Beehive Health Monitoring with .NET Gadgeteer

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 hive 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 bee keepers could be better-informed about the health of their hives than is possible at present with traditional tools, through the use of web services and user interfaces accessing internet-connected embedded devices
- Produce easily extensible open-source code with extensive documentation so that the bee keeping community can more easily make further progress in the field of 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 lab-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 getting potentially unreliable hardware to work (e.g. GSM, batteries).

- The Gadgeteer platform is to be used for the hardware, though some sensors will require interfacing with this platform as not all are currently available in pre-prepared form
- A web-facing application will be the single system front-end
- Reliability of data is a secondary hardware concern, whilst software robustness is a priority; In general, hardware sophistication (including enclosures and power supply) will be limited by time and cost
- The full system will be designed to work on a single hive, as testing multiple devices is beyond limitations of available time and funding.

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 'hackathon' Mr Johnston ran for experienced developers to experiment with Gadgeteer [s9]. 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 prototype model. By frequently building prototypes and testing them on users and in the laboratory, valuable early feedback can be introduced to better inform the project direction and the final system [Mythical Man Month][]. This iterative method is ideal given the rather broad areas to be researched and developed, from low-level hardware concerns up to high-level software design and implementation with a number of disparate technologies.

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 Bee colonies

The European honeybee is in trouble. In the past decade, colonies have been dying out a rate of about 30% a year [bee book], with no known cause [ibra.org.uk – look for research paper]. This is sometimes termed Colony Collapse Disorder (CCD), and has sparked a great deal of research into what may be responsible []. Theories include climate change, human pesticide use, and parasitic disease []. What's needed 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 [s7]. This huge impact makes CCD a great concern, and recently there have 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 []. Moreover, most apiaries (beehive 'farms') are run by amateurs []. 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 [], and is 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 [].

2.2 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 []. 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 don't 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 [email – Mark Allan]. 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 configurable warning systems are therefore crucial.

2.3 Properties to measure

2.3.1 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 [p12, p17]. In particular, it is the temperature differential between the hive core and the natural external environment that is the most important indicator [s3] of hive activity and health. This differential approaches zero when a colony dies [s4], so by observing this an intervention can 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 [].

2.3.2 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 [p64, p68], sharp changes are expected from bulk departure (swarming, a common problem for beekeepers), and long-term trends are expected to indicate honey storage []. At present, honey quantities are estimated by the practice of 'hefting', whereby a part of the hive is physically lifted from the base and jiggled around to gather an estimate [p70]. 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.

2.3.3 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 [p69]. By detecting movement with an accelerometer, and opening of the hive lid with a light sensor, it is expected that any form of disturbance can be detected.

2.3.4 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 []. Additionally, the entrance is where foreign invaders such as wasps will stage an attack []. It is therefore desirable to be able to visually monitor the events in this area. It is expected that a camera will be perform this job admirably.

2.3.5 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 and disease-detection [], 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.

2.3.6 Sound

Sounds in the hive could be important indicator of activity (drone ratio [p22], swarming [pxx], worker jobs (nurse vs. food collector) and queen presence [s5], disease detection/CCD [s1]). . Sensitive accelerometers embedded in the hive wall have been shown to pick up the ultrasound-induced vibrations of bees communication (swarm prediction for example) [s6], though Gadgeteer is likely not sensitive enough. Microphone could, however, provide a crude estimate, though analysis would be tricky.

2.4 System components

2.4.1 Gadgeteer platform

Microsoft's Gadgeteer platform is a set of open-source hardware consisting of small modules which plug into an ARM-powered mainboard - a microcontroller running on the .NET Micro Framework (NETMF), along with an open-source SDK for each module and mainboard. NETMF is essentially a subset of Microsoft's .NET framework that can run on devices with limited memory and processing power, with some additional

features specifically targeted to embedded applications []. Programs on NETMF are written in C# and developed in the Visual Studio IDE.

Gadgeteer's plug-based architecture 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 devices such as the Arduino and Raspberry Pi []. The downside is the limited availability of off-the-shelf modules, though it is possible to expose the inner mainboard connections and thus make a custom module, then program it using the core NETMF library. Programming the microcontroller is done using the Gadgeteer SDK [], which runs on top of NETMF.

For beehive monitoring, the useful Gadgeteer modules are mainly passive sensors like thermometers, though other utility modules will be needed, such as one for internet connection.

2.4.2 Web API

In early prototypes of the system, an Internet of Things web service called Xively [] was used. This provides the tools to connect an embedded device to the internet through an API and database backend. This service worked well for extracting current data from a fixed number of sensors. However, it has a number of major drawbacks which ultimately meant that a replacement had to be found. The principal problem is the limitation and difficulty in retrieval of historical data, though the less-than-desirable flexibility for changing the sensor base proved problematic too.

To this effect, it was decided that a custom API would be built with Node.js, and designed to better fit the flexibility requirement of this project. Node is a server-side software platform ideal for building networking applications such as RESTful APIs []. Writing a Node application is done in JavaScript, meaning seamless integration with the client-side web application – easy passing of JSON objects and no impedance mismatch as is common with rivals such as ASP.NET and PHP []. The networking model is single threaded asynchronous, in contrast with Apache's synchronous model – when IO is required, Node continues running the thread and sets up a callback to execute when the IO is finished, rather than Apache's technique of halting the thread on IO and using other threads. Apache is better for algorithmically intense applications such as multimedia processing, but Node performs better for lightweight

RESTful services []. 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 database. Microsoft's Azure Table service [] was chosen to fulfil this role because of its flexibility and scalability, important given that vast amounts of data can be generated by a continuously running monitoring device. Azure Tables offer structured storage of key-value pairs in a NoSQL database. The service is strongly oriented towards providing scalability by offering automatic load balancing [NEED TO REVIEW THE ADVANTAGES OF ATS].

2.4.3 Web front-end

The system's frontend GUI will be built using the industry-standard web technology stack: HTML5, CSS3, and JavaScript with JQuery []. Additionally, to speed up development of UI components, the Bootstrap framework will be used []. Since frontend development is not part of this project's scope, 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 [].

2.4.4 Version control

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

2.4.5 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 the best possible feedback, 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. The current project is running on Azure with a free student trial license.

2.5 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 [s1], swarm-detection with thermometers and microphones [s2], and detecting signs of colony development by monitoring temperature differentials [s3]. Various patents have also been submitted for acoustic monitoring devices [s5] [apidictor]. All these efforts lack the web connection and front-end aspect of this project's intention, but have provided the author with useful ideas for monitoring (see §2.3).

The HOBOS teaching project is a sophisticated and fairly complete all-round monitoring system with a live video web stream of the hive entrance, and live graphs of all the sensors []. 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-used hive monitoring system.

A few commercial integrated monitoring systems exist too, at least one of which has an online user interface for the live data []. 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 by anyone.

Various hobbyist attempts at a few of the sensors, some with web service connection, can be found in the literature [][][]. However, no serious integrated multi-sensor solution can be found, and the web interfaces that do exist leave much to be desired in functionality and modernity (see appendix?).

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 [], due to its arguable position as the most important beast on the planet []. Despite this, the millions of bee colonies across the UK and everywhere globally 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, and even to professionals. The small number of commercial solutions lack expansibility, and the few hobbyist solutions are inadequate or overly specific, and mostly poorly documented.

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' includes 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 (§2.5)
- Results from prototyping and testing in laboratory conditions
- Limitations provided by the project scope (namely time and cost).

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. An important point to note is that beekeepers come from a wide range of backgrounds [], so it important to cater for both developers and those less-technically minded.

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, for example from the GHI catalogue [ ]):

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 [], which are desirable in any system for data distribution. The core constraints followed, as identified by the author of REST [], 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 [], 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 limit per query).

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 []. 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

4.2.3.1 Class design

An MVC architectural design approach was adopted, as is natural and very common for web applications [].

4.2.3.2 User Experience

5 Testing

Field tests, user-acceptance for the web app. Replacing components (e.g. Gadgeteer simulator acting as Monitoring Device).

6 Evaluation

6.1 Extension Scenarios [Think about where to put this]

How easy is for developers to do the following, given the current system/code design?

1. Implement new sensor on the monitoring device
2. Implement new chart on the web app
3. Change API to use a different database

Current cost of device - ~£200 []. Expected drop to £50 in specialised production, at most. As mention in sec.2, beehives' value is £200m across 250,000 hives in UK due to

pollination alone (a few 10s more from honey [s8]) => > £800 per hive => saving the life of a colony is very valuable.

Bibliography

[1] x
[2] y
[3] z

Node.js start-up: https://leanpub.com/nodebeginner

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

GHI documentation for bits of code helpful for using specific modules (specific references marked in the code listing itself).

Xively IoT API used as inspiration for the web API, with some conventions and naming borrowed.

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: 1. Triggered by the MCU booting-up 2. The MCU uses the SD card module to read the XML configuration file into memory 3. The Device parses the XML and updates its default configuration with the new values 4. The MCU closes the file and terminates the use case.
Alternative flows: 2.1 The configuration file is not found so the use case ends. 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: 1. Triggered by UC-M1 (MCU reads configuration file) 2. The MCU accesses the Wi-Fi module 3. The Device searches for the specified network, joins the network and synchronises its clock 4. The MCU leaves the Wi-Fi module operational for future use.
Alternative flows: 1.1 Triggered by Time when network connectivity has been down for ten minutes. 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: 1. Triggered by Time when an update is required as per the specified interval 2. The MCU retrieves current data from all its sensors 3. The Device saves all channels into a single data point 4. The MCU opens an internet connection 5. The Device creates an HTTP request with the data point as the body 6. The MCU pushes the request to the specified API endpoint.
Alternative flows: 4.1 The MCU finds no internet available 4.2 The Device buffers the data point to SD storage 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: 1. Triggered by failure of UC-M3 (transmit data point) 2. The MCU loads the SD storage module 3. The Device saves the data point to the buffer file 4. The MCU closes the SD connection.
Alternative flows: 3.1. The buffer is not found 3.2. The Device requests the MCU to create the file 3.3. 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: 1. Triggered by Time when an update is required as per the specified interval 2. The Device checks for the existence of data points in the buffer file 3. The MCU loads the data points into memory 4. The Device reads 100 points at a time, bundles them into API-compatible format and makes an HTTP request 5. The MCU fulfils the request and receives the server response 6. The Device reads the 200 response and so deletes the buffer**.**
Alternative flows: 2.1. No data points are found so the use case ends 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: 1. Triggered when the User navigates to the settings area of the Application 2. The User completes and submits the form for the setting they wish to be changed 3. The Application processes the changes and updates the UI 4. The User reviews the changes and likes them 5. The User submits their password to commit the changes 6. The Application processes the commit 7. The User receives feedback that it was successful
Alternative flows: 4.1. The User rejects the changes and ends the use case 7.1. The commit fails due to an incorrect password 7.2. The User retries; go to step 6 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: 1. Triggered when the User navigates to the history area of the Application 2. The User selects which date range and data format to export 3. The Application validates and fulfils the request 4. The User receives the data in their chosen format.
Alternative flows: 3.1. The Application finds no valid data for the specified period 3.2. The User receives a corresponding error message 3.3. The User reties; go to step 2 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: 1. Triggered when the User navigates to the Application graphs 2. The User modifies the graph by selecting which period and variables to view 3. The Application fulfils the request dynamically 4. The User observes various interesting trends so resumes the use case at step 2.
Alternative flows: 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).

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

See RESTmodel.js for notes on connecting to Azure

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 modules as 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 'configuration.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.

Node.js Web API

Prerequisites

- Node.js 0.8+ environment
- Azure Table Service environment

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 testing)

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 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

A7 User Manual

Gadgeteer 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/configuration.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

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 Mid-project presentation

As-per Dean’s suggestion

A10 Work process

Sample of activity log plus full weekly summary