Configuration Wizard

Restarting... Connect your computer to the wireless network called shani

50%

Figure 15 Arduino Restart

6. After entering the details accurately, you will be asked to save and restart the Arduino Yun.
7. After you save and restart, the Arduino will start to apply the settings and restart itself. After a couple of minutes, you will be asked to connect your computer to the network of your mobile hotspot as shown in Figure 15.
8. After you connect your system to your mobile hotspot, the Arduino will redirect to the address with *Arduino.local* shown in Figure 16. This address can change depending on your name selection of your Arduino Yun.
9. Now, your Arduino Yun is connected to internet and you can upload any code sending or receive data from cloud servers and it should work fine.


Do you want to leave this site?

Changes that you made may not be saved.

Redirecting to http://arduino.local/

100%

Figure 16 Configuration Complete

Summary:

In this chapter, we described in detail our design for the prototype for urine measurement. Firstly, we presented a simple design for our prototype that takes the input from the liquid source, stores it in the container. The sensors attached to the dropper and container measure the data and with the help of a micro-controller, transmit that data to the public cloud through wifi. Moreover, we also presented our design of the whole system that includes reading the data through sensors, transmitting the data using Arduino libraries, communication channels involved and the data visualization techniques. Finally, we described in detail, the process of configuration of the micro-controller with Wifi and Mobile Hotspot. This is a necessary step to make our prototype completely wireless.

CHAPTER 4

PROTOTYPE DEVELOPMENT

In this chapter, we will describe in detail the development of the prototype for IoT based Urine Measurement System. The idea is to combine both high and low-flow sensors, to record and save the data from both the sensors and calculate the total output of urine. For achieving this task, we conducted a thorough research of the sensors that could serve our purpose. The options included Ultrasonic sensors, microphone based sensors Photo Diodes, Photo Interrupter sensor and module etc.

Firstly, we will discuss High-Flow and Low-Flow sensors separately and then we will combine both to develop the full prototype. The readings for both the sensors will be combined to calculate the total volume of the Liquid. As mentioned above, for high-flow, we used Hall-effect based sensor.

Comparison of our Design to Literature

The Urine Measurement System described in the literature that is the closest to our system was developed by *Atigorn Sanguansri and Team (*see Figure 17) at the Bournemouth University a couple of years ago. During their experiments, they showed that among the ultrasonic, photo diodes and photo interrupter sensors, the photo interrupter produced the best results. Atigorn used the photo interrupter and weight sensors to verify the measurements from the photo interrupter. As we can see from Figure 16 that liquid first passes through the dropper and then there is another dripping chamber placed just above the urine bag. Firstly, the drops have to pass through more tubes and part of liquid might be lost by attaching to the tube due to the adhesive force of the liquid molecules and some data might be lost. Secondly, when the urine flow is fast, the drop frequency increases and photo interrupter is not able to detect the drops everytime it passes and the measurements is considered to vary. Therefore to solve these challenges, we used two sensors instead of one. One measures the drops when the flow is slow and if the flow increases, the liquid passes through the high-flow sensor and it is measured separately.

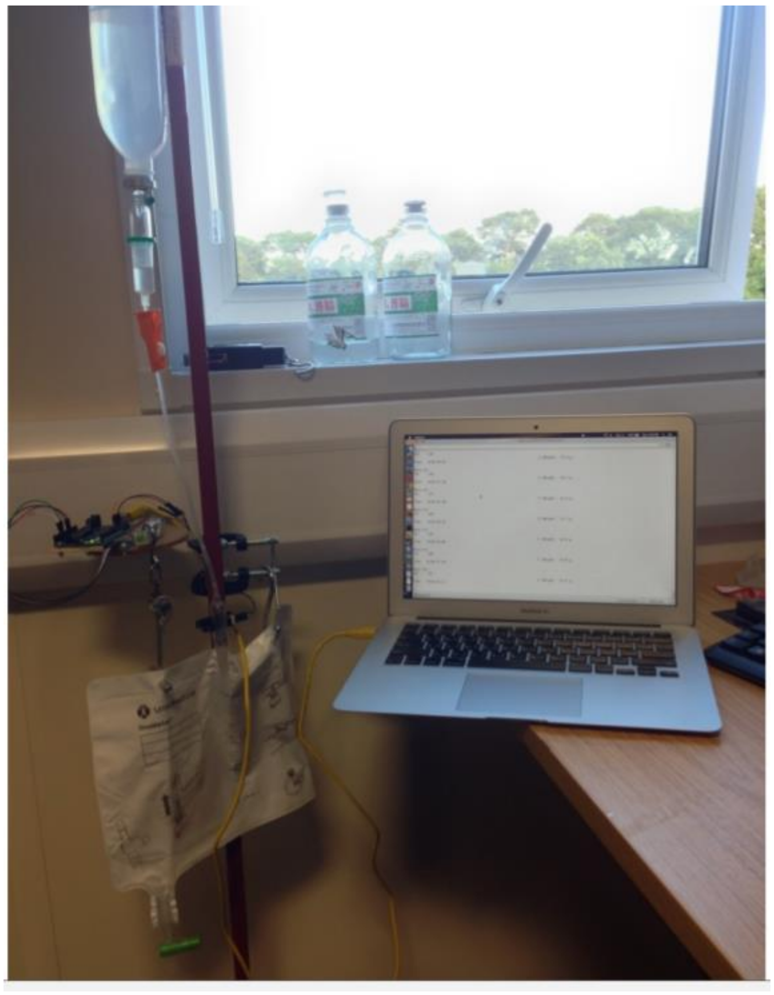


Figure 17 Urine Measurement System by Atigorn

High-Flow Liquid Monitoring

Hall-effect principle refers to the induction of voltage difference in a conductor across the electric current and a magnetic field is thus produced perpendicular to it. In this sensor (delineated by figure 18) hall-effect is used by placing a rotor in the way of the flowing liquid. When the liquid is passed through that rotor, it rotates and produces a magnetic field and pulses are generated.

The number of pulses generated per second for one liter of liquid flown through the sensor varies through different companies and measuring compacity of the sensor. Some sensors can measure as low as 0.1L per minute and some have the lowest capacity of several liters. It is worth noting here that our prototype will have wider implications and applications across the process industry as well as a medical industry because it would be able to measure the high flow of liquid. The sensor we used could measure from 0.1L per minute to 10L per minute.

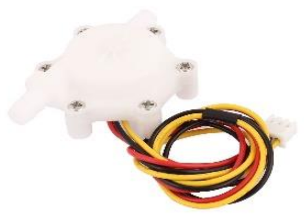


Figure 18 Hall Effect Liquid Flow Sensor

Connection to Arduino

The liquid sensor has 3 pins which are usually of Red, Black, and Yellow colors. The red wire is connected with 5V on the Arduino, Yellow with the number 3 digital pin and to complete the circuit, the black wire is connected to GND (ground) pin. The circuitry is shown in Figure 19.

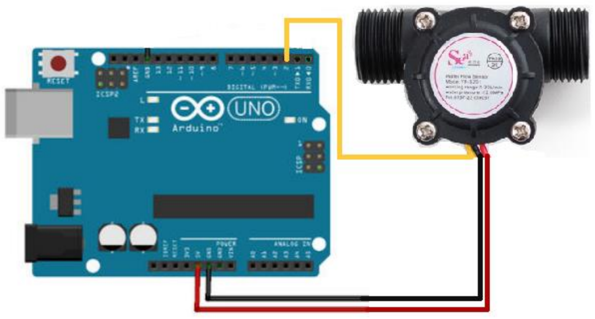


Figure 19 Circuit Connections of Flow Sensor with Arduino

The sensor we used generated 9.5 pulses per second for each liter of liquid passed through it in a minute. Here we should consider an important point that we noticed through experimentations. Sometimes, due to the difference in pressure of the liquid, the number of pulses written in the datasheet differs from what it produces in the real scenario. For example, for liquid from a pipe in a garden, the sensor generates 9.5 pulses per second whereas the same sensor when attached to a bottle, generates less than 9.5 pulses. So, for ensuring the readings, we must calculate the pulses for our own case study and experiments.

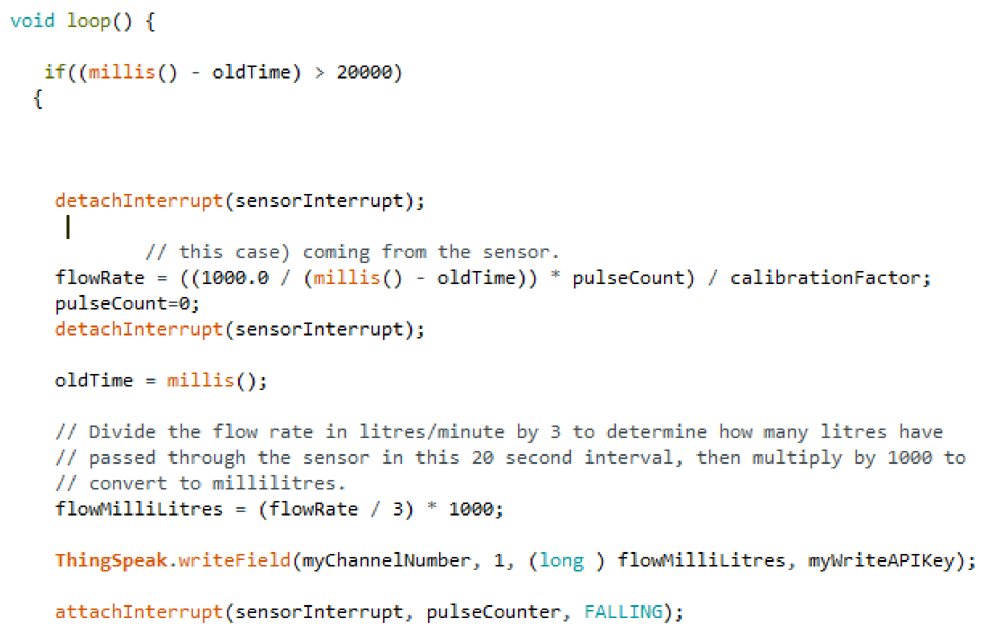
The cross-sectional area of the pipe is constant and already known so to determine the pulse frequency, the following formula is used:

$$f = 9.5Q$$

Where 9.5 is the pulses generated per second and Q is the flow of liquid in liters. So, we can get the approximate total number of pulses generated by keeping the time log and measuring the liquid on our end. Although this seems to be a time consuming and hectic work but it is only done a couple of times for a sensor. After that, we can rely on our readings without any doubt. According to experts and our own experiments, for large amounts of liquid, this sensor gives the readings with up to 10% error which is very much acceptable.

The complete code is not provided in this section (provided in Appendix) but one can get the main idea how we can get the main idea of how the liquid volume is calculated. This, however, gives only the liquid volume on serial monitor in the Arduino Integrated Development Environment (IDE). What we intend to do in this project is to visualize the output in such a way that is both convenient and interactive to the users. Moreover, for this program to execute, the serial port must be used and the idea of wireless connectivity and interaction, which is the main feature of Arduino Yun, remains unutilized. So, for this purpose, we wrote another algorithm that used the WIFI module of Arduino Yun to send the data to a cloud server.

Now, we will describe the algorithm that we used to measure the high-flow of the liquid (see Figure 20). First of all, we set the time of the fluid measurement to every 20 seconds as the data is transmitted every 20 seconds to the cloud. To calculate the flow-rate we use the pulseCount and calibration factor. pulseCount is the number of pulses that are generated for the flowing liquid and the calibration factor is 9.5 as described above. Once we get the flow-rate, we can easily calculate the total milli-liters of liquid that has been flown through the sensor in the last 20 seconds. After that, every 20 seconds, number of milli-liters of liquid is sent to the cloud using a specific channel number and its API Key. It is worth noting here that flow-rate calculated is per minute, to which we then converted to per 20 seconds by dividing it by 3.



```
void loop() {

   if((millis() - oldTime) > 20000)
  {


    detachInterrupt(sensorInterrupt);

            // this case) coming from the sensor.
    flowRate = ((1000.0 / (millis() - oldTime)) * pulseCount) / calibrationFactor;
    pulseCount=0;
    detachInterrupt(sensorInterrupt);

    oldTime = millis();

    // Divide the flow rate in litres/minute by 3 to determine how many litres have
    // passed through the sensor in this 20 second interval, then multiply by 1000 to
    // convert to millilitres.
    flowMilliLitres = (flowRate / 3) * 1000;

    ThingSpeak.writeField(myChannelNumber, 1, (long ) flowMilliLitres, myWriteAPIKey);

    attachInterrupt(sensorInterrupt, pulseCounter, FALLING);
```

Figure 20 High-Flow Algorithm

Low-Flow Measurement

Now, we will discuss the low-flow measurement of the liquid using the Photo Interrupter Sensor, shown in Figure 21. The main idea behind this experiment is that the sensor will be attached to a dropper and with each drop falling, the sensor will generate the voltage between 0-5V (0-1023 Analog reading). By measuring the fluctuations in the readings, we can count the number of drops passing through the sensor. According to the estimations and experiments were done by (*Atigorn Sanguansri* 2016), one drop is approximately equal to 0.05ml.

To calculate this parameter, we take 100ml of water. Then, by counting the drops from the photo interrupter sensor we have the total drop count for the 100ml of liquid. Finally, by using the following equation, we can calculate the volume of one drop, V

= L/N, where L is the total liquid and N is the total number of drops. In our case, the total drop count was 1892. If we apply our data to the above equation, the volume of one drop is 0.05ml.

So, by multiplying the counted number of drops by the volume of one drop, we can easily count the total volume of the liquid passing through the sensor in unit time.

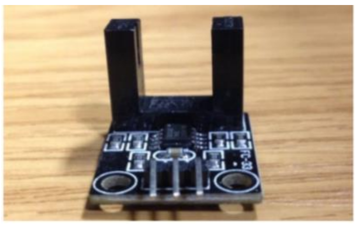
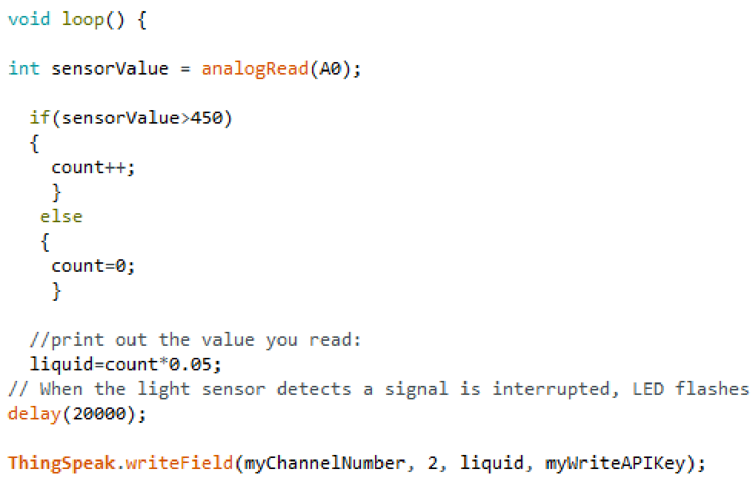


Figure 21 Photo Interrupter Sensor

For low-flow measurement, we developed an algorithm (shown in figure 22) in which we kept a record of the readings of the analog sensor. After series of experiments, we found out that when the drop passes through the sensor, the analog reading is greater than 450. So, if the reading is greater than 450, the drop counter is increased by 1, otherwise, it remained the same. And by multiplying the number of drops by 0.05, we calculated the total liquid passed.



```
void loop() {

int sensorValue = analogRead(A0);

  if(sensorValue>450)
  {
    count++;
    }
   else
   {
    count=0;
    }

  //print out the value you read:
  liquid=count*0.05;
// When the light sensor detects a signal is interrupted, LED flashes
delay(20000);

ThingSpeak.writeField(myChannelNumber, 2, liquid, myWriteAPIKey);
```

Figure 22 Low-flow Algorithm

The data from the low-flow sensor goes to the ThingSpeak cloud where it is analyzed in the form of graphs. A sample data from the cloud is shown in Figure 24 It is worth noting here that the data sent to the cloud is calculated at the Arduino end with the help of an algorithm written by the authors to facilitate the interactive visualizations at the cloud end. The prototype at our lab is shown in Figure 23.

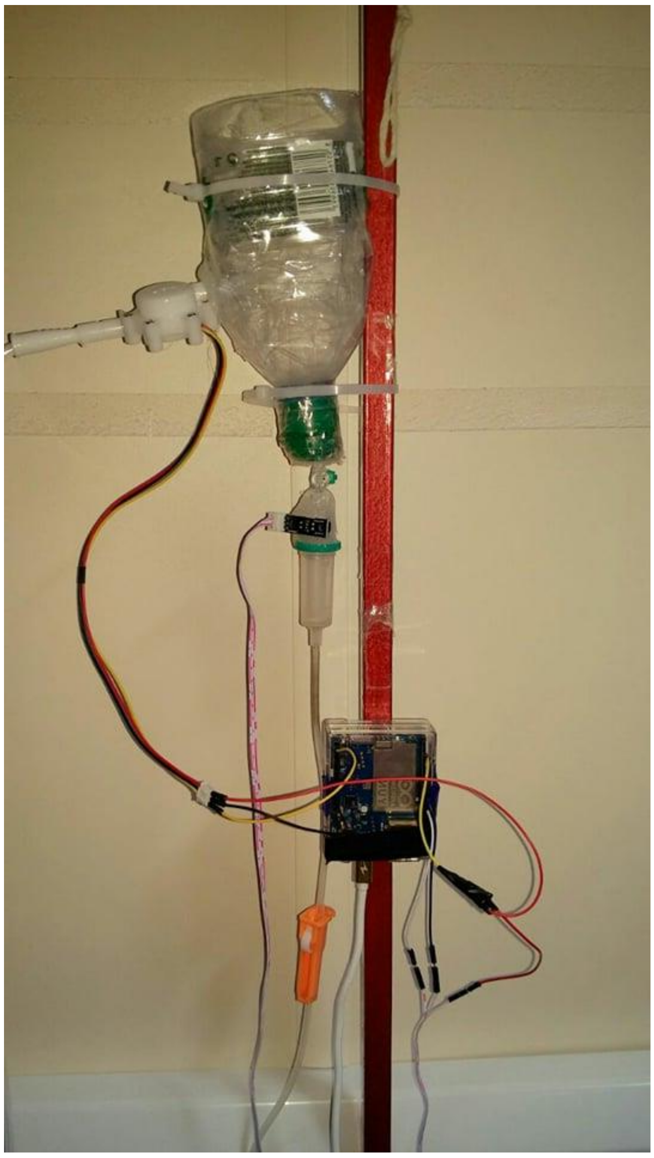


Figure 23 Prototype Development

Comparison of Data Analysis Mechanism with the Literature

Here, we compare our Data Analysis design with the one developed by Atigorn at the Bournemouth University a couple of years ago. Atigorn used TEMBOO library to send the data from Arduino to Google Cloud. Firstly, the library just provides a trial version and it needs to be licensed to use it on a large scale. Secondly, there is a tedious process of authentication first at TEMBOO and then at Google Cloud which might slow down the data transmission process. Furthermore, the data was sent every 10 minutes to the cloud which cannot be classified as real-time. Finally, the process of including credentials such as OAuth, and other IDs from TEMBOO on the Arduino programs makes it complex to manage and update the code.

So, we came up with a much simpler solution for this. We used ThingSpeak cloud service that is based on MATLAB for data analysis specifically designed for IoT Data Analytics. The data from Arduino is sent every 20 seconds because this is the minimum time interval after which the data can be sent from Arduino to the Cloud, so, the chance of data loss or corruption is low. This time interval is the closest we can get to the real time. Instead of complex credentials, we just have to add Channel ID and API Key to our program and the data is sent smoothly. Furthermore, the data is easily accessible via a smart phone application that has the option to optimize and select the data range that we want to see. The details are shown in the next chapter.

It is worth noting here that our system runs much more smoothly and it is easy to update the code and change the series of data according to our requirements. Although it is a public cloud service but it was sufficient enough and efficient enough to depict our proof of concept and novelty which is the main objective here.

Introducing Thing Speak Cloud

Thing Speak is a cloud service that uses MATLAB for data visualizations. It facilitates the IoT based applications to access data remotely from anywhere at any time. The most fascinating feature of Thing Speak is its integration with MATLAB which is considered by Data Scientists as one of the most powerful tools for data analysis and data visualizations. We used Thing Speak for our project with Arduino which sent data every 20 seconds to Thing Speak private channel. We then wrote the algorithm for creating visualizations with MATLAB on the cloud (shown in Figure 25).

Data Collection (Data Layer)

This process is a part of Data Layer of our system model in which the data is collected and then filtered based on the parameters set out in the programs described by Figures 20, 22 and 39. The data is filtered when the drops are not passing but the low-flow sensor still sends the readings. The solution of which is given in Figure 39. A channel at Thing Speak cloud is a container which stores the data. We can create as many channels we want, and we can keep data of similar type or project in one channel. A channel can be Private and Public. A private channel has API keys for reading and writing to a channel along with a unique ID. A public channel can be accessed by anyone who has the channel ID or the public link shared by the channel creator or moderator. To create a channel, we can follow the following steps:

- Assuming the account has already been created at thingspeak.com, go to the ‘Channels’ tab and press the New Button. (see Figure 24)

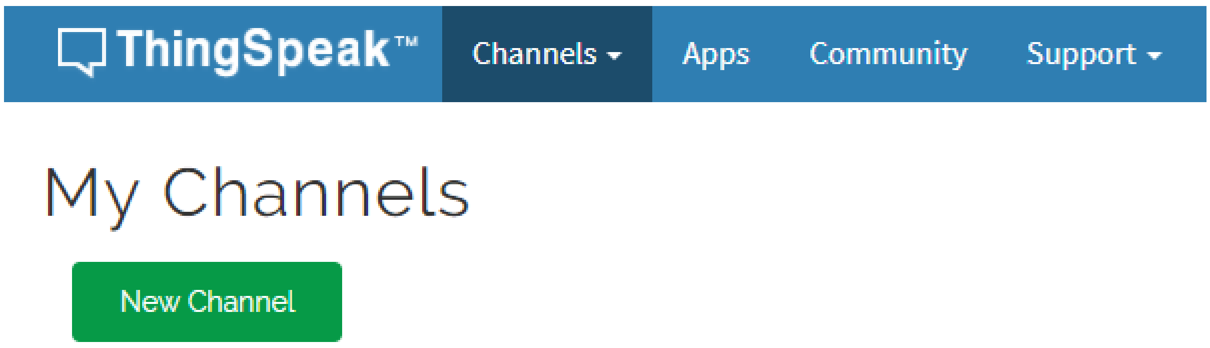
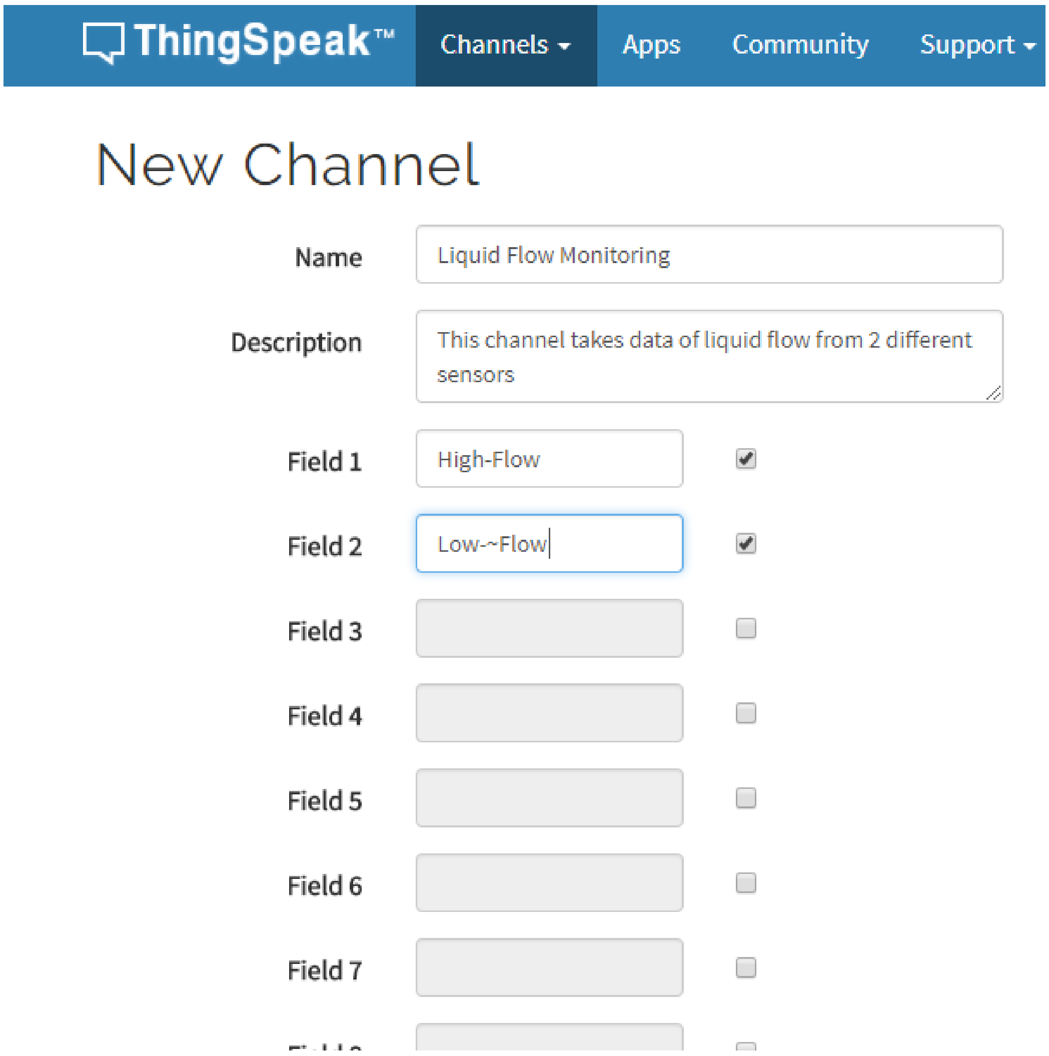


Figure 24 Create a New Channel

- After that, you will have a screen similar to Figure 25. Enter the Channel's name and the fields you want to add. A field is a small container that stores a specific data from one sensor and visualizes it. You can choose a total of 8 fields within one channel.



New Channel

Name	Liquid Flow Monitoring	
Description	This channel takes data of liquid flow from 2 different sensors	
Field 1	High-Flow	✓
Field 2	Low-~Flow	✓
Field 3		☐
Field 4		☐
Field 5		☐
Field 6		☐
Field 7		☐

Figure 25 Channel Details

- We can add other information to your channel also like a website link, a YouTube or vimeo video that may depict the experimentation of our project.

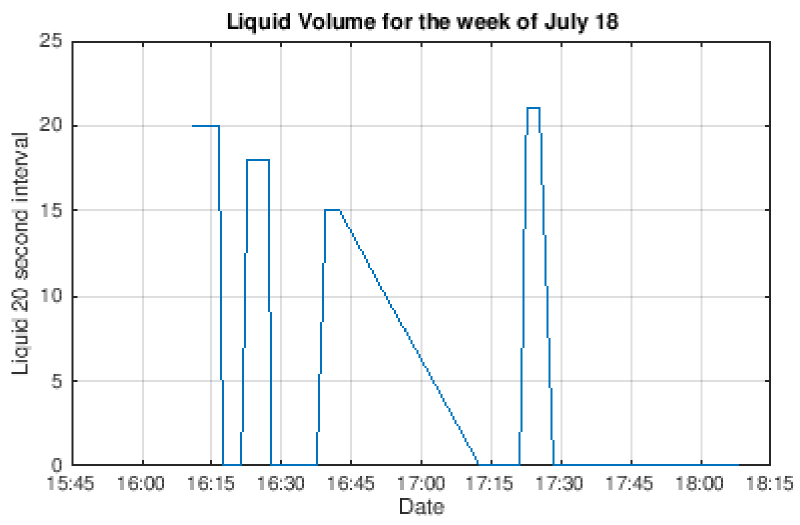


Figure 26 Liquid Data for July 18

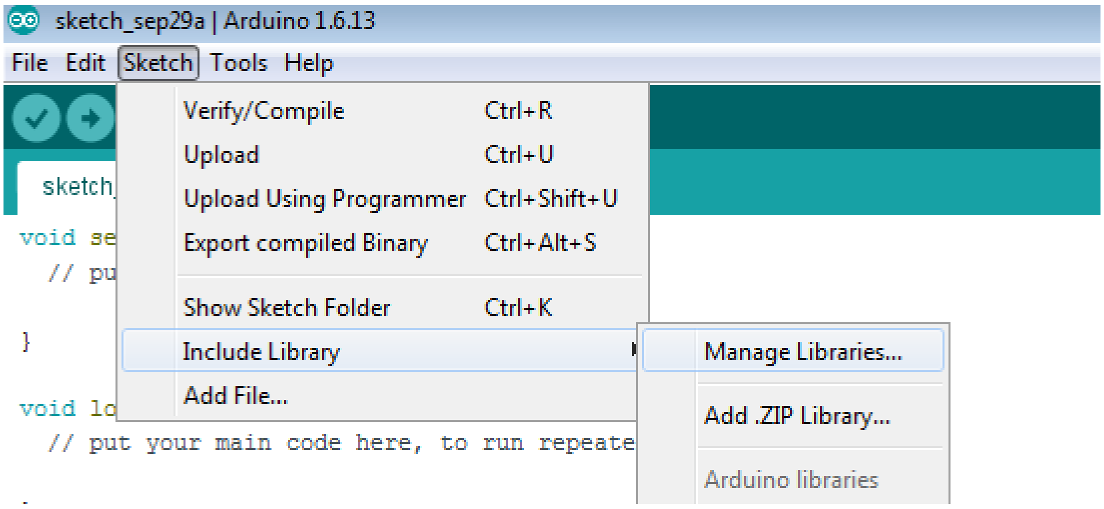
The graph (see Figure 26) shows the data points for July 18, 2017 data from our liquid sensor. The x-axis shows the timestamp and the Y-axis shows the volume of liquid that passed through the sensor. The volume of the liquid is in milliliters. Each data point represents the liquid passed through the sensors in one interval which is 20 seconds. If we add up all the data points, we can get the daily total volume of liquid. In this way, we can create as much visualization with different conditions using powerful MATLAB tool.

Data Transmission from Arduino to Thing Speak

Data is transmitted from Arduino to Thing Speak using a Thing Speak library that is specifically built for Arduino. This process of data transmission is a part of Communication Layer where Arduino Library is used and the data is transmitted using

Mobile Hotspot through WiFi. We can install this library through Arduino IDE with the help of following steps:

- Go to the sketch tab, and then to Include Library and then select Manage Libraries (see Figure 27)

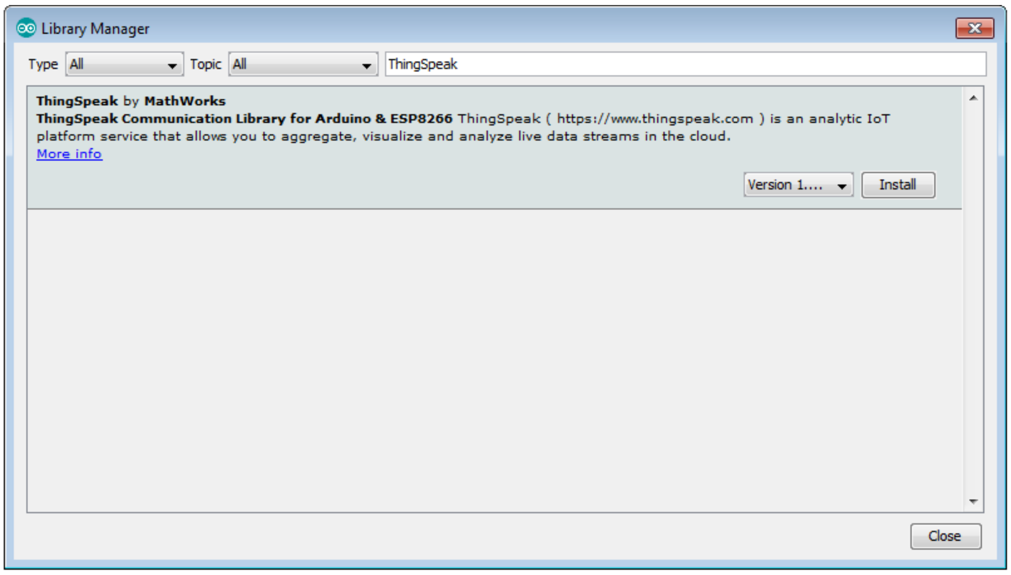

sketch_sep29a | Arduino 1.6.13
File Edit Sketch Tools Help
Verify/Compile Ctrl+R
Upload Ctrl+U
Upload Using Programmer Ctrl+Shift+U
Export compiled Binary Ctrl+Alt+S
Show Sketch Folder Ctrl+K
Include Library
Add File...
Manage Libraries...
Add .ZIP Library...
Arduino libraries
sketch

```
void se
  // pu

}

void lo
  // put your main code here, to run repeate
```

Figure 27 Manage Libraries



ThingSpeak

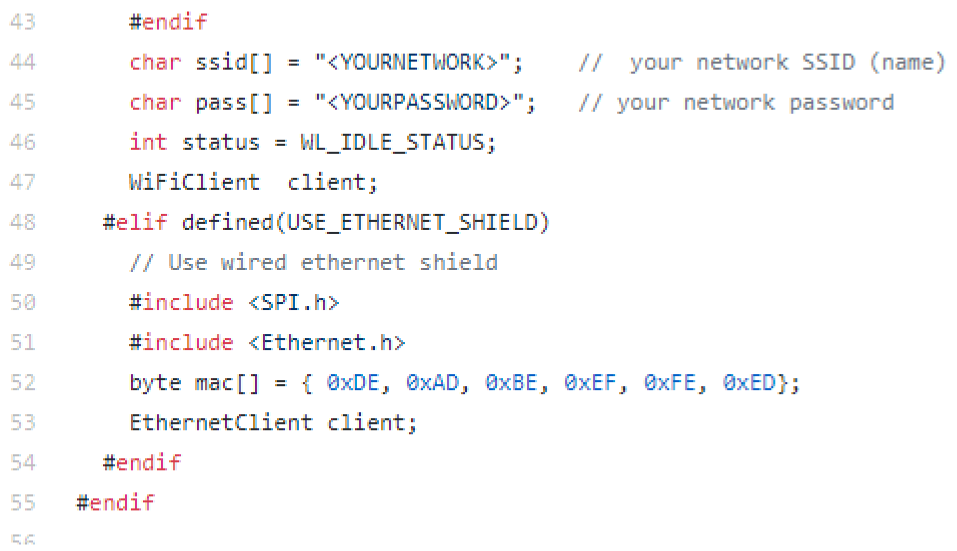
ThingSpeak by **MathWorks**
ThingSpeak Communication Library for Arduino & ESP8266 ThingSpeak (https://www.thingspeak.com) is an analytic IoT platform service that allows you to aggregate, visualize and analyze live data streams in the cloud.
More info

Figure 28 Adding Thing Speak Library

- Then search ‘Thing Speak’ from the filter search space and you will see the latest version of the Thing Speak library from Math works (see Figure 28) on your screen. Install the library from here and the code for Data Transmission to Thing Speak cloud should work smoothly.

Now that the Thing Speak library is installed in the Arduino IDE, we have to write the algorithm to send the data to Thing Speak. The complete code is provided in the appendix section, but we will glance through a few important and necessary parts of the algorithm with an example.

For example, we want to send any type of Data from Arduino Yun to Thing Speak Cloud; firstly, Yun should have been assigned an IP address by Mobile Hotspot. As we can see from Figure 29, in the Arduino code, we have to enter the SSID and Password of the network that Arduino is connected to.

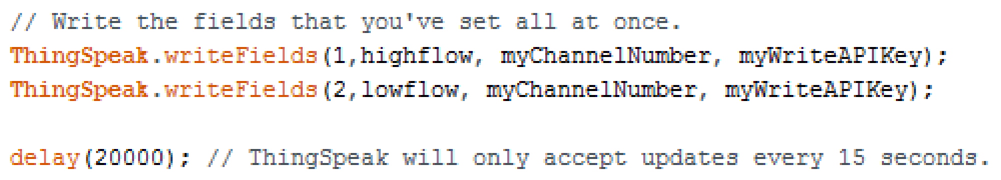


```
      #endif
      char ssid[] = "<YOURNETWORK>";    //  your network SSID (name)
      char pass[] = "<YOURPASSWORD>";   // your network password
      int status = WL_IDLE_STATUS;
      WiFiClient  client;
    #elif defined(USE_ETHERNET_SHIELD)
      // Use wired ethernet shield
      #include <SPI.h>
      #include <Ethernet.h>
      byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
      EthernetClient client;
    #endif
  #endif
```

Figure 29 Sending Data to Thing Speak

Then from Figure 29, we can see that the Channel ID and Write API Key of the channel are required to write the data to that channel. The same process can be repeated to Read the data by replacing Write API Key with the Read API Key of the channel you want to read the data from.

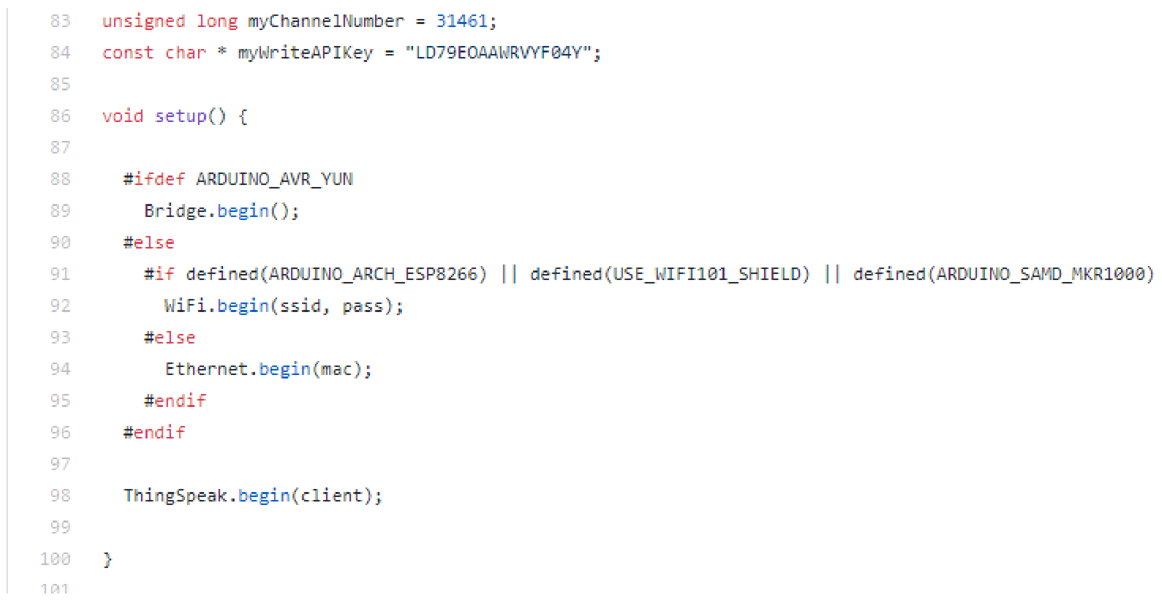
Furthermore, you can send multiple types of data from one program of Arduino just by sending the data to different fields of the same channel. Before this, you will need to create different fields when you are creating the channel which is shown in the previous section. From Figure 30, you can see how can the data be sent from multiple sensors to a single channel.



```
// Write the fields that you've set all at once.
ThingSpeak.writeFields(1,highflow, myChannelNumber, myWriteAPIKey);
ThingSpeak.writeFields(2,lowflow, myChannelNumber, myWriteAPIKey);

delay(20000); // ThingSpeak will only accept updates every 15 seconds.
```

Figure 30 Sending Data to Thing Speak



```
unsigned long myChannelNumber = 31461;
const char * myWriteAPIKey = "LD79EOAAWRVYF04Y";

void setup() {

  #ifdef ARDUINO_AVR_YUN
    Bridge.begin();
  #else
    #if defined(ARDUINO_ARCH_ESP8266) || defined(USE_WIFI101_SHIELD) || defined(ARDUINO_SAMD_MKR1000)
      WiFi.begin(ssid, pass);
    #else
      Ethernet.begin(mac);
    #endif
  #endif

  ThingSpeak.begin(client);

}
```

Figure 31 Sending Data to ThingSpeak

MATLAB Visualizations

One of the most powerful features of Thing Speak Cloud is the visualization facility provided by MATLAB in co-operation with Thing Speak team. There are many sample codes and templates provided by Thing Speak for us to play with and we can extend those templates according to our requirements with few tinkering with the MATLAB code. The range of graphics that we can add is the most fascinating thing about MATLAB visualizations. Some of the examples are given below:

Creating a Plot with Labels of Years:

We can create the 2-D line plot with a few lines of MATLAB code and we can also label the axes with timestamps like shown in figure 32.

```
t = (1900:10:1990)';
p = [75.995  91.972 105.711 123.203 131.669 ...
    150.697 179.323 203.212 226.505 249.633]';
thingSpeakPlot(datetime(t,1,1),p,'Grid','on')
```

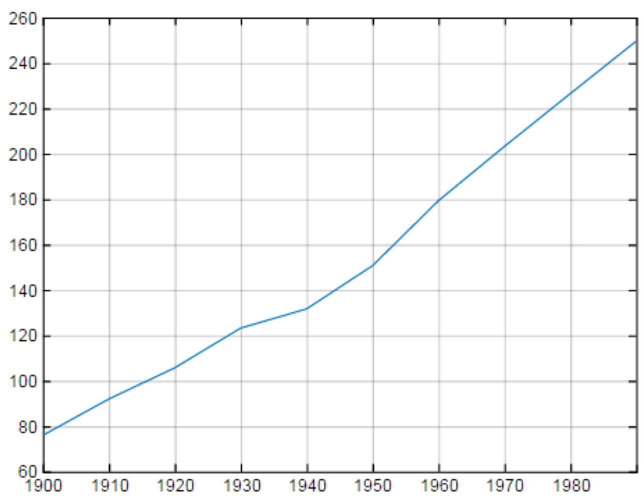
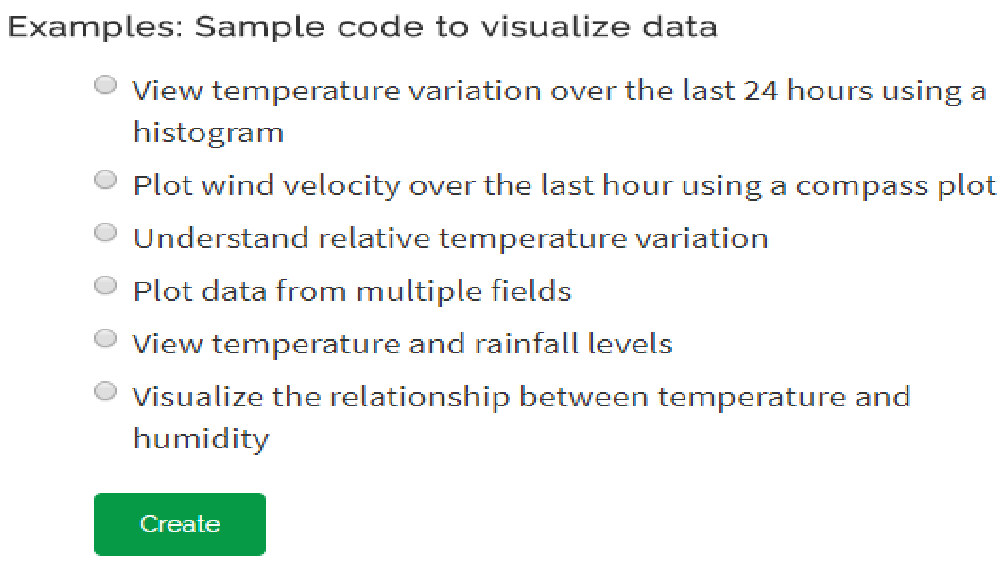


Figure 32 2-D Plot with Timestamp

Creating a New Visualization on Thing Speak Cloud:

To create a new Visualization using MATLAB on Thing Speak, we follow the following steps:

- Open the MATLAB Visualization App which depicts the examples and templates like Figure 33.

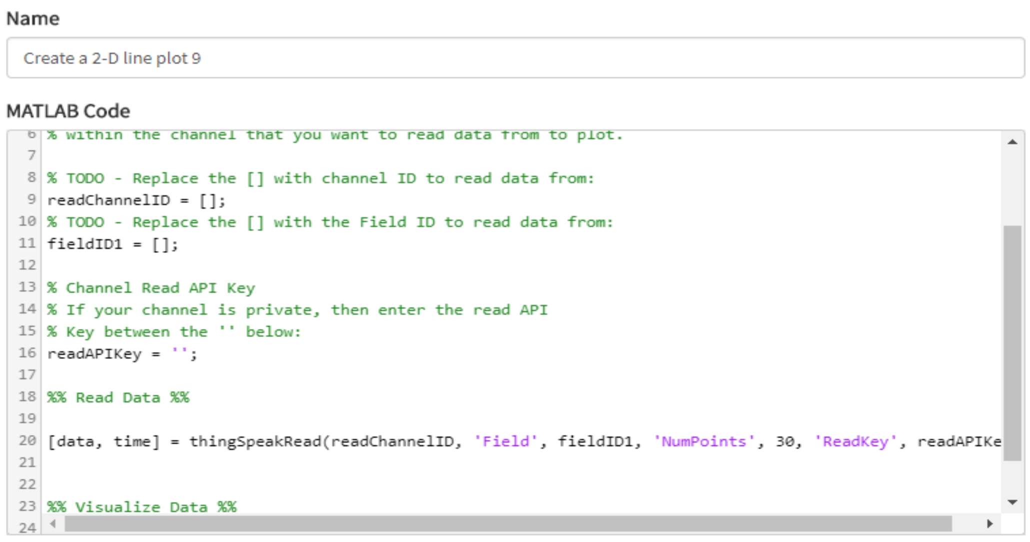


Examples: Sample code to visualize data

- ◯ View temperature variation over the last 24 hours using a histogram
- ◯ Plot wind velocity over the last hour using a compass plot
- ◯ Understand relative temperature variation
- ◯ Plot data from multiple fields
- ◯ View temperature and rainfall levels
- ◯ Visualize the relationship between temperature and humidity

Figure 33 Creating a New Visualization

- When we press the Create button, it takes us into the MATLAB code editor where we can write our own code or play with the default templates like shown in Figure 34.



Name

Create a 2-D line plot 9

MATLAB Code

```
% within the channel that you want to read data from to plot.

% TODO - Replace the [] with channel ID to read data from:
readChannelID = [];
% TODO - Replace the [] with the Field ID to read data from:
fieldID1 = [];

% Channel Read API Key
% If your channel is private, then enter the read API
% Key between the '' below:
readAPIKey = '';

%% Read Data %%

[data, time] = thingSpeakRead(readChannelID, 'Field', fieldID1, 'NumPoints', 30, 'ReadKey', readAPIKe


%% Visualize Data %%
```

Figure 34 MATLAB Code Editor

Summary:

In this chapter, we described in detail our development phase of the prototype. The algorithms developed for measuring both high and low-flow of the liquid has been separately explained. The Data transmission mechanisms like ThingSpeak library of Arduino and the public cloud for Data Visualization are thoroughly discussed. Furthermore, usage of MATLAB in analyzing the data at the public cloud is highlighted. Finally, the fully developed prototype at our lab is depicted and described in detail.

CHAPTER 5

RESULTS AND IMPROVEMENTS

In this chapter, we will show the complete results from combing the data from the two sensors depicted in the previous sections. For this thing, we explore the powerful feature of MATLAB visualizations on the Thing Speak cloud platform. We developed our own code in MATLAB which explores and stores each data point from the sensors into a new container.

Data Visualization

In this final phase, the part of Information Layer, we depict the data that is analyzed using the ThingSpeak Cloud application. The data from the high-flow liquid is shown in Figure 35. The curve shows the data points which depict the amount of liquid passed every 20 seconds as the data is updated every 20 seconds from Arduino to Thing Speak Cloud. As we can see, when the data is not passing through the sensor, the curve goes to 0 and it suddenly goes up the moment liquid started flowing. This ensures that the data is recorded in real time.

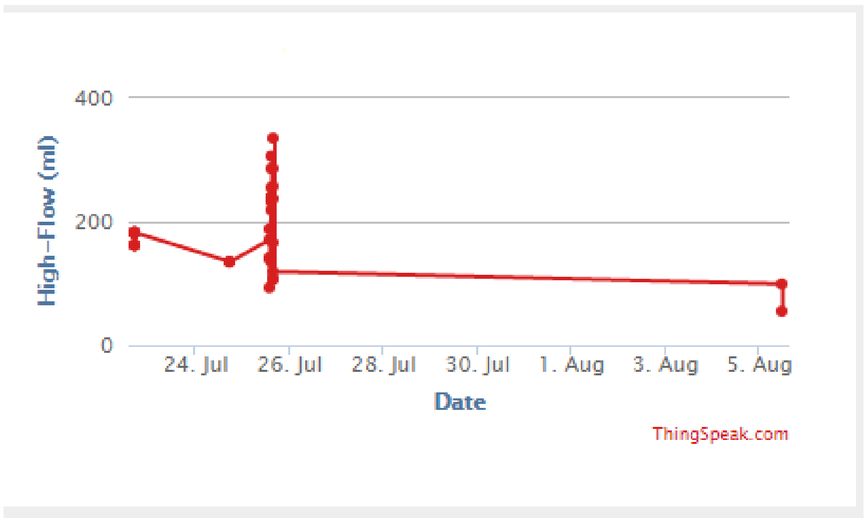


Figure 35 High-Flow Data

Now, we look at the low-flow data depicted in Figure 36. It is worth noting that the data points in this figure represent the drops of the liquid. From one drop is 0.05ml, so one data point here represents 0.05ml of liquid which passes through the sensor.

This data is also accessible from the Android Smart Phone, depicted in Figure 37 and Figure 38. The data accessibility and availability on the go using different smart devices and the cost-effectiveness of the whole system makes it a unique fully integrated system for urine flow monitoring. As we can see in Figure 36, it is very easy to search the data for any custom period of time as well as hourly data and daily data.

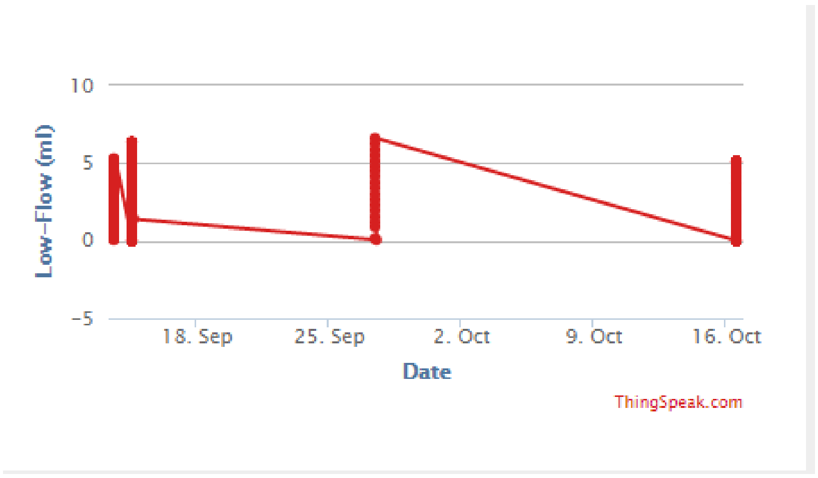
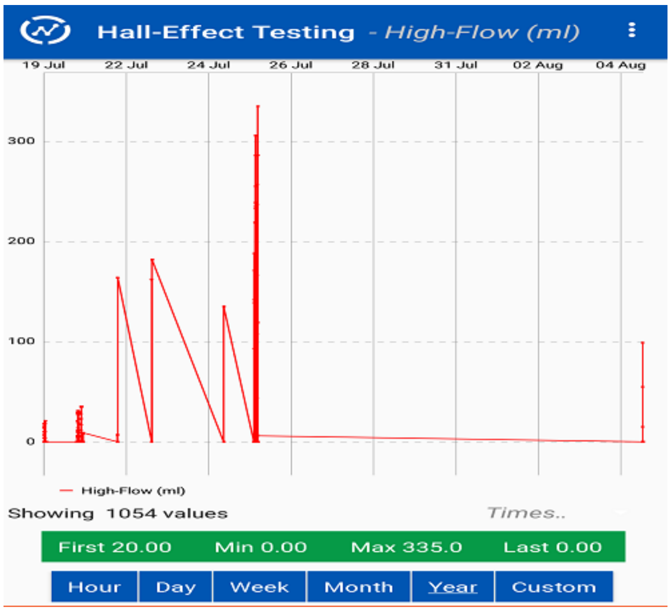


Figure 36 Low Flow Data



Showing 1054 values

First 20.00 Min 0.00 Max 335.0 Last 0.00

Figure 37 Android App High-Flow

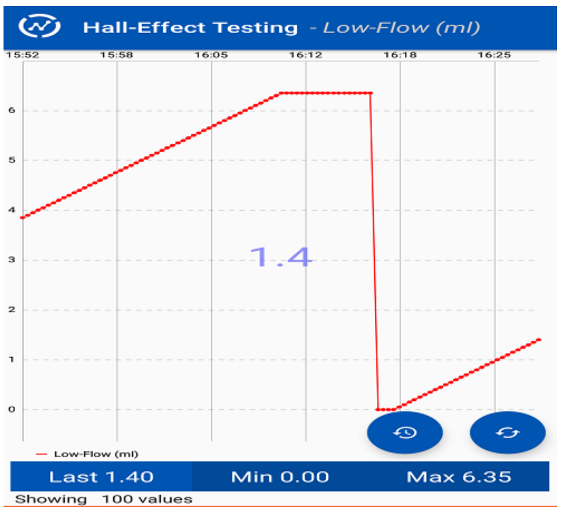
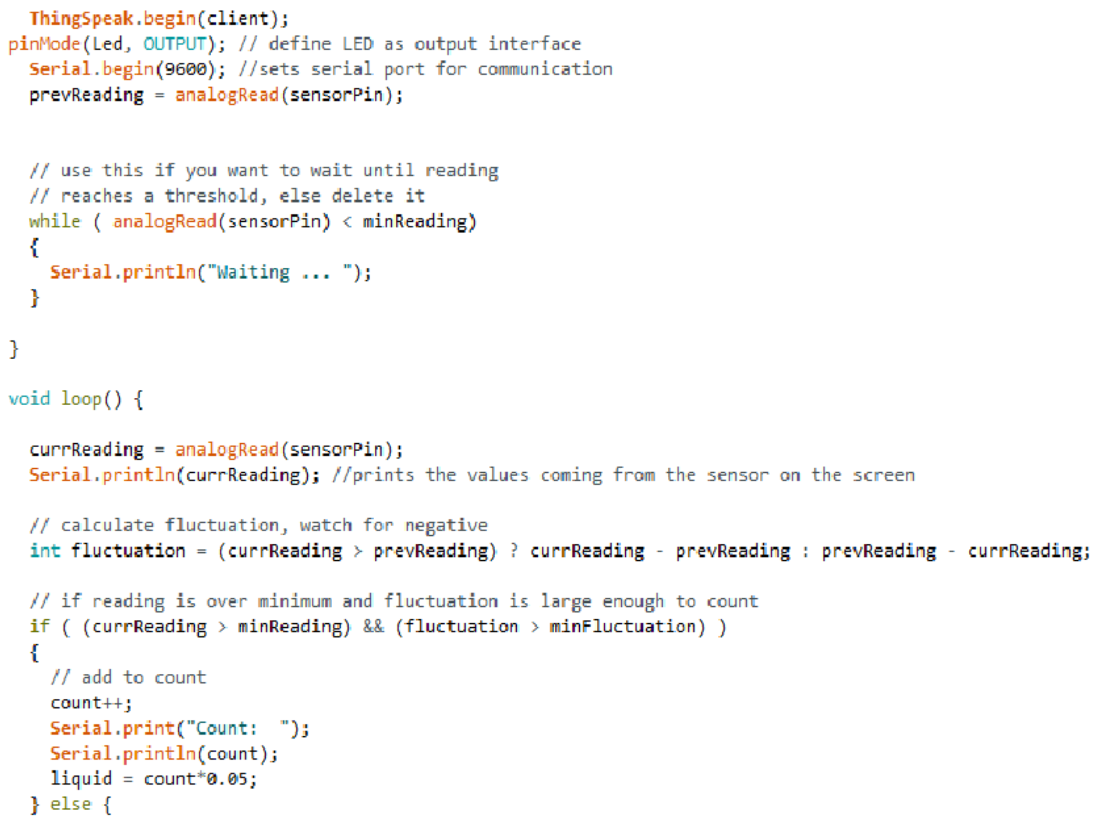


Figure 38 Android App Low-Flow

Results Evaluation:

After combining and visualizing the data, we noted down a problem in the hardware configuration. As due to the cost constraints, we had to use the off-the-shelf sensors, there were some drawbacks of the photo interrupter sensor. Due to the small size of the slot, we had to place it within the dropper and the liquid would pass through it. During our lengthy experiments, we noted that sometimes, the drop of the liquid touches the sensor and after that, even if the drops are not flowing, the sensor gave a positive reading means it counted the drops when there weren’t any. We tried many hardware and mechanical solutions, but none were convincing. So, we developed a software solution (see figure 39) for that. The algorithm maintains the record of the latest reading of analog sensor and the fluctuation in the reading. When the fluctuations increase or decrease by a certain amount, then the drop is counted, otherwise not.



```
  ThingSpeak.begin(client);
pinMode(Led, OUTPUT); // define LED as output interface
  Serial.begin(9600); //sets serial port for communication
  prevReading = analogRead(sensorPin);


  // use this if you want to wait until reading
  // reaches a threshold, else delete it
  while ( analogRead(sensorPin) < minReading)
  {
    Serial.println("Waiting ... ");
  }

}

void loop() {

  currReading = analogRead(sensorPin);
  Serial.println(currReading); //prints the values coming from the sensor on the screen

  // calculate fluctuation, watch for negative
  int fluctuation = (currReading > prevReading) ? currReading - prevReading : prevReading - currReading;

  // if reading is over minimum and fluctuation is large enough to count
  if ( (currReading > minReading) && (fluctuation > minFluctuation) )
  {
    // add to count
    count++;
    Serial.print("Count:  ");
    Serial.println(count);
    liquid = count*0.05;
  } else {
```

Figure 39 New Algorithm for low-flow measurement

During experimentation, we noted that the analog readings fluctuated with a minimum of 40 points when the drops passed through the sensor and when the drops weren’t passing, the fluctuations were very small. So, from this observation, we set the minimum fluctuation to 40 points from the current reading to make sure the drops passed. We maintain records of current and previous readings with the algorithm. So, as a result of this algorithm, we found out that now we could minimize the detection of the droplets in their absence by a considerable amount and this reduced our error to below 10%. The data after these improvements can be seen in Figure 40.

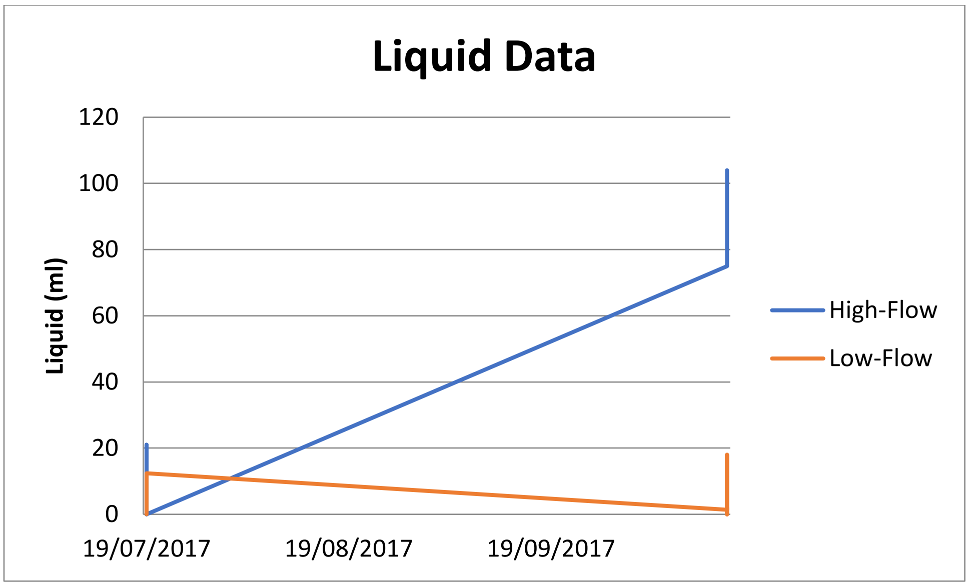


Figure 40 Data after Improvements

We then developed our own Plugin at Thing Speak Cloud, with the help of which, we could easily extract data from the two sensors using data points for any specific hour relative to the current time. For example, if we want to extract the data of last 5 hours we can simple write 5 in the hour's column and the data will be populated in the form of a graph depicting the milliliters of liquid at any specific time. One sample data is shown in Figure 41.

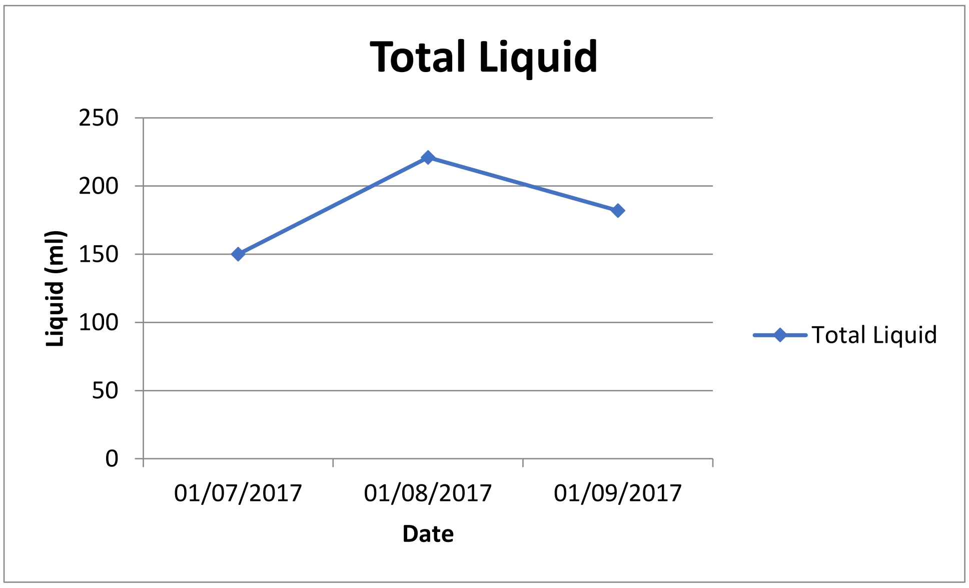


Figure 41 Combined Average Liquid Data

The above figure shows a line graph depicting the total liquid passed through the sensors on a day. With the above experiments and results, we have tried to prove our concept of using prototyping technique and IoT platform to develop and fully integrated urine measurement system.

Results Comparison with Existing System

In this section, we will compare the results of our system with the system developed by Atigorn at Bournemouth University a couple of years ago. The results shown by their system were also very impressive and the error difference was minimal, but we believe and our experiments and results prove that by developing the algorithm shown in Figure 38, we have improved the working of the low-flow sensor. The drop counting mechanism of the low-flow sensor is not very efficient when it comes to the continuous flow of liquid. But, with the help of our algorithm, we have been able to improve the measurements and hence the error was reduced. Therefore, the algorithm we developed is a significant contribution to the literature and it can be used in any scenario with the low-flow sensor we used in precise measurements.

Comparison of Costs, Accuracy, and Simplicity

In comparison with Atigorn's system, which is the closest to our system, the cost of our device is approximately £80 which is close to Atigorn's system. But, we used two sensors (for Low-flow and High-flow) and the reliability of our system is higher than the previous systems. In terms of accuracy, both systems have similar accuracy as presented in Atigorn's thesis (both approximately 90%). But, we figured out a problem in the Photo Interrupter Sensor (for drop counting) that the sensor readings that to be optimized for accurate readings (Solution is given on Page 67) and to the best of our knowledge, Atigorn did not discuss this problem (the same photo interrupter sensor was used). So, we believe, our system has more reliable accuracy. Finally, when simplicity is under consideration, we believe both systems have their advantages and disadvantages. In our system, the main concern was to test the concept and make sure the data is being measured accurately. We achieved this aim while using simple procedures and the data can be accessed in real time from the mobile application. This was done in a much more complicated way in the system in comparison where the data accessibility was not as simple and Google Scripts and Google Spreadsheets were used. We understand that to deploy our system completely in a hospital, it needs some manufacturing changes and a state of the art cyber security mechanism.

Discussion

The prototype and the software application developed on the cloud measured the amount of liquid passing through the sensors (low-flow and high-flow). The two sensors, connected to the Arduino micro-controller sent the data in real time to the public cloud. We optimized the sensors data by first mechanical and then by the software solution. There was a significant improvement made in the data of the low-flow sensor, solving the possible flaw of detecting the drops when there weren't any liquid passing through it. An algorithm was developed to solve that issue by recording the previous and current readings of the sensor and by continuous monitoring of the fluctuation of the sensor both when the drops are passing and when they are not. By recording the fluctuations for more than 12 hours, we came to a mean fluctuation point of 40 points in the analog sensor used for drop counting. If the fluctuation between two points is more than 40 points, only then the drop has passed and this algorithm helped us by reducing the error by more than 5% which is a significant achievement. This algorithm can be used along with this sensor in other applications such as object detection, light detection etc.

CHAPTER 6

CONCLUSION AND FUTURE IMPLICATIONS

There are many systems and devices that have been proposed and claimed over the past few years, but none have been able to meet the standards of the commercial market and industry requirements. Some were too complex to be implemented and some were not accurate enough.

In this chapter, we will review the research goals set out and the extent to which they have been achieved. Furthermore, we compare our system with the existing systems described in the literature review section. Finally, we highlight our contribution to the research and the future implications and applications of our proposed system.

Summary of the Invention:

The system proposed here conceptualizes the phenomenon of an integrated system of liquid measurement through felicitous technologies like IoT and Cloud Computing. We used very simple and basic concepts of IoT and software engineering to develop an efficient and automated system for urine or liquid measurement. Firstly, it is worth noting that none of the previous systems have been successful in designing a system that can measure the low as well as high-flow of liquid simultaneously. Furthermore, we also used modern concept of cloud computing to display and visualize the data that is accessible through all the smart devices like smart phones and tablets etc.

We used the hall-effect based sensor for high-flow measurement and a photo interrupter sensor for low-flow. The high-flow measurements and low-flow measurements were below 10% error which is acceptable according to the requirements provided by the doctors at Royal Bournemouth Hospital. This accuracy is rarely found in any literature to the best of our knowledge. It is also worth noting that we performed tests for up to 6 hours repeatedly for both the sensors.

We tested two scenarios for IoT based architecture for our prototype. We test the sensors individually by attaching them with different Arduino Micro Controllers and powered them with AA batteries and transmitted the data using Mobile Hotspot, making the prototype completely wireless. This was also not found in the literature making our design unique. Furthermore, after initial tests, we combine the algorithms for two different sensors and burned them on a single Arduino Yun. The results from both sensors were combined to give a total amount of liquid. The whole idea of making the device easy, simple and completely wireless distinguishes it from the current literature. Although this system still can be improved with improvements in sensor accuracy and compactness for commercialization, it does give a concept with proof that can be taken as a platform to proceed with further development.

Comparison of Complete System with the Current Literature:

The closest system from the literature to our prototype is developed by Atigorn. That system also used photo interrupter sensor but it only measured the low-flow of the liquid accurately and when the liquid flow is high, it was not equipped enough to measure it accurately. Whereas our system used two different sensors for high and low-flow of liquid, this makes our system much more reliable and accurate.

The system in comparison used Google Spreadsheets to analyze the data using a public library called TEMBOO. This library gives a trial version to use and needs to subscribe for permanent use and it also requires writing extensive code to analyze the data properly. On the other hand, we used Thing Speak which has better functionality and simpler process of data transmission through WIFI in real-time and it uses MATLAB which gives us the power to make advanced and extensive visualizations.

Comparison with Commercial Bluetooth Flowmeters

In this section, we will draw a comparison between our system and the commercially available device(s) for measuring liquid flow. Orcas [1], a Bluetooth enabled flow meter by Sound Water Ltd which measures the liquid flow through pipes and displays the output on a smart phone application via Bluetooth. The hardware device is attached to the pipe through which the liquid flows and the measurements are sent directly to the application via Bluetooth. This system is quite efficient in the applications where the flow rate is high as the minimum flow rate supported by this system 0.03m/s which is very high. Another system manufactured by Sierra[2] is also used to measure the high flow rate of liquids flowing through pipes. The accuracy is very high for this system, claimed to be up to 0.25% but the minimum flow rate is 0.49m/s. However, our system can measure the flow as low as 20ml/hr which is not possible with the flow meters manufactured by Sound Water Ltd and Sierra Ltd. Our application demands very low flow measurements, so, the commercial systems are deemed not suitable. We believe, to the best of our knowledge, there is no commercially available system that can measure the low-flow of liquid as achieved by our system.

[1] https://www.ayyeka.com/img/kits/SE00051_Semi-Permanent-Ultrasonic-Flow-Meter_SoundWater-Technologies_Orca_Datasheet.pdf

[2] http://www.sierrainstruments.com/products/210-portable-tablet.html

Research Goals and Outcomes:

Our first two Research Questions were to understand the requirements of the user, in this case, the hospital or the industry where the fluids need to be monitored and the current practices in the real world. Through thorough literature review and our visit to the Royal Bournemouth Hospital, we investigated the current practices and technologies and also the shortcomings of the current systems deployed or claimed to measure the liquid flow.

Furthermore, our second set of research questions included the design and testing of a simplified prototype to improve human-computer interaction and to ease the workload of the nursing staff. By using off-the-shelf sensors and combining felicitous technologies like IoT and Cloud Computing, we made sure that the data collection and visualization is simple and efficient.

Hence, as described above, we achieved our research goals and developed an integrated system that is both efficient and accurate. But, we faced challenges and our system also have some shortcomings which will be highlighted the below:

Although this system has many advantages over the current systems described in the literature, it has some short comings which we must highlight to give the directions for the future development and improvements. These are as follows:

- The high-flow sensor is not accurate enough and development of a new sensor specifically to measure the liquid flow must be carried out.
- The device is not compact as it has micro-controller and two sensors for which the orientation is difficult to set up in the first place which needs an understanding of the engineering technologies. This might create some

problems for the hospital staff. But, once set up, it does not need any further mechanical changes.

- Research must be carried out for the development of a hybrid sensor to measure both the high and low-flow of liquid to minimize the complexity of the current system.

Future Work

In this section, we have presented the future work for the researchers especially the implications of our model in the Industry 4.0 and Industrial IoT. This future work is an extension of our proposed model in Chapter 3 (Figure 7). The model that we have proposed needs validation by implementing it in the Industrial IoT scenario. The main improvements in this model are the security aspects that need to be addressed which were beyond the scope of our current project.

Implications of our System in Industry 4.0:

The term *Industrial Internet of things (IIoT)* or *Industrie 4.0,* announced by German Federal Government as one of the most important initiatives of modern era (Hermann, Pentek and Otto, 2016), is being widely used today for the revolution of the industry using Internet of Things, Business Intelligence, Data Analytics etc. Conservative estimations made by experts suggest that by 2030, $15 Trillion of the World GDP will be spent on the IIoT (Waitzinger, Ohlhausen, and Spath, 2015). The Industry 4.0 Initiative by Germany, called a 4th Industrial revolution, may revolutionize the industrial norms for better product and process customization. So, to turn Industrie 4.0 into reality, industrial practices need the efficient integration of novel technologies mentioned above.

A 5-step framework based on Cyber-Physical Systems (CPS) (Lee, Bagheri and Kao, 2015) was proposed for realizing the concept of Industrie 4.0. Their 5C concept had *Physical Components, Data to Information Conversion, Clustering, Information Visualization* and *Self-awareness*. Each step has its own significance and all the steps combined portrayed an efficient framework for the implementation of the smart factories and Industrie 4.0.

A framework for a smart factory in compliance with Industry 4.0 described in (Wang *et al.*, 2016) and (Schuh *et al.*, 2014) has four components. *Physical Objects* which includes machines and smart things which communicate with each other via *Industrial Network*. Enterprise Resource Platform (ERP) is implemented on *Cloud* which has all the available data from the *Smart Things*. The whole system is supervised by *Terminals for Supervision and Control* which helps in decision making regarding the product customization and process management. This whole system conforms with CPS (Drath and Horch, 2014) in which Physical Objects and Information Analytics are interconnected.

In this section, we propose a conceptual model that can be used for the implementation of IIoT for smart factories and Industry 4.0. Moreover, we also discuss our ongoing project for development of an integrated system for urine measurement in hospitals based on IoT.

Challenges for IIoT Implementation:

- Selection of Distributed Sensors that can efficiently perform the required tasks. Sometimes, the sensors are not commercially available, so, development of sensors and other hardware is needed.

- Data Filtration is another task that is the need of every industrial application. The data that is surplus to requirements should be eliminated.
- Data Privacy and Security should also be guaranteed because of the stiff competition in the market. If any data related to business model or production process is leaked, it can have serious financial consequences.
- Data Analytics also hold an important position in the implementation of IIoT or smart factories. The non-technical people need the data in the form that they can understand and can make decisions based on that. So, data should be analyzed according to the requirements.

Our proposed concept depicted in Figure 42 is based on the framework presented by (Wang *et al.*, 2016) with some changes to meet the challenges of Smart Factories and Industrie 4.0. Our conceptual model consists of 5 layers: *Physical Layer, Information Layer, Cyber Security, Data Analytics* and *Data Visualization.*

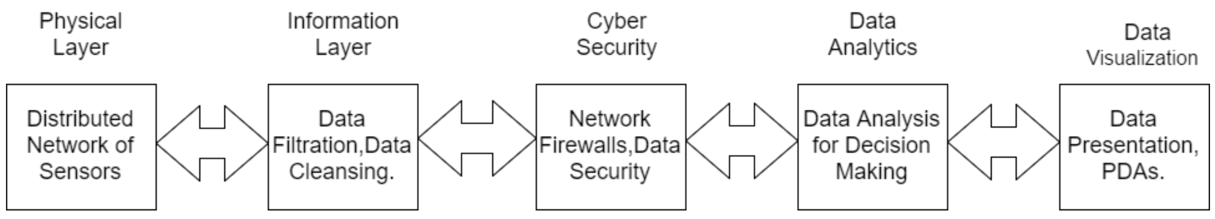

Physical Layer
Distributed Network of Sensors
Information Layer
Data Filtration,Data Cleansing.
Cyber Security
Network Firewalls,Data Security
Data Analytics
Data Analysis for Decision Making
Data Visualization
Data Presentation, PDAs.

Figure 42 Conceptual Model for Smart Factories and Industry 4.0

Physical Layer consists of a distributed network of sensors deployed according to the requirements of a specific factory. Each sensor has its own defined function and responsibility. The data collected from these sensors might not be clean and representable straight away. So, this data is then sent to the next layer that is

Information Layer where this data is filtered and converted into valuable information. The data is mostly confidential and must be secured. So, there is a *Cyber Security Layer* which deals with the security and privacy of the data. *Data Analytics* deals with the analysis of data in a certain way to show the relevant authorities with the information that they need to make efficient decisions about improving the production process or business process model of the factory. *Data Visualization* as clear from the name has the responsibility of representing the data to show the non-technical people the analyzed data in the previous layer.

We are in the process of implementing our conceptual model on a real-world industrial application. We are developing a urine measurement system for the use in hospitals for monitoring the urine output of the patients automatically using sensors and with the help of a micro-controller that data is then sent to a remote computer where it is analyzed for the physicians to monitor.

Due to the specific requirements of our application, we have modified our initial model to increase the efficiency. On the Physical Layer, we have deployed two sensors that will measure the high and low flow of the urine that is being collected from patients in a container using the catheter. Instead of Data Filtration at this stage, we first ensured the secure transmission of our data to a remote computer via Ethernet and Wi-Fi. On the remote computer, we are developing an analytics application that converts the signals from the sensors into meaningful information (Urine Volume) and analyze it based on hourly and daily requirements of the Doctors. Lastly, this information can be accessed remotely through Laptops and Mobile Phones through a web application.

Although, we are in process of finalizing the prototype, we anticipate that the results will be efficient enough for the device to be implemented in a real scenario.

This model can improve the diagnosis process in the hospitals and reduce the burden of the nursing staff, which, at present, collect the readings on hourly basis manually. Furthermore, our model has the potential to improve the healthcare process and can increase the reliability of the diagnosis.

REFERENCES

Badampudi, D., Wohlin, C. and Petersen, K. (2015) ‘Experiences from using snowballing and database searches in systematic literature studies’, *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering - EASE ’15*, pp. 1–10. doi: 10.1145/2745802.2745818.

Bellomo, R., Ronco, C., Kellum, J. A., Mehta, R. L. and Palevsky, P. (2004) ‘Acute renal failure–definition, outcome measures, animal models, fluid therapy and information technology needs: the Second International Consensus Conference of the Acute Dialysis Quality Initiative (ADQI) Group’, *Critical care*. BioMed Central, 8(4), p. R204.

Caragliu, A., Del Bo, C. and Nijkamp, P. (2011) ‘Smart Cities in Europe’, *Journal of Urban Technology*, 18(2), pp. 65–82. doi: 10.1080/10630732.2011.601117.

Chase, J. (2013) ‘The Evolution of the Internet of Things’, *Texas intrument*. Texas Instruments, p. 7.

Chelsey Fontaine, Stephen Tully, L. S. (2013) ‘Anti-reflux mechanism for urine collection systems’, *US8357105 B2*. doi: 10.1021/n10602701.

Cho, I. S., Chung, E. J. and others (2007) ‘Assessment of a Prototype Diagnostic Nursing Decision Support System for Inpatients with Type II Diabetes Mellitus’, in *Medinfo 2007: Proceedings of the 12th World Congress on Health (Medical) Informatics; Building Sustainable Health Systems*, p. 2167.

Chughtai, B., Forde, J. C., Thomas, D. D. M., Laor, L., Hossack, T., Woo, H. H., Te, A. E. and Kaplan, S. A. (2016) ‘Benign prostatic hyperplasia’, *Nature Reviews Disease Primers*. Macmillan Publishers Limited, 2, p. 16031. Available at: http://dx.doi.org/10.1038/nrdp.2016.31.

Devaraj, S., Ow, T. T. and Kohli, R. (2013) ‘Examining the impact of information technology and patient flow on healthcare performance: A Theory of Swift and even Flow

(TSEF) perspective', *Journal of Operations Management*. Elsevier B.V., 31(4), pp. 181–192. doi: 10.1016/j.jom.2013.03.001.

Dong, J., Ou, Z., Ylä-Jääski, A. and Cui, Y. (2013) 'Mobile hotspots cooperation towards better energy efficiency', *2013 IEEE Globecom Workshops, GC Wkshps 2013*, pp. 760–765. doi: 10.1109/GLOCOMW.2013.6825080.

Drath, R. and Horch, A. (2014) 'Industrie 4.0: Hit or hype?', *IEEE Industrial Electronics Magazine,* 8(2), pp. 56–58. doi: 10.1109/MIE.2014.2312079.

Galley, H. F. (2000) 'Can acute renal failure be prevented?', *Journal of the Royal College of Surgeons of Edinburgh,* 45(1).

Garrett, R. E. (1991) 'Principles of Siphons', *Journal of the World Aquaculture Society*. Blackwell Publishing Ltd, 22(1), pp. 1–9. doi: 10.1111/j.1749-7345.1991.tb00710.x.

Gelenbe, E. and Caseau, Y. (2015) 'The impact of information technology on energy consumption and carbon emissions', *Ubiquity,* 2015(June), pp. 1–15. doi: 10.1145/2755977.

Greenwald, F. F. (2012) 'REAL TIME URINE MONITORING SYSTEM', 2(12), p. 1799.

Grover, C. and Barney, K. (2003) 'Operating safely in surgery and critical care with perioperative automation', *Journal of healthcare information management: JHIM,* 18(3), pp. 56–61.

Hersch, M., Einav, S. and Izbicki, G. (2009) 'Accuracy and ease of use of a novel electronic urine output monitoring device compared with standard manual urinometer in the intensive care unit', *Journal of Critical Care*. Elsevier Inc., 24(4), p. 629.e13-629.e17. doi: 10.1016/j.jcrc.2008.12.008.

Hoy, W. K. and Adams, C. M. (2015) *Quantitative research in education: A primer*. Sage Publications.

Ilan Paz, Gush EtZion (IL); Harold Jacob, Cederhust, N. (2003) 'DROPLET COUNTER

FOR LOW FLOW MEASUREMENT', 2(12).

Jess, G., Pogatzki-Zahn, E. M., Zahn, P. K. and Meyer-Frieem, C. H. (2016) 'Monitoring heart rate variability to assess experimentally induced pain using the analgesia nociception index', *European Journal of Anaesthesiology*, 33(2), pp. 118–125. doi: 10.1097/EJA.0000000000000304.

KIOURTI, A., Lee, C. and Volakis, J. (2015) 'Fabrication of Textile Antennas and Circuits with 0.1 mm Precision', *IEEE Antennas and Wireless Propagation Letters*, PP(99), pp. 1–1. doi: 10.1109/LAWP.2015.2435257.

Lee, C. S., Kim, D. H. and Kim, J. D. (2014) 'An energy efficient active RFID protocol to avoid overhearing problem', *IEEE Sensors Journal*, 14(1), pp. 15–24. doi: 10.1109/JSEN.2013.2279391.

Lee, J., Bagheri, B. and Kao, H. A. (2015) 'A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems', *Manufacturing Letters*. Society of Manufacturing Engineers (SME), 3(December), pp. 18–23. doi: 10.1016/j.mfglet.2014.12.001.

Li, S., Xu, L. Da and Zhao, S. (2015) 'The internet of things: a survey', *Information Systems Frontiers*, 17(2), pp. 243–259. doi: 10.1007/s10796-014-9492-7.

MacEdo, E., Malhotra, R., Claure-Del Granado, R., Fedullo, P. and Mehta, R. L. (2011) 'Defining urine output criterion for acute kidney injury in critically ill patients', *Nephrology Dialysis Transplantation*, 26(2), pp. 509–515. doi: 10.1093/ndt/gfq332.

De Melo Bezerra, C. T., Vaz Cunha, L. C. and Libório, A. B. (2013) 'Defining reduced urine output in neonatal ICU: Importance for mortality and acute kidney injury classification', *Nephrology Dialysis Transplantation*, 28(4), pp. 901–909. doi: 10.1093/ndt/gfs604.

Miorandi, D., Sicari, S., De Pellegrini, F. and Chlamtac, I. (2012) 'Internet of things: Vision, applications and research challenges', *Ad Hoc Networks*. Elsevier B.V., 10(7), pp. 1497–1516. doi: 10.1016/j.adhoc.2012.02.016.

Molitoris, B. A. (ed.) (2015) *Critical Care Nephrology, Critical Care Clinics*. Remedica. doi: 10.1016/j.ccc.2015.07.001.

Mukherjee, A., Paul, H. S., Dey, S. and Banerjee, A. (2014) 'ANGELS for distributed analytics in IoT', *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*, pp. 565–570. doi: 10.1109/WF-IoT.2014.6803230.

Murray, P. T., Brady, H. R. and Hall, J. B. (eds) (2001) *Intensive Care Nephrology*, *Journal of the American Society of Nephrology*. doi: 10.1016/B978-1-4160-6640-8.00036-1.

Otero, A.Panigrahi B, Palacios F, Akhinfiev, T, R, F. (2009) *A Prototype Device to measure and supervise urine output of critical patients*. doi: 10.5772/60142.

Otero, A., Palacios, F., Akinfiev, T. and Apalkov, A. (2010) 'A low cost device for monitoring the urine output of critical care patients', *Sensors*, 10(12), pp. 10714–10732. doi: 10.3390/s101210714.

Pang, Z., Chen, Q., Han, W. and Zheng, L. (2015) 'Value-centric design of the internet-of-things solution for food supply chain: Value creation, sensor portfolio and information fusion', *Information Systems Frontiers*, 17(2), pp. 289–319. doi: 10.1007/s10796-012-9374-9.

Parikh, C. R. (2005) 'Urine IL-18 Is an Early Diagnostic Marker for Acute Kidney Injury and Predicts Mortality in the Intensive Care Unit', *Journal of the American Society of Nephrology*, 16(10), pp. 3046–3052. doi: 10.1681/ASN.2005030236.

PAULSEN, L., Elliott, S., McAdams, S., Fenoglietto, F. L. and Pisansky, A. (2016) *Devices, systems, and methods for obtaining and analyzing urine flow rate data using acoustics and software*. Google Patents. Available at: https://www.google.com/patents/US20160029942.

Ramos, R., O'Grady, M. and Chen, F. (2016) 'Urine Monitoring Systems and Methods'.

Google Patents. Available at: https://www.google.com/patents/US20160051176.

Ronco, C., Costanzo, M. R., Bellomo, R. and Maisel, A. S. (eds) (2010) *Fluid Overload Diagnosis and Management*. S. Karger AG.

Schuh, G., Pitsch, M., Rudolf, S., Karmann, W. and Sommer, M. (2014) 'Modular sensor platform for service-oriented cyber-physical systems in the european tool making industry', *Procedia CIRP*. Elsevier B.V., 17, pp. 374–379. doi: 10.1016/j.procir.2014.01.114.

SENSOR-INTEGRATED URINE BAG FOR REAL-TIME MEASURING Atigorn Sanguansri A thesis submitted in partial fulfilment of the requirements of Bournemouth University for the degree of Master by Research May 2016 Bournemouth University (2016).

Silverman, D. (2016) *Qualitative research*. Sage.

Singh, D. and Jara, A. J. (2014) 'A survey of Internet-of-Things : Future Vision , Architecture , Challenges and Services', *IEEE World Forum on Internet of Things (WF-IoT),* pp. 287–292.

Sippel, M., Collin, R. and Voges, K. F. (2000) 'Urine measuring device'. Google Patents. Available at: https://www.google.com/patents/US6129684.

Stol, K., Ralph, P. and Fitzgerald, B. (2016) 'Grounded Theory in Software Engineering Research : A Critical Review and Guidelines', *Proceedings of the 38th International Conference on Software Engineering - ICSE '16,* (Aug), pp. 120–131. doi: http://dx.doi.org/10.1145/2884781.2884833.

Sultan, N. (2014) 'Making use of cloud computing for healthcare provision: Opportunities and challenges', *International Journal of Information Management*. Elsevier Ltd, 34(2), pp. 177–184. doi: 10.1016/j.ijinfomgt.2013.12.011.

Sun, C. (2012) 'Application of RFID Technology for Logistics on Internet of Things', *AASRI Procedia,* 1, pp. 106–111. doi: 10.1016/j.aasri.2012.06.019.

Tsai, C. W., Lai, C. F., Chiang, M. C. and Yang, L. T. (2014) ‘Data mining for internet of things: A survey’, *IEEE Communications Surveys and Tutorials*, 16(1), pp. 77–97. doi: 10.1109/SURV.2013.103013.00206.

Voges, K. F., Sippel, M. and Collin, R. (2002) ‘Urine measuring device’. Google Patents. Available at: https://www.google.com/patents/US6348046.

Waitzinger, S., Ohlhausen, P. and Spath, D. (2015) ‘The industrial internet: Business models as challenges for innovations’, *23rd International Conference for Production Research, ICPR 2015*. Available at: http://www.scopus.com/inward/record.url?eid=2-s2.0-84949668461&partnerID=tZOtx3y1.

Wang, S., Wan, J., Li, D. and Zhang, C. (2016) ‘Implementing Smart Factory of Industrie 4.0: An Outlook’, *International Journal of Distributed Sensor Networks*, 2016. doi: 10.1155/2016/3159805.

Weber, R. H. (2010) ‘Internet of Things – New security and privacy challenges’, *Computer Law & Security Review*. Elsevier Ltd, 26(1), pp. 23–30. doi: 10.1016/j.clsr.2009.11.008.

Xu, L. Da, He, W. and Li, S. (2014) ‘Internet of things in industries: A survey’, *IEEE Transactions on Industrial Informatics*, 10(4), pp. 2233–2243. doi: 10.1109/TII.2014.2300753.

Zanella, a, Bui, N., Castellani, a, Vangelista, L. and Zorzi, M. (2014) ‘Internet of Things for Smart Cities’, *IEEE Internet of Things Journal*, 1(1), pp. 22–32. doi: 10.1109/JIOT.2014.2306328.

Zhang, D., Yang, L. T., Chen, M., Zhao, S., Guo, M. and Zhang, Y. (2014) ‘Real-Time Locating Systems Using Active RFID for Internet of Things’, *IEEE Systems Journal*, pp. 1–10. doi: 10.1109/JSYST.2014.2346625.

Zhu, C., Leung, V., Shu, L. and Ngai, E. (2015) ‘Green Internet of Things for Smart World’, *IEEE Access*, pp. 1–1. doi: 10.1109/ACCESS.2015.2497312.

APPENDICES

Appendix I Code for High-Flow-Sensor

```
#include "ThingSpeak.h"

//#define USE_WIFI101_SHIELD

#if defined(ARDUINO_AVR_YUN)

  #include "YunClient.h"

  YunClient client;

#else

 #if defined(USE_WIFI101_SHIELD) || defined(ARDUINO_SAMD_MKR1000) || defined(ARDUINO_ARCH_ESP8266)

  // Use WiFi

  #ifdef ARDUINO_ARCH_ESP8266

   #include <ESP8266WiFi.h>

  #else

   #include <SPI.h>

   #include <WiFi101.h>

  #endif

  char ssid[] = "<shani>";    //  your network SSID (name)

  char pass[] = "<Rushan11>";   // your network password

  int status = WL_IDLE_STATUS;
```

```
    WiFiClient  client;
  #elif defined(USE_ETHERNET_SHIELD)
    // Use wired ethernet shield
    #include <SPI.h>
    #include <Ethernet.h>
    byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
    EthernetClient client;
  #endif
#endif
 unsigned long myChannelNumber = 303355;
 const char * myWriteAPIKey = "YSKR6BO7B4DCSCXM";
 byte statusLed    = 13;
 byte sensorInterrupt = 1;  // 0 = digital pin 2
 byte sensorPin       = 3;
// The hall-effect flow sensor outputs approximately 4.5 pulses per second per
// litre/minute of flow.
float calibrationFactor = 7.5;
volatile byte pulseCount;
float flowRate;
 unsigned int flowMilliLitres;
 int totalMilliLitres;
```

```
unsigned long oldTime;


int count=0;//for Low-Flow

float liquid=0;

int sensorPina = A0; // select the input pin for LDR

int sensorValue = 0;

int prevReading = 0;// variable to store the value coming from the sensor

int currReading = 0;

//int count = 0;

int minReading = 450;     // minimum reading needed to count

int minFluctuation = 20;

void setup() {

  #ifdef ARDUINO_AVR_YUN

    Bridge.begin();

  #else

    #if    defined(ARDUINO_ARCH_ESP8266)    ||    defined(USE_WIFI101_SHIELD)    ||
defined(ARDUINO_SAMD_MKR1000)

      WiFi.begin(ssid, pass);

    #else

      Ethernet.begin(mac);

    #endif
  #endif


  ThingSpeak.begin(client);

  pinMode(statusLed, OUTPUT);

  digitalWrite(statusLed, HIGH);  // We have an active-low LED attached

  pinMode(sensorPin, INPUT);

  digitalWrite(sensorPin, HIGH);

  pulseCount        = 0;

  flowRate          = 0.0;

  flowMilliLitres   = 0;

  totalMilliLitres  = 0;

  oldTime           = 0;

  attachInterrupt(sensorInterrupt, pulseCounter, FALLING);

void loop() {

//int sensorValue = analogRead(sensorPin);

   if((millis() - oldTime) > 20000)

  {

    // Disable the interrupt while calculating flow rate and sending the value to

    // the host

    detachInterrupt(sensorInterrupt);

    // Because this loop may not complete in exactly 1 second intervals we calculate
    // the number of milliseconds that have passed since the last execution and use

    // that to scale the output. We also apply the calibrationFactor to scale the output

    // based on the number of pulses per second per units of measure (litres/minute in

    // this case) coming from the sensor.

    flowRate = ((1000.0 / (millis() - oldTime)) * pulseCount) / calibrationFactor;

    pulseCount=0;

    detachInterrupt(sensorInterrupt);

    // Note the time this processing pass was executed. Note that because we've

    // disabled interrupts the millis() function won't actually be incrementing right

    // at this point, but it will still return the value it was set to just before

    // interrupts went away.

    oldTime = millis();

    // Divide the flow rate in litres/minute by 3 to determine how many litres have

    // passed through the sensor in this 20 second interval, then multiply by 1000 to

    // convert to millilitres.

    flowMilliLitres = (flowRate / 3) * 1000;

    // Add the millilitres passed in this second to the cumulative total

    //totalMilliLitres += flowMilliLitres;

    ThingSpeak.writeField(myChannelNumber, 1, (long ) flowMilliLitres, myWriteAPIKey);

    attachInterrupt(sensorInterrupt, pulseCounter, FALLING);

    //unsigned int frac;
```

```
  // Serial.print("Flow rate: ");
    //Serial.print(int(flowRate));  // Print the integer part of the variable
    //Serial.print(".");             // Print the decimal point
    // Determine the fractional part. The 10 multiplier gives us 1 decimal place.
    //frac = (flowRate - int(flowRate)) * 10;
    // Enable the interrupt again now that we've finished sending output
    //Serial.print(frac, DEC) ;      // Print the fractional part of the variable
    //Serial.print("L/min");
    // Print the number of litres flowed in this second
    //Serial.print("  Current Liquid Flowing: ");             // Output separator
    //Serial.print(flowMilliLitres);
    //Serial.print("mL/Sec");
    // Print the cumulative total of litres flowed since starting
    //Serial.print("  Output Liquid Quantity: ");             // Output separator
    //Serial.print(totalMilliLitres);
    //Serial.println("mL");
  }
   //ThingSpeak.writeField(myChannelNumber, 2, liquid, myWriteAPIKey);
  //unsigned long tl = totalMilliLitres;
}
```

```
void pulseCounter()
{
  // Increment the pulse counter
  pulseCount++;
}
```

Appendix II Code for Low-Flow Sensor

```
#include "ThingSpeak.h"
//#define USE_WIFI101_SHIELD


#if defined(ARDUINO_AVR_YUN)
    #include "YunClient.h"
    YunClient client;
#else
  #if defined(USE_WIFI101_SHIELD) || defined(ARDUINO_SAMD_MKR1000) || defined(ARDUINO_ARCH_ESP8266)
    // Use WiFi
    #ifdef ARDUINO_ARCH_ESP8266
      #include <ESP8266WiFi.h>
    #else
      #include <SPI.h>
      #include <WiFi101.h>
    #endif
    char ssid[] = "<shani>";    //  your network SSID (name)
    char pass[] = "<Rushan11>";   // your network password
    int status = WL_IDLE_STATUS;
    WiFiClient  client;
  #elif defined(USE_ETHERNET_SHIELD)
    // Use wired ethernet shield
    #include <SPI.h>
    #include <Ethernet.h>
    byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
    EthernetClient client;
   #endif
 #endif
 unsigned long myChannelNumber =  303355;
 const char * myWriteAPIKey = "YSKR6BO7B4DCSCXM";
 int count=0;
 float liquid=0;
 int Led = 13; // define LED Interface
 int sensorPin = A0; // define the photo interrupter sensor interface
//int val; // define numeric variables val
```

```
//int sensorPin = A0; // select the input pin for LDR

int sensorValue = 0;

int prevReading = 0;// variable to store the value coming from the sensor

int currReading = 0;

//int count = 0;

int minReading = 400;     // minimum reading needed to count

int minFluctuation = 5;  // minimum fluctuation to count

void setup() {

  #ifdef ARDUINO_AVR_YUN

    Bridge.begin();

  #else

    #if defined(ARDUINO_ARCH_ESP8266) || defined(USE_WIFI101_SHIELD) ||
defined(ARDUINO_SAMD_MKR1000)

      WiFi.begin(ssid, pass);

    #else

      Ethernet.begin(mac);

    #endif

  #endif

  ThingSpeak.begin(client);

pinMode(Led, OUTPUT); // define LED as output interface

  Serial.begin(9600); //sets serial port for communication
  prevReading = analogRead(sensorPin);



  // use this if you want to wait until reading
  // reaches a threshold, else delete it
  while ( analogRead(sensorPin) < minReading)
  {
    Serial.println("Waiting ... ");
  }
}
void loop() {
  currReading = analogRead(sensorPin);
  Serial.println(currReading); //prints the values coming from the sensor on the screen
  // calculate fluctuation, watch for negative
  int fluctuation = (currReading > prevReading) ? currReading - prevReading : prevReading -
currReading;
  // if reading is over minimum and fluctuation is large enough to count
  if ( (currReading > minReading) && (fluctuation > minFluctuation) )
  {
    // add to count
    count++;
    Serial.print("Count:  ");
    Serial.println(count);
    liquid = count*0.05;
  } else {

    // flucuation too small
    // uncomment for debugging
    //    if( fluctuation <=  minFluctuation )
    //    {
    //      Serial.print("Fluctation too small:  ");
    //      Serial.println(fluctuation);
    //    }
    // reading too small
    // uncomment for debugging
    //    if( currReading <= minReading )
    //    {
    //      Serial.print("Reading too small:  ");
    //      Serial.println(currReading);
    //    }
  } // else
  // current reading becomes previous reading
```

```
  prevReading = currReading;

  ThingSpeak.writeField(myChannelNumber, 2, liquid, myWriteAPIKey);



  delay(3000);

}
```

Appendix III ThingSpeak Library for Arduino

```
 /*
          ThingSpeak(TM) Communication Library For Arduino and ESP8266
          Enables an Arduino or other compatible hardware to write or read data
        to or from ThingSpeak,
          an open data platform for the Internet of Things with MATLAB analytics
        and visualization.
          ThingSpeak ( https://www.thingspeak.com ) is an analytic IoT platform
        service that allows you to aggregate, visualize and
          analyze live data streams in the cloud.

          Copyright 2017, The MathWorks, Inc.

          See the accompaning licence file for licensing information.
        */
        /**
          @mainpage
         *
         * \ref ThingSpeakClass "For technical documentation, visit this page"
         *
         * ThingSpeak offers free data storage and analysis of time-stamped
        numeric or alphanumeric data.
         * Users can access ThingSpeak by visiting http://thingspeak.com and
        creating a ThingSpeak user account.
         *
         * ThingSpeak stores data in channels.  Channels support an unlimited
        number of timestamped observations (think of these as rows in a
        spreadsheet).
         * Each channel has up to 8 fields (think of these as columns in a
        speadsheet).  Check out this <a
        href="http://www.mathworks.com/videos/introduction-to-thingspeak-
        107749.html">video</a> for an overview.
         *
 * Channels may be public, where anyone can see the data, or private,
where only the owner and select users can read the data.
 * Each channel has an associated Write API Key that is used to control
who can write to a channel.
 * In addition, private channels have one or more Read API Keys to
control who can read from private channel.
 * An API Key is not required to read from public channels.  Each
channel can have up to 8 fields. One field is created by default.
 *
 * You can visualize and do online analytics of your data on ThingSpeak
using the built in version of MATLAB, or use the desktop version of
MATLAB to get
 * deeper historical insight.  Visit https://www.mathworks.com/hardware-
support/thingspeak.html to learn more.
 *
 * <h3>Compatible Hardware</h3>
 * * <a href="http://www.arduino.cc">Arduino/Genuino</a> or compatible
using a WiFi101 or Ethernet shield (we have tested with <a
href="http://www.arduino.cc/en/Main/ArduinoBoardUno">Uno</a> and <a
href="http://www.arduino.cc/en/Main/ArduinoBoardMega2560">Mega</a>)
 * * <a href="http://www.arduino.cc/en/Main/ArduinoBoardYun">Arduino
Yun</a> running OpenWRT-Yun Release 1.5.3 (November 13th, 2014) or
later.  There are known issues with earlier versions.  Visit [this
page](http://www.arduino.cc/en/Main/Software) to get the latest version.
 * * <a href="http://www.arduino.cc/en/Main/ArduinoMKR1000">Arduino
MKR1000</a>
 * * ESP8266 (tested with <a
href="https://www.sparkfun.com/products/13711">SparkFun ESP8266 Thing -
Dev Board</a> and <a href="http://www.seeedstudio.com/depot/NodeMCU-v2-
Lua-based-ESP8266-development-kit-p-2415.html">NodeMCU 1.0 module</a>)
  *
 * <h3>Examples</h3>
 * The library includes several examples to help you get started.  These
are accessible in the Examples/ThingSpeak menu off the File menu in the
Arduino IDE.
 * * <b>CheerLights:</b> Reads the latest <a
href="http://www.cheerlights.com">CheerLights</a> color on ThingSpeak,
and sets an RGB LED.
 * * <b>ReadLastTemperature:</b> Reads the latest temperature from the
public <a href="https://thingspeak.com/channels/12397">MathWorks weather
station</a> in Natick, MA on ThingSpeak.
 * * <b>ReadPrivateChannel:</b> Reads the latest voltage value from a
private channel on ThingSpeak.
 * * <b>ReadWeatherStation:</b> Reads the latest weather data from the
public <a href="https://thingspeak.com/channels/12397">MathWorks weather
station</a> in Natick, MA on ThingSpeak.
```

```
 * * <b>WriteMultipleVoltages:</b> Reads analog voltages from pins 0-7
and writes them to the 8 fields of a channel on ThingSpeak.
 * * <b>WriteVoltage:</b> Reads an analog voltage from pin 0, converts
to a voltage, and writes it to a channel on ThingSpeak.
 */
#ifndef ThingSpeak_h
#define ThingSpeak_h
//#define PRINT_DEBUG_MESSAGES
//#define PRINT_HTTP
#if defined(ARDUINO_ARCH_AVR) || defined(ARDUINO_ARCH_ESP8266) ||
defined(ARDUINO_ARCH_SAMD) || defined(ARDUINO_ARCH_SAM)
  #include "Arduino.h"
  #include <Client.h>
#else
  #error Only Arduino MKR1000, Yun, Uno/Mega/Due with either WiFi101 or
Ethernet shield. ESP8266 also supported.
#endif
#define THINGSPEAK_URL "api.thingspeak.com"
#define THINGSPEAK_IPADDRESS IPAddress(184,106,153,149)
#define THINGSPEAK_PORT_NUMBER 80
#ifdef ARDUINO_ARCH_AVR
    #ifdef ARDUINO_AVR_YUN
        #define TS_USER_AGENT "tslib-arduino/1.3 (arduino yun)"
    #else
        #define TS_USER_AGENT "tslib-arduino/1.3 (arduino uno or mega)"
    #endif
#elif defined(ARDUINO_ARCH_ESP8266)
    #define TS_USER_AGENT "tslib-arduino/1.3 (ESP8266)"
#elif defined(ARDUINO_SAMD_MKR1000)
       #define TS_USER_AGENT "tslib-arduino/1.3 (arduino mkr1000)"
#elif defined(ARDUINO_SAM_DUE)
       #define TS_USER_AGENT "tslib-arduino/1.3 (arduino due)"
#elif defined(ARDUINO_ARCH_SAMD) || defined(ARDUINO_ARCH_SAM)
       #define TS_USER_AGENT "tslib-arduino/1.3 (arduino unknown sam or
samd)"
#else
       #error "Platform not supported"
#endif
#define FIELDNUM_MIN 1
#define FIELDNUM_MAX 8
#define FIELDLENGTH_MAX 255  // Max length for a field in ThingSpeak is
255 bytes (UTF-8)
#define TIMEOUT_MS_SERVERRESPONSE 5000  // Wait up to five seconds for
server to respond
#define OK_SUCCESS              200     // OK / Success
```

```
#define ERR_BADAPIKEY          400     // Incorrect API key (or invalid
ThingSpeak server address)
#define ERR_BADURL             404     // Incorrect API key (or invalid
ThingSpeak server address)
#define ERR_OUT_OF_RANGE       -101    // Value is out of range or
string is too long (> 255 bytes)
#define ERR_INVALID_FIELD_NUM  -201    // Invalid field number
specified
#define ERR_SETFIELD_NOT_CALLED -210   // setField() was not called
before writeFields()
#define ERR_CONNECT_FAILED     -301    // Failed to connect to
ThingSpeak
#define ERR_UNEXPECTED_FAIL    -302    // Unexpected failure during
write to ThingSpeak
#define ERR_BAD_RESPONSE       -303    // Unable to parse response
#define ERR_TIMEOUT            -304    // Timeout waiting for server to
respond
#define ERR_NOT_INSERTED       -401    // Point was not inserted (most
probable cause is the rate limit of once every 15 seconds)
/**
 * @brief Enables an Arduino, ESP8266 or other compatible hardware to
write or read data to or from ThingSpeak, an open data platform for the
Internet of Things with MATLAB analytics and visualization.
 */
class ThingSpeakClass
{
  public:
        ThingSpeakClass()
        {
                resetWriteFields();
            this->lastReadStatus = OK_SUCCESS;
        };
        /**
         * @brief Initializes the ThingSpeak library and network settings
using a custom installation of ThingSpeak.
         * @param client EthernetClient, YunClient, TCPClient, or
WiFiClient created earlier in the sketch
         * @param customHostName Host name of a custom install of
ThingSpeak
         * @param port Port number to use with a custom install of
ThingSpeak
         * @return Always returns true
     * @comment This does not validate the information passed in, or
generate any calls to ThingSpeak.
         * @code
                #include <SPI.h>
```

```
               #include <Ethernet.h>
               byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
               EthernetClient client;
               #include "ThingSpeak.h"
               void setup() {
                 Ethernet.begin(mac);
                 ThingSpeak.begin(client,"api.thingspeak.com", 80);
               }
         * @endcode
         */
    bool begin(Client & client, const char * customHostName, unsigned
int port)
       {
#ifdef PRINT_DEBUG_MESSAGES
               Serial.print("ts::tsBegin    (client: Client URL: ");
Serial.print(customHostName); Serial.println(")");
#endif
       this->setClient(&client);
       this->setServer(customHostName, port);
       resetWriteFields();
       this->lastReadStatus = OK_SUCCESS;
       return true;
       };
       /**
        * @brief Initializes the ThingSpeak library and network settings
using a custom installation of ThingSpeak.
        * @param client EthernetClient, YunClient, TCPClient, or
WiFiClient created earlier in the sketch
        * @param customIP IP address of a custom install of ThingSpeak
        * @param port Port number to use with a custom install of
ThingSpeak
        * @return Always returns true
     * @comment This does not validate the information passed in, or
generate any calls to ThingSpeak.
        * @code
               #include <SPI.h>
               #include <Ethernet.h>
               byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
               EthernetClient client;
               #include "ThingSpeak.h"
               void setup() {
                 Ethernet.begin(mac);
                 ThingSpeak.begin(client,IPAddress(184,106,153,149),
80);
               }
        * @endcode
```

```
         */
        bool begin(Client & client, IPAddress customIP, unsigned int
port)
        {
#ifdef PRINT_DEBUG_MESSAGES
                Serial.print("ts::tsBegin    (client: Client IP: ");
Serial.print(customIP); Serial.println(")");
#endif
                this->setClient(&client);
                this->setServer(customIP, port);
                resetWriteFields();
                this->lastReadStatus = OK_SUCCESS;
                return true;
        };
        /**
         * @brief Initializes the ThingSpeak library and network settings
using the ThingSpeak.com service.
         * @param client EthernetClient, YunClient, TCPClient, or
WiFiClient created earlier in the sketch
         * @return Always returns true
     * @comment This does not validate the information passed in, or
generate any calls to ThingSpeak.
         * @code
                #include <SPI.h>
                #include <Ethernet.h>
                byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
                EthernetClient client;
                #include "ThingSpeak.h"
                void setup() {
                  Ethernet.begin(mac);
                  ThingSpeak.begin(client);
                }
         * @endcode
         */
        bool begin(Client & client)
        {
#ifdef PRINT_DEBUG_MESSAGES
                Serial.print("ts::tsBegin");
#endif
                this->setClient(&client);
                this->setServer();
                resetWriteFields();
                this->lastReadStatus = OK_SUCCESS;
                return true;
        };
        /**
         * @brief Write an integer value to a single field in a
ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to write
to.
         * @param value Integer value (from -32,768 to 32,767) to write.
         * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Visit https://thingspeak.com/docs/channels for more
information about channels, API keys, and fields.  ThingSpeak limits the
number of writes to a channel to once every 15 seconds.
         * @code
                void loop() {
                        int sensorValue = analogRead(A0);
                        ThingSpeak.writeField(myChannelNumber, 1,
sensorValue, myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
        int writeField(unsigned long channelNumber, unsigned int field,
int value, const char * writeAPIKey)
        {
                char valueString[10];  // int range is -32768 to 32768,
so 7 bytes including terminator, plus a little extra
                itoa(value, valueString, 10);
                return writeField(channelNumber, field, valueString,
writeAPIKey);
        };
        /**
         * @brief Write a long value to a single field in a ThingSpeak
channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to write
to.
         * @param value Long value (from -2,147,483,648 to 2,147,483,647)
to write.
         * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
```

```
        * @remark Visit https://thingspeak.com/docs/channels for more
information about channels, API keys, and fields.  ThingSpeak limits the
number of writes to a channel to once every 15 seconds.
        * @code
              void loop() {
                     int sensorValue = analogRead(A0);
                     ThingSpeak.writeField(myChannelNumber, 1,
sensorValue, myWriteAPIKey);
                     delay(20000);
              }
        * @endcode
        */
       int writeField(unsigned long channelNumber, unsigned int field,
long value, const char * writeAPIKey)
       {
              char valueString[15];  // long range is -2147483648 to
2147483647, so 12 bytes including terminator
              ltoa(value, valueString, 10);
              return writeField(channelNumber, field, valueString,
writeAPIKey);
       };
       /**
        * @brief Write a floating point value to a single field in a
ThingSpeak channel
        * @param channelNumber Channel number
        * @param field Field number (1-8) within the channel to write
to.
        * @param value Floating point value (from -999999000000 to
999999000000) to write.  If you need more accuracy, or a wider range,
you should format the number using <tt>dtostrf</tt> and writeField().
        * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
        * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
        * @remark Visit https://thingspeak.com/docs/channels for more
information about channels, API keys, and fields.  ThingSpeak limits the
number of writes to a channel to once every 15 seconds.
        * @code
              void loop() {
                     int sensorValue = analogRead(A0);
                     float voltage = sensorValue * (5.0 / 1023.0);
                     ThingSpeak.writeField(myChannelNumber, 1, voltage,
myWriteAPIKey);
                     delay(20000);
              }
        * @endcode
```

```
         */
        int writeField(unsigned long channelNumber, unsigned int field,
float value, const char * writeAPIKey)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::writeField (channelNumber: ");
Serial.print(channelNumber); Serial.print(" writeAPIKey: ");
Serial.print(writeAPIKey); Serial.print(" field: ");
Serial.print(field); Serial.print(" value: "); Serial.print(value,5);
Serial.println(")");
                #endif
                char valueString[20]; // range is -999999000000.00000 to
999999000000.00000, so 19 + 1 for the terminator
                int status = convertFloatToChar(value, valueString);
                if(status != OK_SUCCESS) return status;
                return writeField(channelNumber, field, valueString,
writeAPIKey);
        };
        /**
         * @brief Write a string to a single field in a ThingSpeak
channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to write
to.
         * @param value String to write (UTF8 string).  ThingSpeak limits
this field to 255 bytes.
         * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Visit https://thingspeak.com/docs/channels for more
information about channels, API keys, and fields.  ThingSpeak limits the
number of writes to a channel to once every 15 seconds.
         * @code
                void loop() {
                        int sensorValue = analogRead(A0);
                        if (sensorValue > 512) {
                                ThingSpeak.writeField(myChannelNumber, 1,
"High", myWriteAPIKey);
                        }
                        else {
                                ThingSpeak.writeField(myChannelNumber, 1,
"Low", myWriteAPIKey);
                        }
                        delay(20000);
                }
         * @endcode
         */
        int writeField(unsigned long channelNumber, unsigned int field,
const char * value, const char * writeAPIKey)
        {
                return writeField(channelNumber, field, String(value),
writeAPIKey);
        };
        /**
         * @brief Write a String to a single field in a ThingSpeak
channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to write
to.
         * @param value Character array (zero terminated) to write
(UTF8).  ThingSpeak limits this field to 255 bytes.
         * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Visit https://thingspeak.com/docs/channels for more
information about channels, API keys, and fields.  ThingSpeak limits the
number of writes to a channel to once every 15 seconds.
         * @code
                void loop() {
                        int sensorValue = analogRead(A0);
                        String meaning;
                        if (sensorValue < 400) {
                                meaning = String("Too Cold!");
                        } else if (sensorValue > 600) {
                                meaning = String("Too Hot!");
                        } else {
                                meaning = String("Just Right");
                        }
                        ThingSpeak.writeField(myChannelNumber, 1, meaning,
myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
        int writeField(unsigned long channelNumber, unsigned int field,
String value, const char * writeAPIKey)
        {
                // Invalid field number specified
                if(field < FIELDNUM_MIN || field > FIELDNUM_MAX) return
ERR_INVALID_FIELD_NUM;
```

```
		// Max # bytes for ThingSpeak field is 255
		if(value.length() > FIELDLENGTH_MAX) return
ERR_OUT_OF_RANGE;

		#ifdef PRINT_DEBUG_MESSAGES
			Serial.print("ts::writeField (channelNumber: ");
Serial.print(channelNumber); Serial.print(" writeAPIKey: ");
Serial.print(writeAPIKey); Serial.print(" field: ");
Serial.print(field); Serial.print(" value: \""); Serial.print(value);
Serial.println("\")");
		#endif
		String postMessage = String("field") + String(field) +
"=" + value;
		return writeRaw(channelNumber, postMessage, writeAPIKey);
	};

	/**
	 * @brief Set the value of a single field that will be part of a
multi-field update.
	 * To write multiple fields at once, call setField() for each of
the fields you want to write, and then call writeFields()
	 * @param field Field number (1-8) within the channel to set
	 * @param value Integer value (from -32,768 to 32,767) to set.
	 * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
	 * @see setLatitude(), setLongitude(), setElevation(),
writeFields()
	 * @code
		void loop() {
			int sensor1Value = analogRead(A0);
			float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
			String sensor3Meaning;
			int sensor3Value = analogRead(A2);
			if (sensor3Value < 400) {
				sensor3Meaning = String("Too Cold!");
			} else if (sensor3Value > 600) {
				sensor3Meaning = String("Too Hot!");
			} else {
				sensor3Meaning = String("Just Right");
			}
			long timeRead = millis();
			ThingSpeak.setField(1, sensor1Value);
			ThingSpeak.setField(2, sensor2Voltage);
			ThingSpeak.setField(3, sensor3Meaning);
			ThingSpeak.setField(4, timeRead);
```

```
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
        int setField(unsigned int field, int value)
        {
                char valueString[10];  // int range is -32768 to 32768,
so 7 bytes including terminator
                itoa(value, valueString, 10);

                return setField(field, valueString);
        };
        /**
         * @brief Set the value of a single field that will be part of a
multi-field update.
         * To write multiple fields at once, call setField() for each of
the fields you want to write, and then call writeFields()
         * @param field Field number (1-8) within the channel to set
         * @param value Long value (from -2,147,483,648 to 2,147,483,647)
to write.
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @see setLatitude(), setLongitude(), setElevation(),
writeFields()
         * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
                        ThingSpeak.setField(3, sensor3Meaning);
                        ThingSpeak.setField(4, timeRead);
```

```
                      ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                      delay(20000);
               }
        * @endcode
        */
       int setField(unsigned int field, long value)
       {
              char valueString[15];  // long range is -2147483648 to
2147483647, so 12 bytes including terminator
           ltoa(value, valueString, 10);
              return setField(field, valueString);
       };
       /**
        * @brief Set the value of a single field that will be part of a
multi-field update.
        * To write multiple fields at once, call setField() for each of
the fields you want to write, and then call writeFields()
        * @param field Field number (1-8) within the channel to set
        * @param value Floating point value (from -999999000000 to
999999000000) to write.  If you need more accuracy, or a wider range,
you should format the number yourself (using <tt>dtostrf</tt>) and
setField() using the resulting string.
        * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
        * @see setLatitude(), setLongitude(), setElevation(),
writeFields()
        * @code
              void loop() {
                      int sensor1Value = analogRead(A0);
                      float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                      String sensor3Meaning;
                      int sensor3Value = analogRead(A2);
                      if (sensor3Value < 400) {
                             sensor3Meaning = String("Too Cold!");
                      } else if (sensor3Value > 600) {
                             sensor3Meaning = String("Too Hot!");
                      } else {
                             sensor3Meaning = String("Just Right");
                      }
                      long timeRead = millis();
                      ThingSpeak.setField(1, sensor1Value);
                      ThingSpeak.setField(2, sensor2Voltage);
                      ThingSpeak.setField(3, sensor3Meaning);
                      ThingSpeak.setField(4, timeRead);
                      ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                      delay(20000);
              }
         * @endcode
         */
    int setField(unsigned int field, float value)
       {
              char valueString[20]; // range is -999999000000.00000 to
999999000000.00000, so 19 + 1 for the terminator
              int status = convertFloatToChar(value, valueString);
              if(status != OK_SUCCESS) return status;
              return setField(field, valueString);
       };
       /**
         * @brief Set the value of a single field that will be part of a
multi-field update.
         * To write multiple fields at once, call setField() for each of
the fields you want to write, and then call writeFields()
         * @param field Field number (1-8) within the channel to set
         * @param value String to write (UTF8).  ThingSpeak limits this
to 255 bytes.
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @see setLatitude(), setLongitude(), setElevation(),
writeFields()
         * @code
              void loop() {
                      int sensor1Value = analogRead(A0);
                      float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                      String sensor3Meaning;
                      int sensor3Value = analogRead(A2);
                      if (sensor3Value < 400) {
                             sensor3Meaning = String("Too Cold!");
                      } else if (sensor3Value > 600) {
                             sensor3Meaning = String("Too Hot!");
                      } else {
                             sensor3Meaning = String("Just Right");
                      }
                      long timeRead = millis();
                      ThingSpeak.setField(1, sensor1Value);
                      ThingSpeak.setField(2, sensor2Voltage);
                      ThingSpeak.setField(3, sensor3Meaning);
                      ThingSpeak.setField(4, timeRead);
                      ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                      delay(20000);
              }
          * @endcode
          */
    int setField(unsigned int field, const char * value)
        {
              return setField(field, String(value));
        };
        /**
          * @brief Set the value of a single field that will be part of a
multi-field update.
          * To write multiple fields at once, call setField() for each of
the fields you want to write, and then call writeFields()
          * @param field Field number (1-8) within the channel to set
          * @param value String to write (UTF8).  ThingSpeak limits this
to 255 bytes.
          * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
          * @see setLatitude(), setLongitude(), setElevation(),
writeFields()
          * @code
              void loop() {
                      int sensor1Value = analogRead(A0);
                      float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                      String sensor3Meaning;
                      int sensor3Value = analogRead(A2);
                      if (sensor3Value < 400) {
                              sensor3Meaning = String("Too Cold!");
                      } else if (sensor3Value > 600) {
                              sensor3Meaning = String("Too Hot!");
                      } else {
                              sensor3Meaning = String("Just Right");
                      }
                      long timeRead = millis();
                      ThingSpeak.setField(1, sensor1Value);
                      ThingSpeak.setField(2, sensor2Voltage);
                      ThingSpeak.setField(3, sensor3Meaning);
                      ThingSpeak.setField(4, timeRead);
                      ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                      delay(20000);
              }
          * @endcode
        */
    int setField(unsigned int field, String value)
       {
              #ifdef PRINT_DEBUG_MESSAGES
                     Serial.print("ts::setField   (field: ");
Serial.print(field); Serial.print(" value: \""); Serial.print(value);
Serial.println("\")");
              #endif
              if(field < FIELDNUM_MIN || field > FIELDNUM_MAX) return
ERR_INVALID_FIELD_NUM;
              // Max # bytes for ThingSpeak field is 255 (UTF-8)
              if(value.length() > FIELDLENGTH_MAX) return
ERR_OUT_OF_RANGE;
              this->nextWriteField[field - 1] = value;
              return OK_SUCCESS;
       };
       /**
        * @brief Set the latitude of a multi-field update.
        * To record latitude, longitude and elevation of a write, call
setField() for each of the fields you want to write, setLatitude() /
setLongitude() / setElevation(), and then call writeFields()
        * @param latitude Latitude of the measurement (degrees N, use
negative values for degrees S)
        * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
        * @see setField(), setLongitude(), setElevation(), writeFields()
        * @code
              void loop() {
                     int sensor1Value = analogRead(A0);
                     float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                     String sensor3Meaning;
                     int sensor3Value = analogRead(A2);
                     if (sensor3Value < 400) {
                            sensor3Meaning = String("Too Cold!");
                     } else if (sensor3Value > 600) {
                            sensor3Meaning = String("Too Hot!");
                     } else {
                            sensor3Meaning = String("Just Right");
                     }
                     long timeRead = millis();
                     ThingSpeak.setField(1, sensor1Value);
                     ThingSpeak.setField(2, sensor2Voltage);
                     ThingSpeak.setField(3, sensor3Meaning);
                     ThingSpeak.setField(4, timeRead);
```

```
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        setLatitude(42.2833);
                        setLongitude(-71.3500);
                        setElevation(100);
                        delay(20000);
                }
         * @endcode
         */
        int setLatitude(float latitude)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::setLatitude(latitude: ");
Serial.print(latitude,3); Serial.println("\")");
                #endif
                this->nextWriteLatitude = latitude;
                return OK_SUCCESS;
        };
        /**
         * @brief Set the longitude of a multi-field update.
         * To record latitude, longitude and elevation of a write, call
setField() for each of the fields you want to write, setLatitude() /
setLongitude() / setElevation(), and then call writeFields()
         * @param longitude Longitude of the measurement (degrees E, use
negative values for degrees W)
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @see setField(), setLatitude(), setElevation(), writeFields()
         * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
                        ThingSpeak.setField(3, sensor3Meaning);
```

```
                                ThingSpeak.setField(4, timeRead);
                                setLatitude(42.2833);
                                setLongitude(-71.3500);
                                setElevation(100);
                                ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                                delay(20000);
                        }
         * @endcode
         */
        int setLongitude(float longitude)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::setLongitude(longitude: ");
Serial.print(longitude,3); Serial.println("\")");
                #endif
                this->nextWriteLongitude = longitude;
                return OK_SUCCESS;
        };
        /**
         * @brief Set the elevation of a multi-field update.
         * To record latitude, longitude and elevation of a write, call
setField() for each of the fields you want to write, setLatitude() /
setLongitude() / setElevation(), and then call writeFields()
         * @param elevation Elevation of the measurement (meters above
sea level)
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @see setField(), setLatitude(), setLongitude(), writeFields()
         * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
```

```
				ThingSpeak.setField(3, sensor3Meaning);
				ThingSpeak.setField(4, timeRead);
				setLatitude(42.2833);
				setLongitude(-71.3500);
				setElevation(100);
				ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
				delay(20000);
			}
	 * @endcode
	 */
	int setElevation(float elevation)
	{
		#ifdef PRINT_DEBUG_MESSAGES
			Serial.print("ts::setElevation(elevation: ");
Serial.print(elevation,3); Serial.println("\")");
		#endif
		this->nextWriteElevation = elevation;
		return OK_SUCCESS;
	};
	/**
	 * @brief Set the status of a multi-field update.
	 * To record a status message on a write, call setStatus() then
call writeFields(). Use status to provide additonal
	 * details when writing a channel update.  Additonally, status
can be used by the ThingTweet App to send a message to
	 * Twitter.
	 * @param status String to write (UTF8).  ThingSpeak limits this
to 255 bytes.
	 * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
	 * @see writeFields()
	 * @code
		void loop() {
			int sensor1Value = analogRead(A0);
			float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
			String sensor3Meaning;
			int sensor3Value = analogRead(A2);
			if (sensor3Value < 400) {
				sensor3Meaning = String("Too Cold!");
			} else if (sensor3Value > 600) {
				sensor3Meaning = String("Too Hot!");
			} else {
				sensor3Meaning = String("Just Right");
			}
```

```
                      long timeRead = millis();
                      ThingSpeak.setField(1, sensor1Value);
                      ThingSpeak.setField(2, sensor2Voltage);
                      ThingSpeak.setField(3, timeRead);
                      ThingSpeak.setStatus(sensor3Meaning);
                      ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                      delay(20000);
               }
        * @endcode
        */

    int setStatus(const char * status)
       {
               return setStatus(String(status));
       };
       /**
        * @brief Set the status of a multi-field update.
        * To record a status message on a write, call setStatus() then
call writeFields(). Use status to provide additonal
        * details when writing a channel update.  Additonally, status
can be used by the ThingTweet App to send a message to
        * Twitter.
        * @param status String to write (UTF8).  ThingSpeak limits this
to 255 bytes.
        * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
        * @see writeFields()
        * @code
               void loop() {
                      int sensor1Value = analogRead(A0);
                      float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                      String sensor3Meaning;
                      int sensor3Value = analogRead(A2);
                      if (sensor3Value < 400) {
                             sensor3Meaning = String("Too Cold!");
                      } else if (sensor3Value > 600) {
                             sensor3Meaning = String("Too Hot!");
                      } else {
                             sensor3Meaning = String("Just Right");
                      }
                      long timeRead = millis();
                      ThingSpeak.setField(1, sensor1Value);
                      ThingSpeak.setField(2, sensor2Voltage);
                      ThingSpeak.setField(3, timeRead);
```

```
                        ThingSpeak.setStatus(sensor3Meaning);
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
    int setStatus(String status)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::setStatus(status: ");
Serial.print(status); Serial.println("\")");
                #endif
                // Max # bytes for ThingSpeak field is 255 (UTF-8)
                if(status.length() > FIELDLENGTH_MAX) return
ERR_OUT_OF_RANGE;
                this->nextWriteStatus = status;
                return OK_SUCCESS;
        };

        /**
         * @brief Set the Twitter account and message to use for an
update to be tweeted.
         * To send a message to twitter call setTwitterTweet() then call
writeFields()
         * @param twitter Twitter account name as a String.
         * @param tweet Twitter message as a String (UTF-8) limited to
140 character.
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Prior to using this feature, a twitter account must be
linked to your ThingSpeak account. Do this by logging into ThingSpeak
and going to Apps, then ThingTweet and clicking Link Twitter Account.
         * @see writeFields(),getLastReadStatus()
         * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
                        ThingSpeak.setField(3, timeRead);

        ThingSpeak.setTwitterTweet("YourTwitterAccountName",sensor3Meanin
g);
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
        int setTwitterTweet(const char * twitter, const char * tweet)
        {
                return setTwitterTweet(String(twitter), String(tweet));
        };
        /**
         * @brief Set the Twitter account and message to use for an
update to be tweeted.
         * To send a message to twitter call setTwitterTweet() then call
writeFields()
         * @param twitter Twitter account name as a String.
         * @param tweet Twitter message as a String (UTF-8) limited to
140 character.
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Prior to using this feature, a twitter account must be
linked to your ThingSpeak account. Do this by logging into ThingSpeak
and going to Apps, then ThingTweet and clicking Link Twitter Account.
         * @see writeFields(),getLastReadStatus()
         * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
                        ThingSpeak.setField(3, timeRead);

        ThingSpeak.setTwitterTweet("YourTwitterAccountName",sensor3Meanin
g);
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
        int setTwitterTweet(String twitter, const char * tweet)
        {
                return setTwitterTweet(twitter, String(tweet));
        };
        /**
         * @brief Set the Twitter account and message to use for an
update to be tweeted.
         * To send a message to twitter call setTwitterTweet() then call
writeFields()
         * @param twitter Twitter account name as a String.
         * @param tweet Twitter message as a String (UTF-8) limited to
140 character.
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Prior to using this feature, a twitter account must be
linked to your ThingSpeak account. Do this by logging into ThingSpeak
and going to Apps, then ThingTweet and clicking Link Twitter Account.
         * @see writeFields(),getLastReadStatus()
         * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
                        ThingSpeak.setField(3, timeRead);

        ThingSpeak.setTwitterTweet("YourTwitterAccountName",sensor3Meanin
g);
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
        int setTwitterTweet(const char * twitter, String tweet)
        {
                return setTwitterTweet(String(twitter), tweet);
        };
        /**
         * @brief Set the Twitter account and message to use for an
update to be tweeted.
         * To send a message to twitter call setTwitterTweet() then call
writeFields()
         * @param twitter Twitter account name as a String.
         * @param tweet Twitter message as a String (UTF-8) limited to
140 character.
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Prior to using this feature, a twitter account must be
linked to your ThingSpeak account. Do this by logging into ThingSpeak
and going to Apps, then ThingTweet and clicking Link Twitter Account.
         * @see writeFields(),getLastReadStatus()
         * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
```

```
                      ThingSpeak.setField(1, sensor1Value);
                      ThingSpeak.setField(2, sensor2Voltage);
                      ThingSpeak.setField(3, timeRead);

       ThingSpeak.setTwitterTweet("YourTwitterAccountName",sensor3Meanin
g);
                      ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                      delay(20000);
               }
        * @endcode
        */
       int setTwitterTweet(String twitter, String tweet){
              #ifdef PRINT_DEBUG_MESSAGES
                      Serial.print("ts::setTwitterTweet(twitter: ");
Serial.print(twitter); Serial.print(", tweet: "); Serial.print(tweet);
Serial.println("\")");
              #endif
              // Max # bytes for ThingSpeak field is 255 (UTF-8)
              if((twitter.length() > FIELDLENGTH_MAX) ||
(tweet.length() > FIELDLENGTH_MAX)) return ERR_OUT_OF_RANGE;

              this->nextWriteTwitter = twitter;
              this->nextWriteTweet = tweet;

              return OK_SUCCESS;
       };

       /**
        * @brief Set the created-at date of a multi-field update.
        * To record created-at of a write, call setField() for each of
the fields you want to write, setCreatedAt(), and then call
writeFields()
        * @param createdAt Desired timestamp to be included with the
channel update as a String.  The timestamp string must be in the ISO
8601 format. Example "2017-01-12 13:22:54"
        * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
        * @remark Timezones can be set using the timezone hour offset
parameter. For example, a timestamp for Eastern Standard Time is: "2017-
01-12 13:22:54-05".  If no timezone hour offset parameter is used, UTC
time is assumed.
        * @see setField(), writeFields()
        * @code
              void loop() {
                      int sensor1Value = analogRead(A0);
                      float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                      String sensor3Meaning;
                      int sensor3Value = analogRead(A2);
                      if (sensor3Value < 400) {
                              sensor3Meaning = String("Too Cold!");
                      } else if (sensor3Value > 600) {
                              sensor3Meaning = String("Too Hot!");
                      } else {
                              sensor3Meaning = String("Just Right");
                      }
                      long timeRead = millis();
                      ThingSpeak.setField(1, sensor1Value);
                      ThingSpeak.setField(2, sensor2Voltage);
                      ThingSpeak.setField(3, sensor3Meaning);
                      ThingSpeak.setField(4, timeRead);
                      ThingSpeak.setCreatedAt("2017-01-06T13:56:28");
                      ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                      delay(20000);
              }
         * @endcode
         */
        int setCreatedAt(const char * createdAt)
        {
                return setCreatedAt(String(createdAt));
        }

/**
         * @brief Set the created-at date of a multi-field update.
         * To record created-at of a write, call setField() for each of
the fields you want to write, setCreatedAt(), and then call
writeFields()
         * @param createdAt Desired timestamp to be included with the
channel update as a String.  The timestamp string must be in the ISO
8601 format. Example "2017-01-12 13:22:54"
         * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
         * @remark Timezones can be set using the timezone hour offset
parameter. For example, a timestamp for Eastern Standard Time is: "2017-
01-12 13:22:54-05".  If no timezone hour offset parameter is used, UTC
time is assumed.
         * @see setField(), writeFields()
         * @code
              void loop() {
                      int sensor1Value = analogRead(A0);
```

```
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
                        ThingSpeak.setField(3, sensor3Meaning);
                        ThingSpeak.setField(4, timeRead);
                        ThingSpeak.setCreatedAt("2017-01-06T13:56:28");
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        delay(20000);
                }
         * @endcode
         */
        int setCreatedAt(String createdAt)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::setCreatedAt(createdAt: ");
Serial.print(createdAt); Serial.println("\")");
                #endif

                // the ISO 8601 format is too complicated to check for
valid timestamps here
                // we'll need to reply on the api to tell us if there is
a problem
                // Max # bytes for ThingSpeak field is 255 (UTF-8)
                if(createdAt.length() > FIELDLENGTH_MAX) return
ERR_OUT_OF_RANGE;
                this->nextWriteCreatedAt = createdAt;

                return OK_SUCCESS;
        }

        /**
         * @brief Write a multi-field update.
        * Call setField() for each of the fields you want to write,
setLatitude() / setLongitude() / setElevation(), and then call
writeFields()
        * @param channelNumber Channel number
        * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
        * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
        * @see setField(), setLatitude(), setLongitude(), setElevation()
        * @code
                void loop() {
                        int sensor1Value = analogRead(A0);
                        float sensor2Voltage = analogRead(A1) * (5.0 /
1023.0);
                        String sensor3Meaning;
                        int sensor3Value = analogRead(A2);
                        if (sensor3Value < 400) {
                                sensor3Meaning = String("Too Cold!");
                        } else if (sensor3Value > 600) {
                                sensor3Meaning = String("Too Hot!");
                        } else {
                                sensor3Meaning = String("Just Right");
                        }
                        long timeRead = millis();
                        ThingSpeak.setField(1, sensor1Value);
                        ThingSpeak.setField(2, sensor2Voltage);
                        ThingSpeak.setField(3, sensor3Meaning);
                        ThingSpeak.setField(4, timeRead);
                        setLatitude(42.2833);
                        setLongitude(-71.3500);
                        setElevation(100);
                        ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
                        delay(20000);
                }
        * @endcode
        */
        int writeFields(unsigned long channelNumber, const char *
writeAPIKey)
        {
                String postMessage = String("");
                bool fFirstItem = true;
                for(size_t iField = 0; iField < 8; iField++)
                {
                        if(this->nextWriteField[iField].length() > 0)
                        {
                              if(!fFirstItem)
                              {
                                     postMessage = postMessage +
String("&");
                              }
                              postMessage = postMessage + String("field")
+ String(iField + 1) + String("=") + this->nextWriteField[iField];
                              fFirstItem = false;
                              this->nextWriteField[iField] = "";
                       }
               }
               if(!isnan(nextWriteLatitude))
               {
                       if(!fFirstItem)
                       {
                              postMessage = postMessage + String("&");
                       }
                       postMessage = postMessage + String("lat=") +
String(this->nextWriteLatitude);
                       fFirstItem = false;
                       this->nextWriteLatitude = NAN;
               }
               if(!isnan(this->nextWriteLongitude))
               {
                       if(!fFirstItem)
                       {
                              postMessage = postMessage + String("&");
                       }
                       postMessage = postMessage + String("long=") +
String(this->nextWriteLongitude);
                       fFirstItem = false;
                       this->nextWriteLongitude = NAN;
               }
               if(!isnan(this->nextWriteElevation))
               {
                       if(!fFirstItem)
                       {
                              postMessage = postMessage + String("&");
                       }
                       postMessage = postMessage + String("elevation=") +
String(this->nextWriteElevation);
                       fFirstItem = false;
                       this->nextWriteElevation = NAN;
               }

               if(this->nextWriteStatus.length() > 0)
```

```
                {
                        if(!fFirstItem)
                        {
                                postMessage = postMessage + String("&");
                        }
                        postMessage = postMessage + String("status=") +
String(this->nextWriteStatus);
                        fFirstItem = false;
                        this->nextWriteStatus = "";
                }

                if(this->nextWriteTwitter.length() > 0)
                {
                        if(!fFirstItem)
                        {
                                postMessage = postMessage + String("&");
                        }
                        postMessage = postMessage + String("twitter=") +
String(this->nextWriteTwitter);
                        fFirstItem = false;
                        this->nextWriteTwitter = "";
                }

                if(this->nextWriteTweet.length() > 0)
                {
                        if(!fFirstItem)
                        {
                                postMessage = postMessage + String("&");
                        }
                        postMessage = postMessage + String("tweet=") +
String(this->nextWriteTweet);
                        fFirstItem = false;
                        this->nextWriteTweet = "";
                }

                if(this->nextWriteCreatedAt.length() > 0)
                {
                        if(!fFirstItem)
                        {
                                postMessage = postMessage + String("&");
                        }
                        postMessage = postMessage + String("created_at=")
+ String(this->nextWriteCreatedAt);
                        fFirstItem = false;
                        this->nextWriteCreatedAt = "";
                }
```

```
		if(fFirstItem)
		{
			// setField was not called before writeFields
			return ERR_SETFIELD_NOT_CALLED;
		}
		return writeRaw(channelNumber, postMessage, writeAPIKey);
	};
	/**
	 * @brief Write a raw POST to a ThingSpeak channel
	 * @param channelNumber Channel number
	 * @param postMessage Raw URL to write to ThingSpeak as a string.
See the documentation at
https://thingspeak.com/docs/channels#update_feed.
	 * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
	 * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
	 * @remark This is low level functionality that will not be
required by most users.
	 * @code
		void loop() {
			const char postMessage[] =
"field1=23&created_at=2014-12-31%2023:59:59";
			ThingSpeak.setField(1, sensor1Value);
			ThingSpeak.setField(2, sensor2Voltage);
			ThingSpeak.setField(3, sensor3Meaning);
			ThingSpeak.setField(4, timeRead);
			ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);
			delay(20000);
		}
	 * @endcode
	 */
	int writeRaw(unsigned long channelNumber, const char *
postMessage, const char * writeAPIKey)
	{
		return writeRaw(channelNumber, String(postMessage),
writeAPIKey);
	};
	/**
	 * @brief Write a raw POST to a ThingSpeak channel
	 * @param channelNumber Channel number
```

```
        * @param postMessage Raw URL to write to ThingSpeak as a String.
See the documentation at
https://thingspeak.com/docs/channels#update_feed.
        * @param writeAPIKey Write API key associated with the channel.
*If you share code with others, do _not_ share this key*
        * @return HTTP status code of 200 if successful.  See
getLastReadStatus() for other possible return values.
        * @remark This is low level functionality that will not be
required by most users.
        * @code
                void loop() {
                        String postMessage =
String("field1=23&created_at=2014-12-31%2023:59:59");
                        ThingSpeak.writeRaw(myChannelNumber, postMessage,
myWriteAPIKey);
                        delay(20000);
                }
        * @endcode
        */
        int writeRaw(unsigned long channelNumber, String postMessage,
const char * writeAPIKey)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::writeRaw   (channelNumber: ");
Serial.print(channelNumber); Serial.print(" writeAPIKey: ");
Serial.print(writeAPIKey); Serial.print(" postMessage: \"");
Serial.print(postMessage); Serial.println("\")");
                #endif
                if(!connectThingSpeak())
                {
                        // Failed to connect to ThingSpeak
                        return ERR_CONNECT_FAILED;
                }
                postMessage = postMessage + String("&headers=false");
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("               POST
\"");Serial.print(postMessage);Serial.println("\"");
                #endif
                postMessage = postMessage + String("\n");
                // Post data to thingspeak
                if(!this->client->print("POST /update HTTP/1.1\r\n"))
return abortWriteRaw();
                if(!writeHTTPHeader(writeAPIKey)) return abortWriteRaw();
                if(!this->client->print("Content-Type: application/x-www-
form-urlencoded\r\n")) return abortWriteRaw();
```

```
			if(!this->client->print("Content-Length: ")) return 
abortWriteRaw();
			if(!this->client->print(postMessage.length())) return 
abortWriteRaw();
			if(!this->client->print("\r\n\r\n")) return 
abortWriteRaw();
			if(!this->client->print(postMessage)) return 
abortWriteRaw();

			String entryIDText = String();
			int status = getHTTPResponse(entryIDText);
			if(status != OK_SUCCESS)
			{
				client->stop();
				return status;
			}
			long entryID = entryIDText.toInt();
			#ifdef PRINT_DEBUG_MESSAGES
			Serial.print("               Entry ID 
\"");Serial.print(entryIDText);Serial.print("\" 
(");Serial.print(entryID);Serial.println(")");
			#endif
			client->stop();

			#ifdef PRINT_DEBUG_MESSAGES
				Serial.println("disconnected.");
			#endif
			if(entryID == 0)
			{
				// ThingSpeak did not accept the write
				status = ERR_NOT_INSERTED;
			}
			return status;
		};

		/**
		 * @brief Read the latest string from a private ThingSpeak 
channel
		 * @param channelNumber Channel number
		 * @param field Field number (1-8) within the channel to read 
from.
		 * @param readAPIKey Read API key associated with the channel. 
*If you share code with others, do _not_ share this key*
		 * @return Value read (UTF8 string), or empty string if there is 
an error.  Use getLastReadStatus() to get more specific information.
		 * @code
```

```
                void loop() {
                  String message = 
ThingSpeak.readStringField(myChannelNumber, 1, myReadAPIKey);
                  Serial.print("Latest message is: "); 
                  Serial.println(message);
                  delay(30000);
                }
         * @endcode
         */
    String readStringField(unsigned long channelNumber, unsigned int 
field, const char * readAPIKey)
        {
                if(field < FIELDNUM_MIN || field > FIELDNUM_MAX)
                {
                        this->lastReadStatus = ERR_INVALID_FIELD_NUM;
                        return("");
                }
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::readStringField(channelNumber: 
"); Serial.print(channelNumber);
                        if(NULL != readAPIKey)
                        {
                                Serial.print(" readAPIKey: "); 
Serial.print(readAPIKey);
                        }
                        Serial.print(" field: "); Serial.print(field); 
Serial.println(")");
                #endif
                return readRaw(channelNumber, String(String("/fields/") + 
String(field) + String("/last")), readAPIKey);
        }
        /**
         * @brief Read the latest string from a public ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to read 
from.
         * @return Value read (UTF8), or empty string if there is an 
error.  Use getLastReadStatus() to get more specific information.
         * @code
                void loop() {
                  String message = 
ThingSpeak.readStringField(myChannelNumber, 1);
                  Serial.print("Latest message is: "); 
                  Serial.println(message);
                  delay(30000);
                }
         * @endcode
         */
        String readStringField(unsigned long channelNumber, unsigned int
field)
        {
                return readStringField(channelNumber, field, NULL);
        };
        /**
         * @brief Read the latest float from a private ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to read
from.
         * @param readAPIKey Read API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return Value read, or 0 if the field is text or there is an
error.  Use getLastReadStatus() to get more specific information.  Note
that NAN, INFINITY, and -INFINITY are valid results.
         * @code
                void loop() {
                  float voltage =
ThingSpeak.readFloatField(myChannelNumber, 1, myReadAPIKey);
                  Serial.print("Latest voltage is: ");
                  Serial.print(voltage);
                  Serial.println("V");
                  delay(30000);
                }
         * @endcode
         */
    float readFloatField(unsigned long channelNumber, unsigned int
field, const char * readAPIKey)
        {
                return
convertStringToFloat(readStringField(channelNumber, field, readAPIKey));
        };
        /**
         * @brief Read the latest float from a public ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to read
from.
         * @return Value read, or 0 if the field is text or there is an
error.  Use getLastReadStatus() to get more specific information.  Note
that NAN, INFINITY, and -INFINITY are valid results.
         * @code
                void loop() {
                  float voltage =
ThingSpeak.readFloatField(myChannelNumber, 1);
```

```
                Serial.print("Latest voltage is: ");
                Serial.print(voltage);
                Serial.println("V");
                delay(30000);
              }
         * @endcode
         */
    float readFloatField(unsigned long channelNumber, unsigned int
field)
        {
                return readFloatField(channelNumber, field, NULL);
        };
        /**
         * @brief Read the latest long from a private ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to read
from.
         * @param readAPIKey Read API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return Value read, or 0 if the field is text or there is an
error.  Use getLastReadStatus() to get more specific information.
         * @code
               void loop() {
                long value = ThingSpeak.readLongField(myChannelNumber,
1, myReadAPIKey);
                Serial.print("Latest value is: ");
                Serial.print(value);
                delay(30000);
              }
         * @endcode
         */
    long readLongField(unsigned long channelNumber, unsigned int field,
const char * readAPIKey)
        {
         // Note that although the function is called "toInt" it really
returns a long.
                return readStringField(channelNumber, field,
readAPIKey).toInt();
        }
        /**
         * @brief Read the latest long from a public ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to read
from.
         * @return Value read, or 0 if the field is text or there is an
error.  Use getLastReadStatus() to get more specific information.
```

```
         * @code
               void loop() {
                 long value = ThingSpeak.readLongField(myChannelNumber,
1);
                 Serial.print("Latest value is: ");
                 Serial.print(value);
                 delay(30000);
               }
         * @endcode
         */
        long readLongField(unsigned long channelNumber, unsigned int
field)
        {
                return readLongField(channelNumber, field, NULL);
        };
        /**
         * @brief Read the latest int from a private ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to read
from.
         * @param readAPIKey Read API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return Value read, or 0 if the field is text or there is an
error.  Use getLastReadStatus() to get more specific information.
         * @remark If the value returned is out of range for an int, the
result is undefined.
         * @code
               void loop() {
                 int value = ThingSpeak.readIntField(myChannelNumber, 1,
myReadAPIKey);
                 Serial.print("Latest value is: ");
                 Serial.print(value);
                 delay(30000);
               }
         * @endcode
         */
    int readIntField(unsigned long channelNumber, unsigned int field,
const char * readAPIKey)
        {
                return readLongField(channelNumber, field, readAPIKey);
        }
        /**
         * @brief Read the latest int from a public ThingSpeak channel
         * @param channelNumber Channel number
         * @param field Field number (1-8) within the channel to read
from.
```

```
        * @return Value read, or 0 if the field is text or there is an
error.  Use getLastReadStatus() to get more specific information.
        * @remark If the value returned is out of range for an int, the
result is undefined.
        * @code
              void loop() {
                int value = ThingSpeak.readIntField(myChannelNumber,
1);
                Serial.print("Latest value is: ");
                Serial.print(value);
                delay(30000);
              }
        * @endcode
        */
    int readIntField(unsigned long channelNumber, unsigned int field)
       {
              return readLongField(channelNumber, field, NULL);
       };
       /**
        * @brief Read the latest status from a private ThingSpeak
channel
        * @param channelNumber Channel number
        * @param readAPIKey Read API key associated with the channel.
*If you share code with others, do _not_ share this key*
        * @return Value read (UTF8 string). An empty string is returned
if there was no status written to the channel or in case of an error.
Use getLastReadStatus() to get more specific information.
        * @code
              void loop() {
                String value = ThingSpeak.readStatus(myChannelNumber,
myReadAPIKey);
                Serial.print("Latest status is: ");
                Serial.print(value);
                delay(30000);
              }
        * @endcode
        */
       String readStatus(unsigned long channelNumber, const char *
readAPIKey)
       {
              String content = readRaw(channelNumber,
"/feeds/last.txt?status=true", readAPIKey);

              if(getLastReadStatus() != OK_SUCCESS){
                     return String("");
              }
```

```
		return getJSONValueByKey(content, "status");
	};

		/**
	 * @brief Read the latest status from a public ThingSpeak channel
	 * @param channelNumber Channel number
	 * @return Value read (UTF8 string). An empty string is returned
if there was no status written to the channel or in case of an error.
Use getLastReadStatus() to get more specific information.
	 * @code
		void loop() {
		  String value = ThingSpeak.readStatus(myChannelNumber,
myReadAPIKey);
		  Serial.print("Latest status is: ");
		  Serial.print(value);
		  delay(30000);
		}
	 * @endcode
	 */
	String readStatus(unsigned long channelNumber)
	{
		return readStatus(channelNumber, NULL);
	};

	/**
	 * @brief Read the created-at timestamp associated with the
latest update to a private ThingSpeak channel
	 * @param channelNumber Channel number
	 * @param readAPIKey Read API key associated with the channel.
*If you share code with others, do _not_ share this key*
	 * @return Value read (UTF8 string). An empty string is returned
if there was no created-at timestamp written to the channel or in case
of an error.  Use getLastReadStatus() to get more specific information.
	 * @code
		void loop() {
		  String value =
ThingSpeak.readCreatedAt(myChannelNumber);
		  Serial.print("Latest update timestamp is: ");
		  Serial.print(value);
		  delay(30000);
		}
	 * @endcode
	 */
	String readCreatedAt(unsigned long channelNumber, const char *
readAPIKey)
```

```
	{
		String content = readRaw(channelNumber,
"/feeds/last.txt", readAPIKey);

		if(getLastReadStatus() != OK_SUCCESS){
			return String("");
		}

		return getJSONValueByKey(content, "created_at");
	};
	/**
	 * @brief Read the created-at timestamp associated with the
latest update to a private ThingSpeak channel
	 * @param channelNumber Channel number
	 * @return Value read (UTF8 string). An empty string is returned
if there was no created-at timestamp written to the channel or in case
of an error.  Use getLastReadStatus() to get more specific information.
	 * @code
		void loop() {
		  String value =
ThingSpeak.readCreatedAt(myChannelNumber);
		  Serial.print("Latest update timestamp is: ");
		  Serial.print(value);
		  delay(30000);
		}
	 * @endcode
	 */
	String readCreatedAt(unsigned long channelNumber)
	{
		return readCreatedAt(channelNumber, NULL);
	};

	/**
	 * @brief Read a raw response from a public ThingSpeak channel
	 * @param channelNumber Channel number
	 * @param URLSuffix Raw URL to write to ThingSpeak as a String.
See the documentation at https://thingspeak.com/docs/channels#get_feed
	 * @return Response if successful, or empty string.	Use
getLastReadStatus() to get more specific information.
	 * @remark This is low level functionality that will not be
required by most users.
	 * @code
		void loop() {
		  String response = ThingSpeak.readRaw(myChannelNumber,
String("feeds/days=1"));
		  Serial.print("Response: ");
```

```
              Serial.print(response);
              delay(30000);
            }
         * @endcode
         */
        String readRaw(unsigned long channelNumber, String URLSuffix)
        {
                return readRaw(channelNumber, URLSuffix, NULL);
        }

        /**
         * @brief Read a raw response from a private ThingSpeak channel
         * @param channelNumber Channel number
         * @param URLSuffix Raw URL to write to ThingSpeak as a String.
See the documentation at https://thingspeak.com/docs/channels#get_feed
         * @param readAPIKey Read API key associated with the channel.
*If you share code with others, do _not_ share this key*
         * @return Response if successful, or empty string.        Use
getLastReadStatus() to get more specific information.
         * @remark This is low level functionality that will not be
required by most users.
         * @code
                void loop() {
                  String response = ThingSpeak.readRaw(myChannelNumber,
String("feeds/days=1"), myReadAPIKey);
                  Serial.print("Response: ");
                  Serial.print(response);
                  delay(30000);
                }
         * @endcode
         */
        String readRaw(unsigned long channelNumber, String URLSuffix,
const char * readAPIKey)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::readRaw   (channelNumber: ");
Serial.print(channelNumber);
                        if(NULL != readAPIKey)
                        {
                                Serial.print(" readAPIKey: ");
Serial.print(readAPIKey);
                        }
                        Serial.print(" URLSuffix: \"");
Serial.print(URLSuffix); Serial.println("\")");
                #endif
                if(!connectThingSpeak())
                {
                        this->lastReadStatus = ERR_CONNECT_FAILED;
                        return String("");
                }
                String URL = String("/channels/") + String(channelNumber)
+ URLSuffix;
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("                GET
\"");Serial.print(URL);Serial.println("\"");
                #endif
                // Post data to thingspeak
                if(!this->client->print("GET ")) return abortReadRaw();
                if(!this->client->print(URL)) return abortReadRaw();
                if(!this->client->print(" HTTP/1.1\r\n")) return
abortReadRaw();
                if(!writeHTTPHeader(readAPIKey)) return abortReadRaw();
                if(!this->client->print("\r\n")) return abortReadRaw();

                String content = String();
                int status = getHTTPResponse(content);

                this->lastReadStatus = status;
                #ifdef PRINT_DEBUG_MESSAGES
                        if(status == OK_SUCCESS)
                        {
                                Serial.print("Read: \"");
Serial.print(content); Serial.println("\"");
                        }
                #endif
                client->stop();
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.println("disconnected.");
                #endif
                if(status != OK_SUCCESS)
                {
                        // return status;
                        return String("");
                }
         // This is a workaround to a bug in the Spark implementation of
String
        return String("") + content;
        };

        /**
         * @brief Get the status of the previous read.
```

```
	 * @return Generally, these are HTTP status codes.  Negative
values indicate an error generated by the library.
	 * Possible response codes:
	 *  * 200: OK / Success
	 *  * 404: Incorrect API key (or invalid ThingSpeak server
address)
	 *  * -101: Value is out of range or string is too long (> 255
characters)
	 *  * -201: Invalid field number specified
	 *  * -210: setField() was not called before writeFields()
	 *  * -301: Failed to connect to ThingSpeak
	 *  * -302: Unexpected failure during write to ThingSpeak
	 *  * -303: Unable to parse response
      *  * -304: Timeout waiting for server to respond
	 *  * -401: Point was not inserted (most probable cause is the
rate limit of once every 15 seconds)
	 * @remark The read functions will return zero or empty if there
is an error.  Use this function to retrieve the details.
	 * @code
		void loop() {
		  String message =
ThingSpeak.readStringField(myChannelNumber, 1);
		  int resultCode = ThingSpeak.getLastReadStatus();
		  if(resultCode == 200)
		  {
			Serial.print("Latest message is: ");
			Serial.println(message);
		  }
		  else
		  {
			Serial.print("Error reading message.  Status
was: ");
			Serial.println(resultCode);
		  }
		  delay(30000);
		}
	 * @endcode
	 */
	int getLastReadStatus()
	{
		return this->lastReadStatus;
	};
private:

	String getJSONValueByKey(String textToSearch, String key)
	{
			if(textToSearch.length() == 0){
				return String("");
			}

			String searchPhrase = String("\"") + key +
String("\":\"");

			int fromPosition = textToSearch.indexOf(searchPhrase,0);

			if(fromPosition == -1){
				// return because there is no status or it's null
				return String("");
			}

			fromPosition = fromPosition + searchPhrase.length();

			int toPosition = textToSearch.indexOf("\"",
fromPosition);


			if(toPosition == -1){
				// return because there is no end quote
				return String("");
			}

			textToSearch.remove(toPosition);

			return textToSearch.substring(fromPosition);
		}

    int abortWriteRaw()
    {
        this->client->stop();
        return ERR_UNEXPECTED_FAIL;
    }
    String abortReadRaw()
    {
			this->client->stop();
			#ifdef PRINT_DEBUG_MESSAGES
				Serial.println("ReadRaw abort - disconnected.");
			#endif
			this->lastReadStatus = ERR_UNEXPECTED_FAIL;
			return String("");
    }
	void setServer(const char * customHostName, unsigned int port)
	{
```

```
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::setServer  (URL: \"");
Serial.print(customHostName); Serial.println("\")");
                #endif
                this->customIP = INADDR_NONE;
                this->customHostName = customHostName;
         this->port = port;
        };
        void setServer(IPAddress customIP, unsigned int port)
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::setServer  (IP: \"");
Serial.print(customIP); Serial.println("\")");
                #endif
                this->customIP = customIP;
                this->customHostName = NULL;
         this->port = port;
        };
        void setServer()
        {
                #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("ts::setServer  (default)");
                #endif
                this->customIP = INADDR_NONE;
                this->customHostName = NULL;
         this->port = THINGSPEAK_PORT_NUMBER;
        };
        void setClient(Client * client) {this->client = client;};
        Client * client = NULL;
        const char * customHostName = NULL;
        IPAddress customIP = INADDR_NONE;
    unsigned int port = THINGSPEAK_PORT_NUMBER;
        String nextWriteField[8];
        float nextWriteLatitude;
        float nextWriteLongitude;
        float nextWriteElevation;
        int lastReadStatus;
        String nextWriteStatus;
        String nextWriteTwitter;
        String nextWriteTweet;
        String nextWriteCreatedAt;
        bool connectThingSpeak()
        {
                bool connectSuccess = false;
                if(this->customIP == INADDR_NONE && NULL == this-
>customHostName)
```

```
        {
                    #ifdef PRINT_DEBUG_MESSAGES
                        Serial.print("                Connect to
default ThingSpeak URL...");
                    #endif
                    connectSuccess = client-
>connect(THINGSPEAK_URL,THINGSPEAK_PORT_NUMBER);
            if(!connectSuccess)
            {
                        #ifdef PRINT_DEBUG_MESSAGES
                            Serial.print("Failed. Try default IP...");
                        #endif
                        connectSuccess = client-
>connect(THINGSPEAK_IPADDRESS,THINGSPEAK_PORT_NUMBER);
            }
       }
       else
       {
                  if(!(this->customIP == INADDR_NONE))
                  {
                        // Connect to the server on port 80 (HTTP) at
the customIP address
                        #ifdef PRINT_DEBUG_MESSAGES
                          Serial.print("                Connect to
");Serial.print(this->customIP);Serial.print("...");
                        #endif
                        connectSuccess = client->connect(this-
>customIP,this->port);
                  }
                  if(NULL != this->customHostName)
                  {
                        // Connect to the server on port 80 (HTTP) at
the URL address
                        #ifdef PRINT_DEBUG_MESSAGES
                               Serial.print("                Connect to
");Serial.print(this->customHostName);Serial.print(" ...");
                        #endif
                        connectSuccess = client-
>connect(customHostName,this->port);
                  }
        }
               #ifdef PRINT_DEBUG_MESSAGES
               if (connectSuccess)
               {
                    Serial.println("Success.");
               }
```

```
                else
                {
                        Serial.println("Failed.");
                }
                #endif
                return connectSuccess;
        };
        bool writeHTTPHeader(const char * APIKey)
        {
         if(NULL != this->customHostName)
         {
                     if (!this->client->print("Host: ")) return false;
                     if (!this->client->print(this->customHostName))
return false;
                     if (!this->client->print("\r\n")) return false;
         }
         else
         {
                     if (!this->client->print("Host:
api.thingspeak.com\r\n")) return false;
         }
                if (!this->client->print("Connection: close\r\n")) return
false;
                if (!this->client->print("User-Agent: ")) return false;
                if (!this->client->print(TS_USER_AGENT)) return false;
                if (!this->client->print("\r\n")) return false;
                if(NULL != APIKey)
                {
                        if (!this->client->print("X-THINGSPEAKAPIKEY: "))
return false;
                        if (!this->client->print(APIKey)) return false;
                        if (!this->client->print("\r\n")) return false;
                }
                return true;
        };
        int getHTTPResponse(String & response)
        {
         long startWaitForResponseAt = millis();
         while(client->available() == 0 && millis() -
startWaitForResponseAt < TIMEOUT_MS_SERVERRESPONSE)
         {
             delay(100);
         }
         if(client->available() == 0)
         {
```

```
                        return ERR_TIMEOUT; // Didn't get server response
in time
        }
                if(!client->find(const_cast<char *>("HTTP/1.1")))
                {
                        #ifdef PRINT_HTTP
                                Serial.println("ERROR: Didn't find
HTTP/1.1");
                #endif
                        return ERR_BAD_RESPONSE; // Couldn't parse
response (didn't find HTTP/1.1)
                }
                int status = client->parseInt();
                #ifdef PRINT_HTTP
                        Serial.print("Got Status of
");Serial.println(status);
                #endif
                if(status != OK_SUCCESS)
                {
                        return status;
                }
                if(!client->find(const_cast<char *>("\r\n")))
                {
                        #ifdef PRINT_HTTP
                        Serial.println("ERROR: Didn't find end of status
line");
                        #endif
                        return ERR_BAD_RESPONSE;
                }
                #ifdef PRINT_HTTP
                Serial.println("Found end of status line");
                #endif
                if(!client->find(const_cast<char *>("\n\r\n")))
                {
                        #ifdef PRINT_HTTP
                                Serial.println("ERROR: Didn't find end of
header");
                        #endif
                        return ERR_BAD_RESPONSE;
                }
                #ifdef PRINT_HTTP
                        Serial.println("Found end of header");
                #endif
                String tempString = client->readString();
                response = tempString;
                #ifdef PRINT_HTTP
                Serial.print("Response:
\"");Serial.print(response);Serial.println("\"");
                #endif
                return status;
        };
        int convertFloatToChar(float value, char *valueString)
        {
                // Supported range is -999999000000 to 999999000000
                if(0 == isinf(value) && (value > 999999000000 || value <
-999999000000))
                {
                        // Out of range
                        return ERR_OUT_OF_RANGE;
                }
                // assume that 5 places right of decimal should be
sufficient for most applications
         #if defined(ARDUINO_ARCH_SAMD) || defined(ARDUINO_ARCH_SAM)
                  sprintf(valueString, "%.5f", value);
                #else
                  dtostrf(value,1,5, valueString);
         #endif
                return OK_SUCCESS;
        };
        float convertStringToFloat(String value)
        {
                // There's a bug in the AVR function strtod that it
doesn't decode -INF correctly (it maps it to INF)
                float result = value.toFloat();

                if(1 == isinf(result) && *value.c_str() == '-')
                {
                        result = (float)-INFINITY;
                }
                return result;
        };
        void resetWriteFields()
        {
                for(size_t iField = 0; iField < 8; iField++)
                {
                        this->nextWriteField[iField] = "";
                }
                this->nextWriteLatitude = NAN;
                this->nextWriteLongitude = NAN;
                this->nextWriteElevation = NAN;
                this->nextWriteStatus = "";
                this->nextWriteTwitter = "";
                this->nextWriteTweet = "";
                this->nextWriteCreatedAt = "";
        };
};
extern ThingSpeakClass ThingSpeak;
#endif //ThingSpeak_h
```