# Beehive Health Monitoring with .NET Gadgeteer


## Abstract

An extensible system to enable beekeepers to more easily monitor the health of their beehives has been developed, comprising software for an embedded device to automate sensor readings, an API for collection and distribution of the data, and a front-end for live monitoring.


## 1 Introduction

### 1.1 Overview

Honeybee colonies are an incredibly important part of agriculture in the UK and elsewhere, as they pollinate most of the crops we eat and produce valuable honey. Unfortunately they are in decline, and beekeepers are facing difficulty keeping their beehives alive from one season to the next. Through automated, more frequent and less intrusive monitoring, this project aims to aid beekeepers in improving the survival chances of their bee colonies.

The physical monitoring will be done using 'Gadgeteer' hardware provided by Microsoft, the external client for this project. This hardware consists of a microcontroller with a range of connected components such as a Wi-Fi transmitter, SD card reader/writer, and a range of sensors, for example those for detecting temperature and light. The microcontroller, which hosts the .NET Micro Framework, will be programmed on top of the Gadgeteer platform in order to, amongst other functions, automate the monitoring and transmit sensor data to an internet service.

The second aspect of the project is the software to collect, display and visualise data coming from the monitoring device located in a beehive. Requirements from UK beekeepers will determine the nature and extent of this web-fronted application, though always ensuring extensibility is not compromised.

## 1.2 Aims and Goals

Beekeeping is a widely practiced hobby as well being involved in large amounts of research due to its great agricultural importance. The primary aim is therefore to develop a system that will be useful for other beekeepers wishing to implement and (if so experienced) further develop a beehive health monitoring system. The ideal outcome is to better the survival chance of bee colonies, and perhaps learn more about the state of UK-based beehives such that the community can come closer to understanding the cause of declining colony numbers.

Thus, the goals for the system being developed are:

- Demonstrate what useful beehive properties can be monitored with off-the-shelf programmable hardware.
- Demonstrate how beekeepers could be better informed about the health of their hives than is possible at present with traditional tools. This will be though the use of web services to connect the device's data to a modern web-based user interface.
- Produce easily extensible open-source software so that the beekeeping community can more easily progress with automated hive health monitoring.

## 1.3 Scope

In general, allowances need to be made for the fact that hive access will be difficult, so testing is mainly or entirely from laboratory simulations. Assumptions about the type and location of the beehive can therefore be freely made (e.g. near mains power and a Wi-Fi hotspot); this is also to limit time wasting from working with potentially unreliable hardware such as GSM and batteries.

- Primary hardware should be commercially available and Gadgeteer-compatible
- A web-facing application will be the single system front-end
- Reliability of data is a secondary hardware concern; in general, hardware sophistication will be limited by time and cost
- The full system will be designed to work with a single beehive.

### 1.4 Structure and strategy

The project is externally supervised by Steven Johnston from Microsoft. He has kindly provided all the Gadgeteer hardware and offered advice on areas to explore, as well as providing contacts within the beekeeping community to discuss requirements and test the prototypes and final systems. The bulk of the work, however, is unsupervised and self-directed through individual research of what is needed.

The stimulus for the project is based on a Gadgeteer 'hack weekend' Mr Johnston was involved with, designed for experienced developers to experiment with Gadgeteer [1]. This was in the same context as this project (beehive monitoring); full details are available in the reference.

Due to the rather experimental nature of the project, and the lack of external constraint on the direction to be taken, it was important to decide on a good development strategy. The technique decided upon was the evolutionary prototype model. By frequently refining prototypes and testing them on users and in the laboratory, valuable early feedback can be introduced to better inform the project direction and requirements for the final system [2]. This iterative method is ideal given the rather broad areas to be researched and developed, and given that the end-users may not initially know what they require.

### 1.5 Report outline

The next chapter will detail the necessary background theory and review existing research as well as any commercial or amateur software for beehive monitoring. Chapter three presents the requirements analysis, wherein the details of requirements capture and analysis are described. Following that is a discussion of the system design and implementation in chapter four, then by testing and an evaluation of the project in chapters five and six respectively.

# 2 Context, Background and Research

## 2.1 Beekeeping

### 2.1.1 Colonies

The European honeybee is in trouble. In the past decade, colonies have been dying out a rate of about 20-30% a year in the UK [3] (and even higher in the US [4]), with no known definite cause [5]. This is sometimes due to a phenomenon known as Colony Collapse Disorder (CCD) [4], which has sparked a great deal of research into what may be responsible. Theories include environmental changes, human pesticide use, and parasitic disease [5].

As well as the threats from CCD, colonies also face the persistent threat of death from other sources, such as unseasonable weather and disturbance from wildlife [6]. The cold, wet summer of 2012 is thought to be the greatest cause of the unusually large losses reported in the UK in 2012/13 [3].

Regardless of the precise cause of colony deaths, what may help improve survival rates is better monitoring of these colonies so researchers and beekeepers can be better informed about what is happening in their hives.

The honeybee has for millennia been of great importance to human agriculture, and today colonies are responsible for more than half of the pollination of UK crops, worth an estimated £200 million [7]. This huge impact makes CCD a great concern, and recently there have been attempts to start monitoring beehives more closely through use of automated systems to replace intrusive, laborious manual checks.

Colonies of domesticated honeybees are housed in man-made beehives, with around a quarter of a million existing in the UK alone [7]. Moreover, most apiaries (beehive 'farms') are run by amateurs – 67% of all hives in the EU [8]. These two facts mean that for effective, widespread monitoring, an inexpensive and easy-to-implement system is needed. Focused and sophisticated research is already underway at research laboratories, but the general market lacks a solution.

A typical beehive houses a colony of around 50,000 honeybees at its summer peak [6], headed by a single queen. Typically, the colony will lie mostly dormant for the

winter as the bees struggle to survive on supplies built up during the summer. In spring, activity resumes and the hive will begin raising new bees (the brood) and gathering honey. In early summer, as the population swells a swarming event may occur. This is how new colonies are formed, as the queen leaves the hive with a subset of her colony and tries to find a new nest.

### 2.1.2 Hive monitoring

Beekeeping mainly concerns monitoring the hive to ensure it is healthy so the benefits of crop pollination and the production of honey and wax can be realised. A secondary goal is to manage the swarming events by either moving the bees to a fresh hive when ready (rather than let them escape to possible death), or by preventing it all together. The monitoring currently done involves manual, intrusive checks – for disease, queen presence, food stores, and brood size amongst others [6]. It would be beneficial to the colony to reduce these checks as far as possible, and make it more convenient for the beekeepers themselves who often do not live alongside their hives.

It is very important to realise that beekeepers typically only check their hives on a weekly basis, and not at all in the winter [9]. Thus, if anything undesirable were to happen between checks, such as swarming or foreign species attack, the beekeeper would not be able to respond to the threat in time, and the colony would likely die. Automation and warning systems are therefore crucial.

### 2.1.3 Beehive health properties

#### 2.1.3.1 Temperature and Humidity

Temperature and humidity are important variables affecting colony survival, both for mature bees and for the raising of larvae in the spring and early summer. In the UK the problem is conditions that are too cold or wet; the colony thrives in a warm, dry environment, and can die when the temperature or humidity stray from a certain range [6][p12, p17]. In particular, it is the temperature differential between the hive core and the natural external environment that is the most important indicator of hive activity and health [10]. This differential approaches zero when a colony dies [11] so by observing this an intervention can be staged to discover and rectify the cause. Present techniques often cannot detect this approaching death, as it is

considered dangerous to open a hive in the winter so they tend to be left alone for many months in the cold season.

#### 2.1.3.2 Mass

The hive's mass is an indicator of honey storage and bee population. Although it is difficult to decouple this signal, along with variable contributions from debris, food and the brood [6], sharp changes are expected from bulk departure (swarming, a common problem for beekeepers), and long-term trends are expected to indicate honey storage [12]. At present, honey quantities are estimated by the practice of 'hefting', whereby a part of the hive is physically lifted from the base and heaved up and down to gather an estimate. Given the shortcomings of this cumbersome technique, and the potential to discover other useful quantities and events from trends, automated electronic mass measurement would evidently be helpful.

#### 2.1.3.3 Disturbance and vandalism

Hives can be knocked over by common wild animals such as moles and badgers, and human vandals have been known to disrupt innocent beehives [6]. By detecting these events early enough for the beekeeper to respond, the colony might be saved.

#### 2.1.3.4 General activity

The entrance to a beehive is typically a thin strip that permits only a few bees at a time. Consequently, the area around the entrance can be a good indicator of the activity of the foragers in the colony, as bees will tend to cluster there in large numbers when activity is high [9]. Additionally, the entrance is where foreign invaders such as wasps will stage an attack [6]. It is therefore desirable to be able to visually monitor the events in this area, using a camera.

#### 2.1.3.5 Other

After extensive research on the possible properties that could be measured, the above are only the ones deemed feasible given the timeframe and project scope. Detection of foreign species (invaders and parasitic mites) through image analysis is one example of a useful property rejected on these grounds. Sound detection with a microphone, which has a multitude of potential uses including queen, swarming and disease-detection [13], was thoroughly researched and partially implemented, but became too complex to proceed.

However, the system design should be such that integrating further sensors as they become available should be easier to achieve.

## 2.2 Technologies

### 2.2.1 Gadgeteer platform

Microsoft's Gadgeteer platform is "an open-source toolkit for prototyping small electronic devices" [14]. A Gadgeteer device consists of modules that plug into a microcontroller (mainboard) running on the .NET Micro Framework (NETMF) [15]. NETMF is essentially a subset of Microsoft's powerful .NET framework that can run on devices with limited memory and processing power, with some additional features specifically targeted to embedded applications. Numerous companies have used Gadgeteer's SDK and hardware specification to produce commercially available mainboards with a growing number of modules, each of which has its own SDK to allow end-users to program the device. All programs on NETMF are written in C# and developed in the Visual Studio IDE.

Gadgeteer's plug-based, solderless design provides an abstraction of the underlying electronics, which makes it quick and simple to set-up and deploy working prototypes, giving it an advantage over similar platforms such as the Arduino and Raspberry Pi. The downside is higher cost and the limited availability of off-the-shelf modules, though it is possible to expose the inner mainboard connections and thus make a custom module.

For beehive monitoring, the useful Gadgeteer modules are mainly passive sensors like thermometers, with other utility modules needed for operations such as internet connection.

### 2.2.2 Web API

In early prototypes of the system, an Internet of Things web service called Xively [16] was used. This provides the tools to connect an embedded device to the internet through a private API, and subsequently retrieve any data pushed from the device. This service worked well for extracting current data from a fixed number of sensors. It has a number of major drawbacks, however, which ultimately meant that a replacement had to be found. The principal problems are the limitation and difficulty

in retrieval of historical data, and the less-than-desirable flexibility for changing what sensors are used. Additionally, a number of other web services were required, so it made sense to combine them all into a single custom platform.

To this effect, it was decided that a custom API would be built with Node.js [17], a server-side software platform ideal for building networking applications such as RESTful APIs [18]. Writing a Node application is done in JavaScript, meaning seamless integration with any client-side web applications, particularly in terms of passing JSON data to and from the API. The concurrency model uses an asynchronous, single threaded approach, in contrast with the conventional (e.g. Apache) multi-threaded technique. When IO is required, Node continues running the thread and sets up an event to trigger a callback to execute when the IO is finished. The multithreaded technique of halting the thread doing IO and switching to other threads is the norm for web servers. The latter is better for algorithmically intense applications such as multimedia processing, but Node performs better for lightweight RESTful services [18]. It is for all these reasons that Node.js was selected to perform all the server-side tasks.

The other component needed for the API is a place for the device data. Microsoft's Windows Azure Storage (WAS) was chosen to fulfil this role because of its flexibility and scalability [19], important given that vast amounts of data can be generated by a continuously running monitoring device. Specifically, the 'Tables' aspect of WAS offers structured storage of key-value pairs in a NoSQL database; this is where the data points will be stored.

### 2.2.3 Web front-end

The system's front-end Graphical User Interface will be built using the standard web technology stack of HTML5, CSS3, and JavaScript. Additionally, to speed the development of UI components, the jQuery JavaScript library [20] and Twitter Bootstrap CSS framework [21] will be used; this is expected to be a valuable way to save time in building a modern, responsive GUI. For similar reasons, a graphing library called flot will be used to power any charts; this is a popular, free, and well-maintained JavaScript library [22].

### 2.2.4 Version control

GitHub was chosen to host the source code and documentation. This is a modern version control platform using Git that has achieved widespread acceptance and integration with other services (see Other Tools below), making it ideal for use over the alternatives. Codeplex (running the Apache Subversion system) was trialled in an early prototype, but its poor integration with other services meant it was eventually disfavoured.

### 2.2.5 Other tools

All the server-side tools mentioned above could be hosted on a local machine. However, in order to improve the visibility of the project so that beekeepers can provide feedback during the project lifecycle, it would be beneficial to host the entire project on the World Wide Web. Windows Azure's Web Site service is ideal given its support for Node.js, and the useful feature of automatic deployment from GitHub each time the linked repository is updated. Local development and testing, as well as command-line access to real time server logs, can be obtained using the Azure SDK. The current project is running on Azure with a free student trial license.

## 2.3 Existing work

A number of research efforts have been conducted in the field of beehive health monitoring, typically involving custom-engineered (i.e. not off-the-shelf) hardware with a view to investigating a specific monitoring property. Notable examples include: using accelerometers to detect swarming [13], swarm-detection with thermometers and microphones [23], and detecting signs of colony development by monitoring temperature differentials [10]. All these efforts lack the crucial internet connection and front-end aspect of this project's intention that will enable future progression in widespread monitoring, but have provided the author with useful ideas for what hive properties to monitor (see §2.1.x).

The HOBOS teaching project is a sophisticated and complete all-round monitoring system with a live video web stream of the hive entrance, and live graphs of all the sensors [24]. However, it is intended as a one-off educational tool, costs many thousands of pounds, and is closed-source. These facts make HOBOS unsuitable for a widely implemented hive monitoring system.

A few commercial integrated monitoring systems exist too, e.g. Arnia [25], BeeWise [26] and Swienty [27]. Due to the proprietary nature of the software and hardware involved, these systems are not considered competitors, as a main goal of this project is to provide a platform for further development. Nevertheless, many of these have achieved an internet-connected system with sophisticated sensors and informative front-ends, offering a sobering perspective on the purpose of the present project.

Various hobbyist attempts at a few of the sensors, some with internet connection and basic front-end, can also be found in the literature, e.g. HiveTool [28], Honey Bee Counter [29], Beehive Scale [30], Bee Hive Monitor [31]. However, no integrated, extensible, internet-connected, multi-sensor solution can be found, and the few web interfaces that do exist leave much to be desired in functionality and modernity (see appendix X. for full review).

It is therefore evident that the work to be undertaken is sufficiently novel to justify its implementation, and it is hoped that it will become a standard for future beehive health monitoring projects.

# 3 Requirements Analysis

## 3.1 Problem statement

The honeybee is the most researched non-human organism [6], in part due to its economic significance. Despite this, the millions of bee colonies around the world face the serious threat of death every season. It is evident that the reasons are complex and not entirely known, but what is certain is that inadequate automated monitoring is currently done of these beehives, yet this can save colonies from death. The research efforts involved in monitoring are inaccessible to the ordinary beekeeper, yet it is they are who manage most of the beehives. The small number of commercial solutions lack expansibility, and the few hobbyist solutions are inadequate in their coverage.

Beekeepers lack a flexible system for automated monitoring; one that can be further developed, is well-documented, easily implemented, and has a useful front-end that can aid them in increasing the survival chance of their bee colonies.

## 3.2 Terminology

The physical hardware to be placed inside the beehive shall be referred to by the component names (microcontroller, sensor etc.), or 'MCU' to refer to the whole. The program running on the microcontroller shall be referred to as the 'Device', an abbreviation of 'hive health monitoring device'.

The 'User' is the target market for deployment and day-to-day use of the system – UK beekeepers.

'API' refers to the server-side software for data input, retrieval and manipulation. 'Application' alludes to the front-end user-facing application and logic.

## 3.3 Requirements

### 3.3.1 Monitoring device

Requirements for the monitoring device to be placed in beehives were gathered from a number of sources:

- Discussions with beekeepers, both hobbyists from the software development community and professionals from the British Beekeepers Association (§A3)
- Research of existing works from the available literature (§2.3)
- Results from prototyping and testing in laboratory conditions
- Limitations provided by the project scope (namely time and cost).

Priorities follow a 1-3 scale, with 1 being top-priority.

| ID | Description | Priority |
|---|---|---|
| RH01 | The MCU shall be capable of measuring the following basic properties of the beehive: internal temperature, external temperature, internal humidity, light level, and movement (Boolean). | 1 |
| RH02 | The MCU shall be capable of measuring the overall mass of the beehive. | 2 |
| RH03 | The MCU shall be capable of capturing still images of the hive entrance. | 2 |
| RH04 | The MCU shall have the ability to connect to wireless internet. | 1 |

| | | |
|---|---|---|
| RH05 | The MCU shall be able to read and write data from local storage. | 1 |

Table 3.1: Hardware Requirements for the monitoring device. Priorities are generally influenced by availabilty, cost, complexity and time limitations, whilst some are essential for the rest of the project to run succesfully.

| ID | Description | Priority |
|---|---|---|
| RS01 | The Device shall transmit live Sensor data, including any binary data (such as images), to the Data API through HTTP. | 1 |
| RS02 | The Device shall additionally commit all numeric (non-binary) Sensor data to local storage, in csv format for easy reading. | 2 |
| RS03 | The Device shall buffer data to local storage when no internet connection is available, writing the full buffer on resumption. | 2 |
| RS04 | The Device shall load all necessary configuration settings from local storage in XML format, so the device does not have to be reprogrammed. | 1 |
| RS05 | The Device shall read Sensor data and take pictures at regular time intervals, specifiable in the configuration file. | 1 |
| RS06 | The Device shall be able to operate with any Wi-Fi network connection through specification in the configuration file. | 1 |
| RS07 | The Device shall commit to the Data API endpoint specified in the configuration file. | 3 |

Table 3.2: Software Requirements for the monitoring device. The emphasis is on configurability and flexibility, reducing the need to reprogram the device if the hive is relocated or front-end requirements change.

### 3.3.2 Front-end application

These were gathered primarily from the beekeepers contacted about the monitoring device, but also by consideration of flexibility to future hardware availability. Refinement of the requirements was done iteratively as front-end prototypes were developed and tested on the same beekeepers initially contacted.

| ID | Description | Priority |
|---|---|---|
| RU01 | The User shall be able to view all live Sensor data, including any binary data (e.g. pictures) in a prominent ‘dashboard’ display. | 1 |
| RU02 | The User shall be able to view recent (~3hr) trends of the numeric Sensors in colourful graphical format. | 1 |

| | | |
|---|---|---|
| RU03 | The User shall be able to view an alarm for each Sensor indicating whether it has breached a threshold. | 1 |
| RU04 | The User shall be able to observe and manipulate graphs of longer-term Sensor trends, configurable on the period, period length, and Sensors to display. | 1 |
| RU05 | The User shall be able to view tabular historical data based on a specified date range; this shall be exportable to CSV. | 1 |
| RU06 | The User shall be able to view reports on simple statistical analysis of the collected data such as monthly min/max/mean. | 3 |
| RU07 | The User shall be able to view a local weather report from a nearby location, fulfilled using a free weather API. | 3 |
| RU08 | The Application shall load dynamically based on settings obtained through the API – in particular, the Sensors and alarms to show and their customisations, plus the Device-dependent update rate and its location. | 1 |
| RU09 | The User shall be able to modify these settings in the GUI itself. | 2 |
| RU10 | The Application shall update any live data automatically as soon as new data is expected. | 1 |
| RU11 | The Application shall connect to data solely through asynchronous requests to the Data API. | 1 |
| RU12 | The Application shall use responsive design to be suitable for use on desktop and mobile devices, and run in a standard modern browser. | 2 |

Table 3.3: Front-end User Application requirements. Priorities are mostly based on aggregated responses from amateur beekeepers (the key user).

### 3.3.3 Data API and Services

These requirements are from direct analysis of how the needs for the other straddling system components (Device and front-end Application) can be met.

| ID | Description | Priority |
|---|---|---|
| RA01 | The API shall be capable of receiving and storing data points from the Device. | 1 |

| | | |
|---|---|---|
| RA02 | The API shall store data in a way that is independent of what and how many Sensors are used. | 1 |
| RA03 | The API shall expose methods for querying the stored data based on date range, most recent periods, and latest data point. | 1 |
| RA04 | The API shall be capable of receiving and storing binary image data. | 1 |
| RA05 | The API shall expose a method to retrieve the current stored image. | 1 |
| RA06 | The API shall expose methods for reading and writing Application settings. | 1 |
| RA07 | The API shall be able to return data in JSON, XML, and CSV format. | 2 |
| RA08 | The API shall require a password for all requests for data input. | 2 |
| RA09 | The API shall be scalable to store many years' worth of data points. | 2 |
| RA10 | The API shall use RESTful design principles for maximal flexibility and robustness. | 2 |
| RA11 | The Services shall automatically send alerts of alarm threshold breaches to a configurable email at a configurable rate. | 1 |

Table 3.4: API requirements based on the needs of other system components, and good software design practices. In addition, requirements to fulfil some non-UI-based server-side logic tasks.

### 3.4 Domain Model

From analysis of the requirements, it is straightforward to identify the key actors and their interaction with the various system components.

[PIC: Domain model; Caption: Core entities are highlighted; some sub-entities have been combined for clarity]

### 3.5 Use Cases

There is little human interaction from the User during run time (though potentially a reasonable amount of setup), but the MCU itself is also an important actor as the external hardware running the software of the monitoring device (hardware is indeed an actor []). The core written use cases are provided in §A1.

### 3.5.1 MCU/Time

Core use cases for the MCU 'interacting' with the monitoring device. These include a few initiated by the MCU on booting, but mainly the operations are time-based triggers.

[PIC: Use Case diagram - MCU]

### 3.5.2 User

An outline of some of the key use cases for the beekeeper interacting with the front-end. Largely the user interaction merely involves viewing various data in very intuitive and obvious ways, so any such use cases have been ignored.

[PIC: use case diagram - user]

## 3.6 Work Packages

The key stages in the development are expected to be as follows, and are based on early mini-prototypes and testing of the various platforms and technologies, as well as experimenting with the Gadgeteer sensors and analysing the above requirements.

1. Core system prototype – initial Gadgeteer sensors, with their data pumped to the Xively API, and displayed on the native Xively interface.
2. Full stack prototype – own JavaScript front-end, API and database back-end using Azure with Node.js, plus a refined monitoring device that implements its full requirement set.
3. Gadgeteer simulator (Windows command line application) - for testing the ability to use new sensors; this negates the need to build new sensors, which proved difficult.
4. Flexible system that is customisable on the sensors (in both the API and User Application), and on any other elements such as the alarms.

# 4 Design and Implementation

## 4.1 System architecture

The overall system is designed consist of three core independent modules, ideally independent from one another. These are:

1. Monitoring Device – software running on the physical Gadgeteer hardware with two core functions
    a. Gather data from the sensors
    b. Transmit all available data to the Data API.
2. Cloud Services – Node.js software running on the Windows Azure platform with a number of core functions
    a. API for retrieval and storage of any incoming data
    b. API for querying and outputting available data (internal and external)
    c. Web server to deliver content
    d. Service for communicating data-driven alerts.
3. Front-end – web application for users with the central operation being
    a. Customisable access to the Data API to visually display live data and trends and set/retrieve settings.

Module 2 is really two sub-modules operating in the same environment; they too are in fact operationally independent. The Data API carries out tasks 2a and 2b, leaving the Web Services to perform tasks 2c and 2d.

[PIC: architecture simple (UML)]

The key point is that any individual module can be replaced by a similarly-operating one to leave the full system running in the same way as before. For example, someone may wish to implement the system on a different platform, perhaps a traditional ASP.NET plus SQL Server implementation, which would require a rewrite of the Data API and Web Services module but no other components need be affected. Similarly, the Gadgeteer module could quite easily be replaced by a similar piece of embedded hardware like the Arduino running a program to send data from a series of sensors. Finally, the front-end is very replaceable due to the designed flexibility of the Data API.

[PIC: architecture detail (graphical)]

## 4.2 System components

### 4.2.1 Monitoring Device

#### 4.2.1.1 Hardware configuration

The Gadgeteer hardware was assembled with the following off-the-shelf modules (provided by Microsoft Research Cambridge but commercially available):

1. Microcontroller - 14 socket FEZ Spider mainboard
2. Power supply - dual support for USB and DC
3. SD card module
4. Wi-Fi - RS21 version
5. Sensors
    a. Light
    b. Temperature / Humidity (combined)
    c. Temperature / Pressure (combined, 'Barometer' in the catalogue)
    d. Accelerometer
    e. Camera - 320 x 240 resolution

In addition, an ordinary SD card (8GB SanDisk SDHC) is used in the SD module for permanent storage, and the mainboard is powered by a commercial 5V USB power pack. It is likely that combinations of similar modules would work just as well; these are merely the most useful ones available for this project.

[PIC: Device-VF. Caption: Fully assembled device. Assembly takes under one hour thanks to the simple modular socket-based architecture. The modules are mounted on a Tamiya prototyping plate (16 x 12 cm)]

Based on the requirements, as informed by the research of what is needed to monitor the health of a beehive, it was decided that the following properties were to be derived from these sensors, in the first instance:

- **Temperature**, inside
- **Temperature**, outside
- **Temperature differential**, in minus out
- **Humidity**, inside

- **Light** level
- **Motion**, as a Boolean (moving or not), indicating external disturbance
- **Image**, at the beehive entrance.

The six core numeric properties are the variables that form a single data point every time a measurement is requested. The still image is requested simultaneously, but is not bundled with the numeric data.

The variable missing from this list but of significant importance from the requirements is beehive mass. No Gadgeteer sensor exists for this purpose, so during the second round of prototyping (work package two) an attempt was made to build such a mass sensor for Gadgeteer. This was ultimately unsuccessful (see §AX), but inspired the idea of making a Gadgeteer simulator to produce semi-random data for any arbitrary numeric sensor (see §X.X). This allows testing of the system to ensure it is able to support new sensors when they become available (see testing sec.).

#### 4.2.1.2 Programming the Device

Having built the device, it is then programmed in Visual Studio with the Gadgeteer and NETMF SDKs. Debugging and deployment is through a USB connection.

[PIC: GadgeteerDesign_V2min.png. Caption: Visual Studio's Gadgeteer Design View. The modules in use are selected and 'connected' to the mainboard socket to which each is connected. Doing so exposes each module's SDK so it can be used when writing the program.]

#### 4.2.1.3 Program design

The Gadgeteer platform differs from traditional embedded device programming, which mainly focuses on compact, relatively low-level C code. It uses a very high-level, object-oriented language, C#. This offers advantages in speeding up development time and allows use of good design practices. The key design pattern this allows is the event-driven model, which is used by many of the Gadgeteer modules [s10]. Rather than running a `while (true)` loop and polling sensors, buttons etc., an event handler is setup, with its callback method fired asynchronously when the hardware responds.

[CODE: e.g. of setting up an event handler for a sensor]

Multiple handlers are queued on the Gadgeteer event dispatcher; the dispatcher then sends events back to the main program thread for execution, with multi-threading handled behind the scenes. Because of this design, the ideal way to perform regular operations is using the native `Gadgeteer.Timer` class, which allows a method to be fired at regular intervals.

[CODE: setting up timer]

The main use of this is to periodically retrieve data from the sensors, which is then transmitted to the RESTful web API over HTTP. In fact, almost all activity after the initial device boot is governed by this one Timer. The initial boot itself sets up all the module handlers and initialises their state (connecting to Wi-Fi, loading settings from the SD card, creating directory structures on the file system, synchronising the system clock etc.), and finally starts the Timer to schedule periodic sensor readings.

[PIC: Activity diagram for one update cycle (get data, save, push, errors etc.)]

Despite the advantages of the Gadgeteer software platform, it cannot get around the limitations of embedded devices. Their small memory means the .NET MF libraries are much reduced from .NET's offering; for example there are no key-value pair based data structures on NETMF, and string operations are much more limited. This fact necessitated implementing a few methods/classes normally taken for granted, such as `string.join()`, and line-by-line file reading.

The overall class structure reflects the delegation of activity from the main program to individual functional modules, both for internal purposes (`SdHandler`, `WifiHandler`), and external communication (`APIconnector`). A number of utility classes provide resources missing from the core libraries, whilst the static Config class packages easily-configurable settings into one location.

[PIC: Device class diagram]

Device flexibility is obtained through two means: good program design, and use of the SD card to load numerous settings from an XML configuration file. Due to the differences between Gadgeteer's implementation of sensor modules, and that fact that multiple variables ('channels') can be derived from a single module (The "temperatureHumidity" module reads both temperature and humidity), the approach

taken was to focus on the variables themselves. The `Channel` class abstracts the details of a variable away from its parent module, so the Device can be modelled as a serious of sensor modules giving rise to a set of distinct channels.

[CODE: channel class]

The `SensorsHandler` class sets up the sensor modules and their event handlers, and lays out which Channels to measure.

[CODE: `SensorHandler` class – fields only]

It is thus straightforward to start measuring a new variable, especially so from an existing module, but also by implementing a new sensor module. No other classes need modifying, and all channels will flow to the API.

Whilst it is essential to change the program code when new sensors are added, the same is not true for many other variables. The XML configuration file offers an easy way to change the Device settings without having to reprogram the device. This is particularly important for non-technically-minded users, whom may be given a pre-built and pre-programmed device but have a need to change the settings.

[CODE: config.xml]

Most usefully, the network settings can be changed so the Device can work on different wireless networks through simple changing of the configuration file. Similarly, the update frequency can be reduced to minimise power requirements.

### 4.2.2 Web API

#### 4.2.2.1 RESTful design

By confirming to RESTful architectural constraints, the data API conforms to the important properties of portability, scalability, flexibility and robustness [32], which are desirable in any system for data distribution. The core constraints followed, as identified by the author of REST [33], were achieved as follows:

Client-Server. Data returned from the API does not conform to the needs of any particular user interface, so it can be freely used by any number. Although a specific

user interface was developed for this project, the goals are clear that we should not be tied to its implementation.

[CODE: sample JSON GET response]

Stateless. Individual requests are self-contained; the server does not keep clients' state between requests.

Cache. All requests have an appropriate caching policy to minimise resources by preventing excess requests; this improves general scalability and client performance, whilst reducing the chance of server failures due to overload. Specifically, static content requests are labelled with a long-lifespan cache, whilst sensor data is given a short cache roughly equal to the update frequency, and PUT requests are never cached. These are all set using the HTTP `Cache-Control` header

[CODE: cache labelling for static content]

Code-On-Demand. Although optional in REST, this was implemented to enhance client functionality. The JSON settings object can be manipulated through the API, or set on the server. These settings are specific to a user interface, but any number of arbitrary settings can be defined, modified, even deleted.

Uniform Interface. This is the most important feature of REST [33], and the key part is identifying which resources are available on the server, and specifying how these can be manipulated by clients. This is implemented through proper use of HTTP methods, caching, URIs, and internet media types. The resources identified that make up the API were:

- Data point feed (numeric sensors)
- Image feed (camera sensor)
- Settings
- External (accessing other APIs, e.g. weather).

Each resource has its own URL and supported methods and internet media types. For example, PUT and GET methods determine respectively whether data are being saved (from Device to API), or retrieved (web application) for the image and data point feeds, whilst the External resource only allows GET.

#### 4.2.2.2 Overall Design

The key goal of the design for the overall web API and services was achieving logical separation of the API resources, the database implementation, the web services, and any web server logic. These are the distinct operations of the Node.js-implemented software, so the components need to be decoupled should any of their implementation requirements change. For example, the API relies heavily on a database, but the particular one chosen should be replaceable without affecting the API itself.

This decoupling has been achieved through effective modularisation. In Node, modules form the core of the system design and effectively act as classes. Apart from a few core functional modules, each module (both from the Node library and those that are imported or created) is isolated from every other until explicitly imported. Once imported, only members (fields and methods) defined as exportable (i.e. 'public') are available, allowing encapsulation to be achieved since other members remain hidden to the importing module.

[CODE: Node.js encapsulation example]

On top of this, a layered approach has been adopted to further constrain the system design into proper separation of operations. This should improve the ability to maintain and extend the system's operations, for example by adding more API resources. The layers are implemented as simple directories, but used such that modules ('classes') in one layer ('package') may only call classes in a higher layer.

[PIC: Node module structure]

#### 4.2.2.3 Data model

The Azure Table Service (ATS) is a highly scalable distributed No-SQL database platform. To maximise the advantages it offers whilst maintaining flexibility in querying, it is necessary to design the storage model effectively. Only one table is needed, which stores all data points from the Device, as received through the API data point feed resource. Each row corresponds to a single data point, comprising the dateTime of the recording and values for each measurement channel taken.

[PIC: ATS data model]

After experimenting with a relational-style model of one row per channel per data point, with a channelID foreign key to a separate channel table, this approach was favoured for a number of important reasons:

- No need to maintain a separate list of channels in another table, which impacts detrimentally on flexibility when new sensors are produced offering new measurement channels
- More compact storage
- Faster retrieval of data points
- More data points returnable (ATS has a 1000 per query limit).

ATS uses a concept of a 'partitionKey', a required field for each row which effectively indexes the table. For time series data, this feature of ATS can be exploited by putting the dateTime field here, achieving two useful results:

Firstly, rows can be returned in a specific order, something not normally possible with ATS, which does not have an 'order by' concept for fields of a table, instead always returning rows ordered by partitionKey. This is used to return the most recent data points, a very useful feature for the API. Secondly, ATS will automatically group rows by similar partitionKey, as its distributed storage model uses the partitionKey to group similar data for faster retrieval [34]. Consequently, data points for similar dates are likely to be partitioned together, making queries for specific date ranges (another feature of the API) faster.

Other data (settings, images) are stored internally on the server file system, since the small quantity and simplicity of such data do not justify use of more sophisticated storage techniques. In fact, the settings are stored as JSON, which offers the advantage of impedance matching with Node.js; JSON is a natural and effective storage format when working with JavaScript since it can be manipulated with native syntax and methods. Moreover, hierarchical data structures can be created and manipulated very easily compared to flat representations such as database tables.

[CODE: sample JSON from the settings]

### 4.2.3 Web application

Following best practices [35], all behavioural code (JavaScript) is separate from the visual content (HTML), which in turn is separate from its styling (CSS). This means adding event handlers (`onLoad`, `onClick` etc.), after the DOM has loaded, which is also when all dynamic content, as obtained asynchronously from the API, is added.

#### 4.2.3.1 Dynamic loading

Using the Settings resource of the API (see above) for storage, many of the GUI components are loaded dynamically at start-up, and can be modified at runtime by the user. Principally, the dynamic components are the alarms and the sensor blocks, the latter being particularly important for achieving flexibility to a change in the sensors used by the Device. Essentially, this means changing a few lines in the settings file (either through a form on the web application, or by hand) will achieve corresponding changes in the GUI to match. The settings themselves are encoded in JSON for optimal ease of manipulation with JavaScript and the API; even complex structures such as the sensors are easy to manage.

[CODE: JSON settings sample]

#### 4.2.3.2 Single-page application

The application is designed as a single-page application (SPA) to maximise user experience by providing a very fluid approach to the loading of resources.

The SPA has been achieved through the 'thick-client' architecture; heavy use of event handlers and AJAX communication, both made easy with the jQuery library. The initial page request loads all static content (CSS, JavaScript source), with all subsequent HTTP requests made asynchronously through AJAX requests to the Node.js server, which responds with JSON data from the API. This in turn is manipulated by the client browser running the JavaScript, using the JSON to inform the modification of the existing DOM, by adding or changing existing HTML.

[CODE: AJAX call, plus some DOM manipulation – simplified]

#### 4.2.3.3 Class design

An MVC architectural design was adopted, as is natural and very common for web applications []. Since most of the View is defined by the HTML page, the JavaScript

classes are contained on just two layers, a View-Controller (VC) for interacting with the DOM and triggering updates, and a Model for communicating with the API and providing content for the VC. As required, the Model cannot use or call methods of the VC, but simply exposes methods for it to use to update the DOM as required by user interaction or automated data updates.

[PIC: class diagram]

Classical class-based object-oriented programming (OOP) does not come naturally to JavaScript development, as evidenced by the native language's absence of explicit class declarations, access modifiers, and namespace support, for example [35]. However, all of these features can be implemented in JavaScript's prototype-style OOP to provide a more maintainable and comprehensible code design.

Namespaces. Classes for the View-Controller and Model are logically separated so have been put in separate namespaces. A namespace is achieved by creating an empty global-scope object with the variable name of the namespace, then subsequently attaching classes to that object.

[CODE: Model namespace declaration]

Classes. Everything in JavaScript is really an object, but class-like behaviour can be achieved in numerous ways, all variations on declaring a `function` 'object' and attaching class members to it, with any arguments of the `function` comprising the constructor parameters. Static classes, whereby all members can be accessed without first creating an object and calling the constructor, are achieved by adding the `new` keyword to the class declaration, which effectively calls the constructor immediately. This latter 'Singleton'-style approach was rather heavily adopted.

[CODE: classes]

Access modifiers. Restricting access to class members is an important OOP concept for its encapsulation effect, and is achieved in JavaScript through careful scoping of variables. There are in fact multiple ways to achieve this, but the approach adopted here was to use the `this` keyword to attach public members to the class 'object' , and the `var` or `function` keyword to maintain private access for fields and methods respectively.

[CODE: public vs. private]

Using these hacks, an application made with some classical OOP principles in mind was developed, though perhaps using a JS framework would have helped achieve better design (see Eval).

# 5 Testing

Broadly, it was desirable to test the key points in the design and ensure that the software is robust and extensible in the way specified in the requirements. The nature of the development phase, involving prototyping and oft-changing requirements, meant than unit testing was not suitable. A higher-level approach based on system and integration testing was therefore followed.

## 5.1 Adding new sensors

Because a key feature of the software is its flexibility to the addition of new sensors, this is a key area of testing across all components. Due to the aforementioned failure to produce a physical load sensor to add to the Device itself, such a sensor was simulated. In fact, a short .NET command-line program was written in C# to simulate the API-connection of the Gadgeteer Device. In this way, data for new sensor channels were sent to the API, allowing its extensibility to be tested, as well as its compatibility with altogether different devices. With new sensor channels available in the API, the front-end's adaptability is then tested by updating the JSON settings to include a new measurement channel.

[CODE: new simulated sensor channel]

[PIC: JSON current_data showing new sensor]

As evidenced, the API responds successfully to the new channel.

[PIC: JSON web showing adding new sensor]

[PIC: GUI Dashboard showing inserted sensor and alarms]

The front-end accommodates the new sensor channel successfully, so the test passes.

## 5.2 Device-API communication

Without data, the API and front-end serve little purpose, so ensuring that data reaches the API from the Device is critical. A series of performance tests were therefore performed to reveal the limitations of this connection, and uncover problems in edge cases.

### 5.2.1 Maximum frequency

It is expected that performance on the Device end will limit how frequently data points can be sent to the API. In this stress test, the Device update frequency was varied to find the maximum frequency possible. The testing revealed a number of bugs in the software that only appear at high frequency, due to overlapping execution of event handlers as they are fired from the dispatch queue. A number of these were fixed (for instance, over synchronising of the clock leading to incorrect system times). However, at frequencies below five seconds, the overlapping is excessive and the dispatch queue eventually fills up and results in the device crashing with an out-of-memory exception. At five seconds, the Device is running almost continually (very little idle time), but the success rate of data points reaching the API is close to 100%. Five seconds is therefore deemed the maximum possible frequency in the current setup, though this is likely to differ under different network conditions and with more/fewer sensors.

To ensure the API was not at fault, the aforementioned CLI simulator was used on a desktop machine to transmit at one second intervals. The Device simulator gave a 100% success rate at this frequency.

| Frequency / s | Success rate / % | Comments |
|---|---|---|
| 1 | 28 | Device crashed before end |
| 3 | 64 | Device crashed before end |
| 5 | 98 | Very little idle time |
| 10 | 100 | No problems |
| 30 | 100 | No problems |

Table 5.1. Success rate for Device-API data point transmission at different frequencies. The test period was 50 data points. Success rate is calculated as no. points saved by API / no. points expected (50). For the three and one second

frequencies, even before the device crashed the rate of transmission was less than expected.

### 5.2.2 Buffering

Many beehives are in remote areas where network availability may be poor. To account for this, buffering of data (when network drops) has been implemented as per the requirements. To test this, a button was added to the Device, which disables the network temporarily to allow the buffer to fill. These small test cases were successful, with all buffered data transmitted to the API without fault.

In practice, network loss may be for a period of up to a week (typical beekeeper visit frequency). By producing faked data to simulate a week of data capture (at standard 60s intervals, this is just over 10,000 data points), the long-term buffering capability was also tested. The simulated data was compiled to API-friendly format and sent to the API on a desktop machine to test the API's ability to receive such a large quantity of data points.

The result revealed a potential problem with the database. Using detailed logging it was discovered that it could only process approximately 12 data points per second, and would 'hang' every minute or so; the full process took almost 20 minutes and gave a success rate of only around 80% compared to the usual 100%. Moreover, during processing of the 10,000-strong batch the database could not process any other requests.

To mitigate this problem, a safe limit of 100 data points was decided-upon and successfully tested; this quantity only takes a few seconds to process so does not interfere with other requests. Therefore, on the Device itself, the buffer will only transmit 100 data points per update cycle, leaving the database time to process and preventing any memory issues with the Device having to send megabytes worth of data points at one instance. This change in strategy required modification of the Device software, which was performed and tested successfully on a buffer of 10,000 data points.

## 5.3 User-acceptance

The remote availability of willing beekeepers for testing the physical device in a real beehive was such that field testing was not feasible. Moreover, the project scope is

clear that this is not necessary as the conceptual demonstration of beehive health monitoring possibilities is more important. However, a core part of the project is the front-end, and this was robustly user- tested in an iterative manner throughout its development, in order that its features best match the needs of beekeepers. This testing was achieved by communicating with three beekeeper contacts provided by the client, Mr Johnston. Using the publically available URL for the front-end, hosted on an Azure Web Site, the beekeepers could provide useful feedback, which was used to inform and refine later prototypes.

# 6 Evaluation and Conclusions

## 6.1 Summary and conclusions

The software developed offers a good insight into what is likely to be a rich and much-developed field. Automated beehive health monitoring is part of the trend in the internet-of-things world, whereby ordinary objects are transformed into networked components producing data to inform and advise. To-date, little work has been done to connect beehives to the 'cloud' and produce visual front-ends for real-time viewing of hive data. The present work sought to address this by targeting the many thousands of beekeepers in the UK with a system that demonstrates how automated hive monitoring using commercially available hardware could aid them in looking-after their bee colonies.

Using simple but useful sensors such as thermometers and accelerometers, pushing this data to a cloud-hosted API for storage, and finally viewing this data on a modern user interface, the project goals have been achieved. Through specific design choices, the extensibility of the system – particularly to emerging sensors on the Gadgeteer platform – has been demonstrated. This should allow the present work to achieve greater practical use, either through implementation of a custom monitoring device or extension of the present one.

The present system is ready to be trialled, and offers the first known non-commercial, ready-to-implement solution to automated, cloud-connected beehive health monitoring. Furthermore, it offers many opportunities for further expansion. With

enormous potential benefits, it is possible to envisage its use in many thousands of beehives.

### 6.1.1 Financial aspects

The current cost of the fully assembled device is approximately £200 in the UK (as per distributor list pricings of each component). However, this high price is largely a result of the prototype-oriented nature of the Gadgeteer platform. The raw hardware (sensors, microprocessor) is mass-produced and inexpensive, so a production device could likely be manufactured for much less than the current prototype.

As mentioned in sec. 2, the net value of the UK's quarter of a million beehives exceeds £200m, largely due to pollination of agricultural crops. This equates to £800 per hive, making the saving of the life of a colony very economically valuable. Even at £200, the investment in the monitoring device could prove very worthwhile, especially given the high colony death rate that means one device could prevent multiple deaths in its lifetime.

## 6.2 Critical evaluation

### 6.2.1 Fulfilling requirements

Almost all requirements were fulfilled successfully. The notable exception was the implementation of a mass sensor for the Device, something desired by all beekeepers asked. Although much researched, the attempt failed. However, given the ability to simulate its behaviour on the API and front-end, this failure was not deemed particularly problematic, and it is expected that others with greater electronics experience will have little trouble producing a successful mass sensor.

### 6.2.2 Technologies

In general, the chosen technologies (after early prototyping) were satisfactory. In the stress testing, however it emerged that the database implementation – Azure's NoSQL Table Service (ATS) – gave unsatisfactory performance in certain cases. In addition, the range of operations is very limited, so doing any significant statistical analysis of the data is unlikely to be straightforward. For these reasons, an alternative implementation may be preferred. Due to good design of the API, which strongly separates the ATS logic, this should not be unduly difficult.

### 6.2.3 Design

Throughout, a principle of low-dependency was adopted, so it easy to add and modify features – of the front-end, API and device. This was achieved through decoupling of logic into appropriate classes, and subsequent layering of these classes. Moreover, the high-level architecture of decoupled API, front-end and Device makes it easier to change any single component without affecting the others. Although these aspects were successful, robust object-oriented principles were not always followed. This is particularly noticeable in the JavaScript code (API and front-end), where, for example, there was excessive use of static classes. It is therefore probable that low-level design of the API and front-end could be much improved, probably with the use of a JavaScript framework which makes it easier to follow OOP principles.

## 6.3 Further work

### 6.3.1 Device operations and practicalities

On a practical level, field testing would certainly be useful. This would require long-term deployment of the Device in an operational beehive – one year at least, to capture the complete bee colony lifecycle. The issues raised by such testing are expected to be more hardware-related, for example battery failure, Device crashes, and weathering of components.

In addition, the range of sensors would need to be improved for the Device to meet the full requirements of most beekeepers. As discussed in the background research, further properties to measure may include hive-weight and colony noise. The ease of adding numeric sensors such as the former has been demonstrated, but properties such as the latter (sound) require specialist interpretation and front-end delivery. Consequently, it may be desirable to improve the ease of adding these specialist binary-producing sensors to the system, and then implement the sound detection sensor to test this and ideally add value to the health monitoring process.

### 6.3.2 API and front-end

The front-end, whilst modern and attractive with an informative dashboard for live data, is somewhat light on features for historical data. This is a potential area for significant development, as the current graphs and tables only offer a snapshot. Incorporation of more in-depth data and trend analysis is expected to be of particular

use to beekeepers. This could inform better warnings as looking for deviation from normal trends may signal certain unusual hive activity. Moreover, different hives could be compared, and this wealth of data may be of use to honeybee researchers looking to arrest the decline in colony deaths.

### 6.3.3 Multiple hives

In meteorological circles, numerous web-based platforms exist for weather observers to connect their weather stations to a central service. Examples include Weather Underground (WU – "the world's largest, with over 25,000 active stations") [36], the Met Office's Weather Observations Website (WOW) [37], and the Citizen Weather Observer Program (CWOP) [38]. These serve numerous purposes: comparison of weather data across the world [36] [37], meteorological research and forecasting [38], and provision of weather services such as web APIs [36] [38].

It is not too ambitious to envisage similar prospects for the beekeeping community. At one time, weather observers were akin to beekeepers in their use of traditional equipment to monitor their environment. The explosion of cheap electronic weather stations, and the availability of software for sending this data to the aforementioned web platforms, changed the weather community. Quite reasonably, devices such as the one in this project could eventually have a similar impact on beekeeping, especially in terms of mass availability of data for research.

Achieving this would be a project in itself, and may involve either redesign of the API to accommodate multiple incoming Devices, or a separate service dedicated to accumulating device data from multiple sources. In addition, a dedicated front-end for viewing aggregated data – in map form for example – could be developed. The former option is similar to what the Xively service does today, but has the drawback of keeping the data private; to reap the full benefits of the thousands of devices that could be deployed in hives across the country it is essential to keep all data openly available for analysis.

With these possibilities for data sharing in mind, the software stack presented in this report could be just the beginning of a process of vastly improving knowledge of honeybee colonies, to the invaluable benefit of beekeepers and the agriculture industry alike.

## Bibliography


[1] J. McLoughlin, ".NET Gadgeteer hack weekend," Developer South Coast, 27 April 2013. [Online]. Available: http://www.meetup.com/DeveloperSouthCoast/events/110389952/. [Accessed 22 Aug 2013].

[2] A. Davis, "Operational prototyping: a new development approach," *Software, IEEE,* vol. 9, no. 5, pp. 70-78, Sept. 1992.

[3] British Beekeepers Association, "BBKA release winter survival survey," 13 June 2013. [Online]. Available: http://www.bbka.org.uk/files/pressreleases/bbka_release_winter_survival_survey_13_june_2013_1371062171.pdf.

[4] J. Bromenshenk, "Colony collapse disorder (CCD) is alive and well," *Bee Culture,* p. 52, 1 May 2010.

[5] US Dept. Agriculture, "Colony Collapse Disorder Progress Report," USDA, Washington DC, 2010.

[6] I. Davis and R. Cullum-Kenyon, BBKA Guide to Beekeeping, London: Bloomsbury, 2012.

[7] N. Carreck and I. Williams, "The economic value of bees in the UK," *Bee World,* vol. 79, no. 3, pp. 115-123, 1998.

[8] European Comission - Directorate-General for Health & Consumers, "Honeybee Health," European Comission, Brussels, 2010.

[9] M. Allan, Interviewee, *Bee Monitoring.* [Interview]. 15 April 2013.

[10] E. Stalidzans and B. A, "Temperature changes above the upper hive body reveal the annual development periods of honey bee colonies," *Computers and*

*Electronics in Agriculture,* vol. 90, pp. 1-6, 2013.

[11] A. Zacepins and E. Stalidzans, "Architecture of automatized control system for honey bee indoor wintering process monitoring and control," in *International Carpathian Control Conference*, High Tatras, 2012.

[12] J. Nickeson, "About Scale Hives," Goddard Space Flight Center, NASA, 8 March 2010. [Online]. Available: http://honeybeenet.gsfc.nasa.gov/About/ScaleHives.htm. [Accessed 22 Aug. 2013].

[13] M. Bencsik and e. al, "Identification of the honeybee swarming process by analysing the time course of hive vibrations," *Computers and Electronics in Agriculture,* vol. 76, no. 1, pp. 44-50, 2011.

[14] Microsoft, "Microsoft .NET Gadgeteer," 19 April 2013. [Online]. Available: http://gadgeteer.codeplex.com/.

[15] Microsoft, "Home - Gadgeteer," 2011. [Online]. Available: http://www.netmf.com/gadgeteer/.

[16] LogMeIn Inc., "Xively - Public Cloud for the Internet of Things," 2013. [Online]. Available: https://xively.com/.

[17] Joyent Inc., "node.js," 2013. [Online]. Available: http://nodejs.org/.

[18] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," *Internet Computing, IEEE,* vol. 14, no. 6, pp. 80-83, 2010.

[19] B. Calder and e. al., "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," in *23rd ACM Symposium on Operating Systems Principles*, Cascais, 2011.

[20] The jQuery Foundation, "jQuery," 2013. [Online]. Available: http://jquery.com/.

[21] Bootstrap, "Bootstrap," 2013. [Online]. Available:

http://twbs.github.io/bootstrap/.

[22] O. Laursen, “Flot: Attractive JavaScript plotting for jQuery,” IOLA, 2013. [Online]. Available: http://www.flotcharts.org/.

[23] S. Ferrari and e. al., “Monitoring of swarming sounds in bee hives for early detection of the swarming period,” *Computers and Electronics in Agriculture,* vol. 64, no. 1, pp. 72-77, 2008.

[24] Universitat Wurzburg, “HOBOS - Honeybee Online Studies,” [Online]. Available: http://www.hobos.de/en.html.

[25] Arnia, “Bee Keepers,” [Online]. Available: http://bee-keepers.co.uk/.

[26] Beewiz, “Bee hive monitoring, bee keeping equipment,” [Online]. Available: http://www.beewise.eu/.

[27] Swienty, “BeeWatch Proffesional - hive scale,” [Online]. Available: http://www.swienty.com/shop/vare.asp?side=0&vareid=102360.

[28] “HiveTool,” [Online]. Available: http://hivetool.org/.

[29] Hydronics, “Honey Bee Counter,” 12 Oct. 2012. [Online]. Available: http://www.instructables.com/id/Honey-Bee-Counter/.

[30] F. w. Technology, “Beehive scale build details,” 15 Sep. 2011. [Online]. Available: http://makingthingswork.wordpress.com/2011/09/15/70/.

[31] G. Hudson, “Bee Hive Monitor,” 2011. [Online]. Available: http://openenergymonitor.org/emon/node/102.

[32] S. Richardson and S. Ruby, RESTful Web Services, O'Reilly Media, 2008.

[33] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures,* University of California, Irvine, 2000.

[34] J. Go, “Designing a Scalable Partitioning Strategy for Windows Azure Table

Storage," 5 Oct. 2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/windowsazure/hh508997.aspx.

[35] S. Stefanov, Object-Oriented JavaScript, Packt Publishing, 2008, pp. To-do.

[36] Weather Underground Inc., "Personal Weather Stations - Wunderground," 2013. [Online]. Available: http://www.wunderground.com/weatherstation/about.asp.

[37] Met Office, "Met Office WOW," [Online]. Available: http://wow.metoffice.gov.uk/home.

[38] NOAA, "Citizen Weather Observer Program," 2011. [Online]. Available: http://wxqa.com/.

[39] M. Kiessling, The Node Beginner Book, Learnpub, 2012.

### 6.4 Other Sources

Node.js webserver design: [39]

Diagrams were produced with http://Gliffy.com, using images from http://OpenClipArt.org

GHI documentation: https://www.ghielectronics.com/docs/ was used for small code samples to aid writing the program for the monitoring device in C#.

The Xively IoT [16] API was used as inspiration for the web API, with some conventions and naming borrowed.

Use case writing: [40]

UML diagrams: [41]

General software engineering: [42].

# Appendices

## A1 Use Cases

| Read Configuration from Storage |
|---|
| **ID:** UC-M1 |
| **Summary:** The MCU access the configuration file on the SD card to load the pre-defined settings. |
| **Actors:** MCU |
| **Main flow:**<br>1. Triggered by the MCU booting-up<br>2. The MCU uses the SD card module to read the XML configuration file into memory<br>3. The Device parses the XML and updates its default configuration with the new values<br>4. The MCU closes the file and terminates the use case. |
| **Alternative flows:**<br>2.1 The configuration file is not found so the use case ends.<br>3.1 The file cannot be parsed so no updates are made. |

| Connect to Wi-Fi |
|---|
| **ID:** UC-M2 |
| **Summary:** The MCU connects to a wireless network |
| **Actors:** MCU, Time |
| **Main flow:**<br>1. Triggered by UC-M1 (MCU reads configuration file)<br>2. The MCU accesses the Wi-Fi module<br>3. The Device searches for the specified network, joins the network and synchronises its clock<br>4. The MCU leaves the Wi-Fi module operational for future use. |
| **Alternative flows:**<br>1.1 Triggered by Time when network connectivity has been down for ten minutes.<br>3.1 The network is not found so the use case terminates. |

| Transmit Data Point |
| --- |
| **ID:** UC-M3 |
| **Summary:** The MCU reads a data point, which is sent to the Data API over HTTP |
| **Actors:** Time**,** MCU |
| **Main flow:**<br>1. Triggered by Time when an update is required as per the specified interval<br>2. The MCU retrieves current data from all its sensors<br>3. The Device saves all channels into a single data point<br>4. The MCU opens an internet connection<br>5. The Device creates an HTTP request with the data point as the body<br>6. The MCU pushes the request to the specified API endpoint. |
| **Alternative flows:**<br>4.1 The MCU finds no internet available<br>4.2 The Device buffers the data point to SD storage<br>4.3 Use case ends, go to UC-M4 (saving data point). |

| Save Data Point |
| --- |
| **ID:** UC-M4 |
| **Summary:** The Device saves a data point to the external SD card buffer when Wi-Fi connection has failed. |
| **Actors:** Time, MCU |
| **Main flow:**<br>1. Triggered by failure of UC-M3 (transmit data point)<br>2. The MCU loads the SD storage module<br>3. The Device saves the data point to the buffer file<br>4. The MCU closes the SD connection. |
| **Alternative flows:**<br>1.1. The buffer is not found<br>1.2. The Device requests the MCU to create the file<br>1.3. The file is created and the use case continues. |

| Transmit Buffered Data Points |
| --- |
| **ID:** UC-M5 |

| |
|---|
| **Summary:** The MCU reads from the SD card data point buffer and transmit all the data to the API |
| **Actors:** Time**,** MCU |
| **Main flow:**<br>1. Triggered by Time when an update is required as per the specified interval<br>2. The Device checks for the existence of data points in the buffer file<br>3. The MCU loads the data points into memory<br>4. The Device reads 100 points at a time, bundles them into API-compatible format and makes an HTTP request<br>5. The MCU fulfils the request and receives the server response<br>6. The Device reads the 200 response and so deletes the buffer**.** |
| **Alternative flows:**<br>2.1. No data points are found so the use case ends<br>5.1. The request fails so the use case terminates. |

| **Modify Settings** |
|---|
| **ID:** UC-U1 |
| **Summary:** The User modifies some of the Application settings in response to a change in their personal requirements or Device hardware availability. |
| **Actors:** User |
| **Main flow:**<br>1. Triggered when the User navigates to the settings area of the Application<br>2. The User completes and submits the form for the setting they wish to be changed<br>3. The Application processes the changes and updates the UI<br>4. The User reviews the changes and likes them<br>5. The User submits their password to commit the changes<br>6. The Application processes the commit<br>7. The User receives feedback that it was successful. |
| **Alternative flows:**<br>4.1. The User rejects the changes and ends the use case<br><br>7.1. The commit fails due to an incorrect password<br>7.2. The User retries; go to step 6<br>7.2.1. The User terminates the use case. |

| Export Data |
| --- |
| **ID:** UC-U2 |
| **Summary:** The User exports some historical data using the Application. |
| **Actors:** User |
| **Main flow:**<br>1. Triggered when the User navigates to the history area of the Application<br>2. The User selects which date range and data format to export<br>3. The Application validates and fulfils the request<br>4. The User receives the data in their chosen format. |
| **Alternative flows:**<br>3.1. The Application finds no valid data for the specified period<br>3.2. The User receives a corresponding error message<br>3.3. The User reties; go to step 2<br>3.3.1. The User terminates the use case |

| Analyse Trends |
| --- |
| **ID:** UC-U3 |
| **Summary:** The User browses the Application to analyse trends in his Beehive. |
| **Actors:** User |
| **Main flow:**<br>1. Triggered when the User navigates to the Application graphs<br>2. The User modifies the graph by selecting which period and variables to view<br>3. The Application fulfils the request dynamically<br>4. The User observes various interesting trends so resumes the use case at step 2. |
| **Alternative flows:**<br>4.1. The User finds no more trends of use so terminates the use case. |

## A2 Competitor Analysis

### Beewatch.biz

Source: http://www.beewatch.biz:8080/Basic/ (username GUEST, password guest, then select scale).

A commercial service for mass and environ monitoring using their sensor. Data transmission is via GSM.

Good graphs but the table is useless and buggy. Decent export function but little else can be done.

[img1]

### Other commercial

Some behind a pay wall – e.g. Hivemind (http://hivemind.co.nz/). Data transmission is via GSM.

Arnia (http://www.arnia.co.uk) claim to have sensors for: environ, mass, sound, vibration, CO2; all sent to web server with a variety of analytics (queen, swarm, brood). Pictures exist of attractive and comprehensive web app, but no details on how to get it as the product is in RnD stage and only being tested in research field (see BBC article).

### Hivetool.org

Source: http://hivetool.org/

Environ and mass data in research/education situations. Data sent to local web server (power hungry). No concrete platform – a confused and unrelated collection of hardware and software. Good points: deployed in about 10-15 hives, open-source.

Atrocious organisation – a real mess of info material and data; shocking graphics, and slow.

[img2]

### OpenEnergyMonitor

Source: http://openenergymonitor.org/emon/beehive/v2

Environ only (but x4 temp in different parts of the hive). Data is wifi-ed to local PC and relayed over http to simple gauges and graphs on blog posting. Battery operated.

[img3]

## A3 Requirements Gathering

Mr Allan (email) primary requirements: camera covering the entrance, *external* temperature, hive/supers mass, movement detection. Secondary detections: disease, stores, pollen, laying pattern, mites, full supers. Not interested: internal temp/hum. Power: battery (on-board or nearby). Data: live broadcast to web.

Mr McGuire (email) primary requirements: internal temp, brood core-edge temp difference, swarm detection (suggests detecting: queen pheromone, drone level, queen cells being tended!), movement, supers' mass. Secondary: vibration, in/out rate. Power: solar or on-board. Data: n/a.

Romsey & District BKA (email) primary requirements: internal temp/hum, queen detection, swarm detection, identify non-bees at entrance (wasps mainly). Secondary: hive mass. Power: n/a. Data: n/a/.

Mr Flottum, *Senior Editor of Bee Culture Magazine* (http://colonymonitoring.com/cmwp/for-entrepreneurs/) primaries: Varroa level, queen detection. Secondary: weight, environ. Power: battery. Data: n/a.

### Round Two

Mr Allan, Mr Mcguire, and Mr Fatland were shown the prototype, and thus evaluated the GUI and specified key requirements for it.

## A4 Building a Gadgeteer Mass Sensor

## A5 Source Code

Gadgeteer, Gadgeteer Simulator, Node.js server, public html.

## A6 System Manual

Full source code is available at the GitHub HiveSense repository.

### Gadgeteer Device

#### Prerequisites

- Gadgeteer + modules' SDK (https://www.ghielectronics.com/support/.net-micro-framework)

- Visual Studio IDE (any edition 2010+)
- Hardware listed in sec x.x.x.

#### Instructions

1. Assemble the physical modules as shown in the diagram (GadgeteerDesign_V2min.png from sec. 4.x.x.x)
2. Create a folder called 'hivesense' at the ROOT of your SD card, and place a plain text file called 'config.xml' into this directory, with the contents of fig.x, modified to match your environment
3. Insert the SD card and connect the device to your PC through USB
4. Load the HiveSense.sln Visual Studio file and hit F5 to deploy and debug.

### Node.js Web API

#### Prerequisites

- Node.js 0.8+ environment
- Azure SDK.

The specifics depend on where you choose to deploy the system: on an Azure web site or on a custom machine.

#### Custom machine (also good for local development)

1. Install Node.js and the Azure SDK
2. Set the custom environment variable `EMULATED=1` on the command line
3. Boot the Azure Storage Emulator (part of the SDK)
4. On the command line, launch the server with `node ./server`
5. Access to the API resources is through port 1337.

#### Azure Web Site (AWS)

1. Set up an Azure Web Site (AWS) and an Azure Table Storage account
2. In your Azure web site management portal, go to configure->app settings and input the following keys:
    a. AZURE_STORAGE_ACCOUNT: [Table Storage account name]
    b. AZURE_STORAGE_ACCESS_KEY: [Primary access key for this account]
3. Deploy the code through GitHub, FTP, or the Azure SDK – see AWS docs.

### Web Application

This is deployed along with the web API. No additional configuration is required.

## A7 User Manual

HiveSense – automated beehive health monitoring for beekeepers.

HiveSense is a software system to aid beekeepers monitor some simple properties of their beehive colonies. It consists of some hardware for the physical monitoring, a web API for storing the data and allowing it to be queried, and a web application to view the hive data in real time. Additionally, an alarms feature allows the beekeeper to be alerted when one of the measurement properties exceeds a user-defined value.

The software stack is flexible to allow hardware developers to make new sensors for measuring more advanced hive properties, which can be easily incorporated into the web API and application. The web API is flexible so that software developers can make their own web application from the available data.

### Device

#### Positioning

The plate hosting the internal sensors should be fixed to the inner wall of the beehive. The camera and secondary thermometer are loose and connected by long cables to the mainboard so these can be positioned wherever you desire.

#### Power

Any source of 5-30V is acceptable (including appropriate batteries). ~9V is recommended. Either micro-USB or 3mm plug-type connections can be used.

#### Maintenance / failure

The push button on the mainboard can be depressed to reset the device in case of failure; no other maintenance is required.

#### Settings

With the SD card inserted into your computer, open the */hiveSense/config.xml* file in a text editor. Modify the text between the xml tags to change settings; the following can be freely configured:

- Details of the Wi-Fi network where the device will be deployed
- Sensitivity of the motion detector
- Update frequency (how often the hive properties are measured)
- URL of the API where data is to be sent.

Specific settings details are given in the file itself, as xml comments.

### Retrieving raw data

Every single data point is recorded onto the SD card for your convenience. The log is available at */hiveSense/datalogAlways.csv,* and can be freely deleted to save space at any time, as it is not used by the device except for writing (the *datalog.csv* is used for internal buffering purposes and should be left alone). Remember to turn-off the power of the device when removing and inserting the SD card.

## API

docs go here, with full text from 'About' tab as a quote

## Web Application

Although the API offers all the resources needed to develop your own web application, a pre-built one has been developed for those seeking a simple and user-friendly way to monitor the health of their beehive.

In a modern (2012+) browser of your choice, navigate to the URL where the web server has been deployed. You may also wish to view an example of a running HiveSense application.

The application is a single-page web application, which means all resources are loaded at start-up, or loaded in the background as needed when the application is running. All data updates occur automatically so no page refreshes should be performed. It has been designed to work in a range of browser sizes, so is suitable for both mobile and desktop devices. There are three main views for viewing the hive data, as tabs, in the application, plus a tab for settings, and one for documentation.

### Dashboard

[PIC: annotated dashboard]

This is the view that loads on start-up, and provides an overview of the current status of the beehive (see fig x.). The dashboard updates when new data is available from the hive, or settings are changed (see sec. Settings). Most of the dashboard elements can be hovered over to provide a description. The core elements are:

- Value for each measurement variable, with trend showing its movement since the last reading (up, down, or flat).
- Graph of the last ~3hrs for each variable.
- Photo from the hive entrance for viewing your bees.
- Data indicator light – red when no recent data, green otherwise (NB: g amber when requesting data from the server.
- Alarms – red or green LED indicating whether the particular threshold has been breached.
- Local weather – temperature and weather condition from the nearest weather station to your beehive.

### Alarms feature

An important side note should be made at this point. The alarms shown on the dashboard are continually monitored by the server as data is sent from the HiveSense Device. This means you can be alerted to the raising of a red alarm state without having to visit the dashboard; the alert will go straight to your email inbox. Details of configuring this are available in the 'Settings' section below.

### Graphs View

This view contains a customisable graph of the hive data. Use the left panel to toggle the variables, and to change the viewing period.

[PIC: GV]

### History View

Here you can view and export the hive data as tables. The default will load a table of the past 24hrs data. To change the period, use the date pickers on the left to select start and end dates. The interval between consecutive data points will be automatically chosen based on the length of the period selected.

To export data, simply choose the format (CSV, perfect for Microsoft Excel, is the default), and click the ‘export’ button.

[PIC: HV]

### Settings

This is available through the tab with the ‘gear’ icon. Most application settings can be changed through the user interface. Any changes made can be previewed immediately before they are saved, so after making changes you should see their effect on the Dashboard. When you are happy with the changes, they can be saved permanently using the ‘commit’ button on the top-right panel of the settings tab – password required (default: ‘livehive’).

[PIC: SV]

All settings, including some not available through the user interface, can be changed by modifying the raw file that stores the settings in JSON format on the server. This is done through the ‘Advanced’ section at the bottom of the settings page, and as the name suggests is only intended for advanced users with a working knowledge of JSON. A full description of the settings available is as follows:

[PIC: ASV with loaded JSON sample]

- “beek”: Your name, used in emails
- “email”: Your email address, where alerts are sent
- “hiveName”: Name of your beehive, used as the web application title
- “wxplace”: City in the UK to use for the local weather report
- “updateRate”: Update frequency of the Dashboard. Highly advisable to make this match the update frequency of the Device.
- “alarms”: Array of alarms for the Dashboard and email alerts
    - “label”: Name of the alarm
    - “sensor”: ID of the sensor (see “sensors” setting below) monitored by this alarm
    - “value”: Threshold value for the sensor, when breached the alarm is raised

    - "type": Must be "high" or "low", the direction of the breach to trigger the alert - when the sensor value goes above or below the threshold "value" respectively. See examples for better understanding.
    - "email": Maximum frequency, in hours, at which to send alert emails for threshold breaches of this alarm. Set to 0 to disable emailing.
- "sensors": Array of sensor variables to display on the Dashboard
    - "id": ID of the sensor variable. Must match one of those produced by the Device (see API – current data feed – for which are available)
    - "label": User-friendly name for the variable
    - "unit": Physical unit of measurement
    - "isdefault": Boolean. Do not delete sensors where this is true, as they are fixed to the user interface. When adding new sensors, set false
    - "graphOptions": Styling for the graphs of the variable
        - "labelShort": Short name for the legend
        - "colourGraph": Line colour in hexadecimal
        - "colourGd1": Start colour for the background gradient
        - "colourGd2": End colour for the background gradient.

## A8 Blog

Grab from WordPress or just link – NB that only posts with 'BenLR' listed as the author were written for this project, the remaining were inherited from the 'hack weekend' which inspired this project.

## A9 Work process

Sample of activity log plus full weekly summary