University of Glasgow | School of Computing Science

ESE3: Chose Your Own

Mustafa Altay
Kristian Hentschel
Joshua Marks
Kyle van der Merwe

Level 3 Project — 18 March 2013

Abstract

The abstract goes here

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

1.1 Introduction

This is the documentation for the Electronic and Software Engineering Level 3 Team Project for Team N. Out of the 3 ESE project proposals our team picked the ”Choose your own project” supervised by Dr. Martin Macauley.

It was suggested by Dr. Macauley that a group within the University may require something built for them and that this would create a real world evironment with a client and system requirements that were externally defined. The UGRacing team had previously given projects to previous teams and were therefore sought out for a source of a project. They requested that a wireless weighing system was built in order to measure the weight of the car that they were building over the course of the year.

1.2 Background

UGR is a group of students competing in a competition called Formula Student. This is a world wide competition run by university groups with 100 entrants from 34 countries (numbers taken from the Formula Student 2012 competition) carried out every year; the goal of each group is to create a single seat race car that competes against the other groups at the Silverstone Grand Prix track in June/July.

The cars produced are assessed on a number of attributes including: handling, robustness, speed and acceleration.

1.3 Motivation

UGRacing needed a way to measure the weight of each wheel in order to optimize the weight distribution of the car. The team are currently using standard bathroom scales to measure the weight distrobution of the car. Being highly unpractical UGRacing require a solution that is safe, accurate

and portable. The creation of a wireless system will allow all readings to be viewed by a generic handheld unit. Using a wireless system will also reduce the number of potential trip hazards in the workshop.

1.4 Requirements

This section details both how the requirements were gathered as well as what the requirements of the system were defined by.

1.4.1 Requirements Gathering

During the first meeting with the UGR liason Jonathan Siviter, the intial problem description was outlined along with motivations and background of the UGR. Once discovering what it was that UGR wanted built the project was accepted and meetings planned to gather further requirements.

Over the course of the project there has been one additional meeting in order to better detail the exact requirements of the system and continuous email correspondence once further questions became apparent.

1.4.2 System Requirements

The system requirements as defined by the UGR team's liason were split into functional and non-functional requirements.

Functional Requirements

- The system must be wireless.
- Each wheel must be weighed simultaniously.
- Basic data analysis such as differential weights must be available.
- Accuracy should be $<$1kg.
- Wireless system must work to a range of 5-10 meters.
- Max expected total load 300kg.

Non-Functional Requirements

- The system should be able to display the readings to a generic device such as an iphone, android phone or tablet.
- There should be a button to initialise readings.

- System must be portable.
- System must be compatible with the load cells that would be produced by a different team.
- Each of the scale control units should be roughly 25cm^2.
- Scale unit meet IP65 requirements(dust sealed, resistant to low powered jets of water from all directions).
- The scale units should be battery powered, using common batteries (coin cells or AAA).
- There should be a physical on-off switch at each unit.

Chapter 2

Design

2.1 Design Overview

The proposed solution is to have 4 simple scale units. These units will simply recieve the analogue signal from a loadcell, convert it into a digital form and then send that to a central unit via a wireless communication module. The central unit will recieve messages containing the weight from each of the 4 different scale units. It will then need to provide this information to a user in a standardised way in order to make it accessible from a generic device as required (see sec**??**).

2.1.1 Scale Units

These units are where the majority of the work is done, they are essentially load cell control units. This means that they are the responsible for interfacing to the base analogue output of the load cell, configuring it to a manageable form and then transmitting it to the central unit. This will require several components cheifly some form of microcontroller with an ADC, a wirelss communication device, a circuit that travels through a load cell into a wheatstone bridge and then into instrumentation amplifier in order to increase accuracy.

2.1.2 Central Unit

This is the central hub of the system; where all the information is brought together, analysed and provideded to the user of the system. This component will need a microcontroller and a wireless communication device capable of communicating with all 4 of the scale units. It will then need to provide this information to a user in a standardised way in order to make it accessible from a generic device as required”

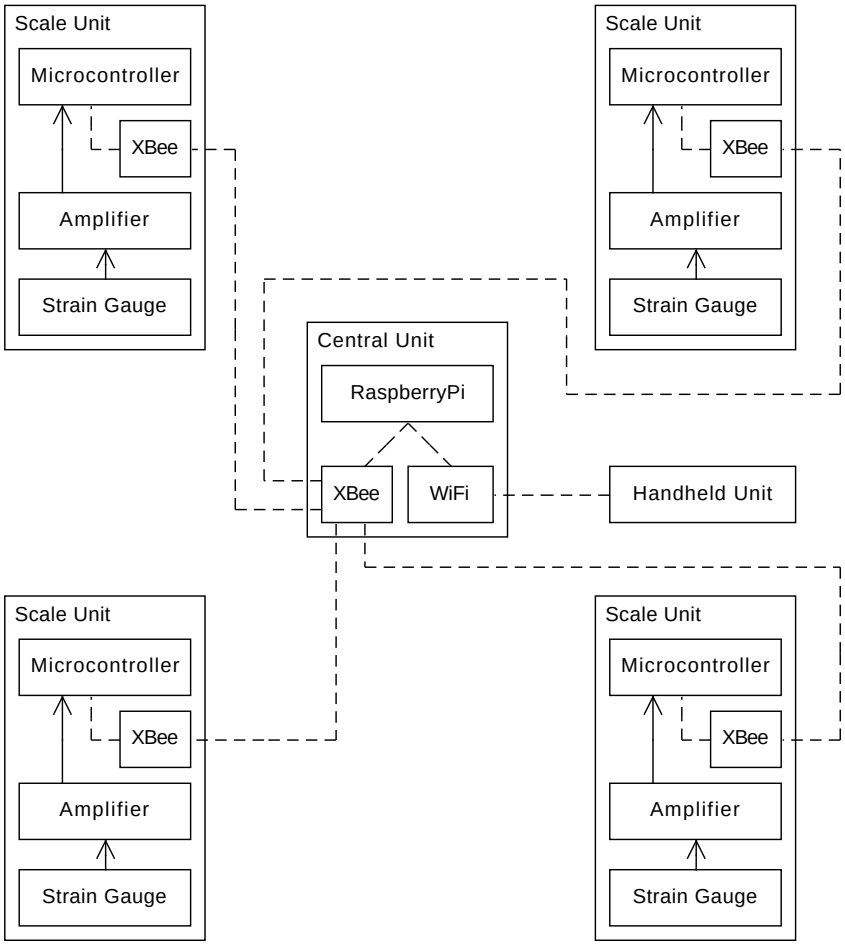

Figure 2.1: Block Diagram

2.2 Hardware Design

2.3 Software Design

Chapter 3

Implementation

3.1 Software Implementation

The "software" encompasses all programs that run on the scale units, the master unit, and the user interface client-side code. Despite the vast differences between these targets, the software had to be designed for maximum re-use. A number of feasibility studies were completed before the final approach was selected.

3.1.1 Master Unit Considerations

The master unit would run a full Linux operating systems, therefore more high-level languages and libraries could in theory be used.

Since the user interface was a web application, the master unit had to run a simple web-server that could serve the interface (HTML, Javascript, CSS files) as well as answer requests for dynamically generated information gathered from the scale units. As an initial prototype, the Python language with a webserver module [**?**] was used. An open-source project [**?**] was discovered that would allow control of the GPIO pins and could potentially be adapted for this purpose. However, the serial port communication, and handling of single bytes for communicating with the radio transmitters were found to be insufficiently reliable and scalable.

Instead, it was decided to program the entire system in C, as this would allow for some portions of the code to be re-used on the scale units. This decision was affected by the fact that not much time was left to complete the project, and the web-server and client-side UI were deemed as optional features that could eventually be added in on top of this.

3.1.2 Layered Approach

A layered approach to the software architecture was taken. The system is modelled in three main layers, roughly following the OSI network systems model [**?**]. Each layer would define a well-known interface, and only the layer above would need to access this. Each layer can have several

implementations to account for the fact that the embedded scale unit system would be very different from the Linux system on the master unit.

- **Transport Layer:** The lowest layer maps primitive read and write methods to the underlying serial devices and provides buffering for incoming data.
- **Packet Layer:** This implements a packet protocol and provides methods for parsing incoming data, as well as packaging outgoing data.
- **Application Layer:** The highest layer implements the individual application functionality and defines responses to incoming packets, as well as interacting with other system components, such as the user interface or the analogue-to-digital converter.

3.1.3 Packet Protocol

The ZigBee protocol defines a wrapper packet protocol that is used for sending command and data frames across the serial link in the API mode [**?**, page 35]. This is then translated into another packet format used for over-the-air transmission.

ZigBee nodes can operate in two seperate modes. Firstly, transparent mode is the most basic communication, where only very basic pacektization is provided. However, this is very simple to get running for a basic point-to-point network. Unfortunately, once multiple nodes attempt to transmit at once, the packets become interleaved, rendering this mode not very useful for the envisioned network architecture. Still, it proved useful for initial debugging and manually sending data by connecting the device directly to a serial terminal.

Originally, a simple packet system was devised to run on top of the transparent mode. This was later found to be too unreliable and inefficient due to the aforementioned interleaving of data. Since this design was very similar to the ZigBee API protocol, few changes were required to send the payload data wrapped in packages conforming to this.

The payload consists of an *Op-Code* to signify the current operation (ping, pong, measure request, measure response), a *device identifier* that is set at compile-time, and the actual data in the case of a measure response. This data is encoded as, in this case four, hexadecimal ASCII characters rather than directly representing it as bytes. This decision was made to avoid having to escape control characters such as `7E` which defines the beginning of a packet in the ZigBee API. In the original design a length byte for the data was included, but this was abandoned since the wrapping API packet already contains a length field.

3.1.4 User Interface

The user interface to the system has been defined during the requirements gathering process to be a very simple web application that should support taking readings of the current system state by intitiating measurements, as well as providing simple calibration of the scales. Therefore a mock-up was drawn up, and based on this a static HTML/CSS prototype was generated (See figure **??**). This was then made interactive by attaching javascript actions to the buttons.

The client side application (the aforementioned javascript, running in a web-browser) would make asynchronous requests to special URLs on the server that are mapped to methods initiating data transfers between the master and scale units, or returning data previously stored on the server.

- `/api/data` returns a JSON [**?**] object containing the data stored on the server for all sensors.
- `/api/calibrate` initiates a measurement on all scale units and sets the values received to be the zero-points for that particular sensor.
- `/api/measure` initiates a measurement and stores the results in a server-side structure to be later retreived by a `data` request.

It was decided that for the prototype, the master unit program was to be integrated with a previously (for Networked Systems 3 coursework [**?**]) produced web server implementation, which should require minimal adjustments to support the added API functionality.

3.1.5 Transport Layer

The transport layer provides an interface to the serial link used to communicate with the ZigBee nodes.

Exposed Transport API

The interface used by higher layers is exposed through `zb_transport.h`. It is modelled as a subset of the most common network operations, only implementing those required by the system – that is, reading data as individual characters for passing them to the packet layer parser, and writing data as blocks of multiple bytes, for sending complete packets.

First of all, the system must be initialised using `zb_transport_init()`. Then data can be received character by character, in the order received, through the blocking `zb_getc()` call. Finally, data can be sent to the device by providing an array of bytes and its size to `zb_send()`. `zb_transport_stop()` has been added to perform clean-up operations such as closing file handles on implementations that require it.

Linux Implementation

Since the Raspberry Pi runs a complete Linux distribution (in this case, a stripped-down version of Raspbian, [**?**], is used), the microprocessor's UART component is exposed as a serial port to the system, named `/dev/ttyAMA0`. This port can be accessed using standard file operation system calls such as `read` and `write` after having been initialised by opening the file and setting a number of standard serial options [**?**].

A separate thread (implemented using pthreads) is running in the background, continuously monitoring the serial port by blocking on `read()`. Any characters are received on the port, these get

stored in a thread safe bounded buffer structure from which they are retrieved by a higher layer implementation. Thread safety is ensured using pthread condition variables and locks.

Bare Metal Implementation

On the microcontroller, no threading capabilities besides raw interrupts are available, and the serial link is implemented as a peripheral component of the microcontroller. ST provides driver libraries for configuring and accessing such peripherals which have been used extensively for this implementation.

The USART peripheral provides a small hardware buffer that is currently used as the only buffer in this implementation due to the low frequency of packets and relatively short non-IO bursts in the software. Therefore, the `zb_getc` implementation is currently busy-waiting for the memory-mapped USART status register to clear its *RXNE* (Receive buffer non-empty) flag. This is a similiar structure to waiting for a condition variable in a pthread program.

This is a polling method and using busy waiting, which is inherently using more power and is less flexible than waiting for interrupts. However, for prototyping and debugging, this made the testing and implementation much easier. It is expected that this can be re-developed using interrupts and additional software buffering.

3.1.6 Packet Layer

The packet layer provides an abstraction for sending and receiving custom data in a fixed format (made up of an op-code, a sender identifier, and an arbitrary number of 'data' bytes) over the transport layer. Its implementation is identical for all units, the only changes required are in the application layer's initialisation calls.

The first implementation, which was used for testing the transport layer due its simplicity, was based on a custom packet format that had nothing more than the aforementioned data content, a delimeter, and a checksum. The ZigBee units were pre-configured with destination addresses and options, and used in transparent mode.

Due to the issues explained in **??**, transparent mode had to be dropped and a second packet layer implementation tailored to the ZigBee API mode was developed using the same API calls. This is described in the following sections.

Packet API

The public API for the packet layer is defined in `zb_packets.h`.

Initialisation and settings are provided through the `zb_packets_init`, `zb_set_broadcast_mode`, and `zb_set_device_id` methods. In the current implementation, broadcast mode is the only way to set the target address: The master unit sends broadcast packages that are received by all scale units, and the scale units send unicast packages targeted only at the master unit. This state is held in a global variable, and it would be trivial to add functionality to support arbitrary target addresses,

though this would require for the application layer to know about network or device addresses of the destination.

ZigBee Packet Structure and Parsing Methodology

The ZigBee API defines data frames transmitted across the serial link between the host and ZigBee device. These packets are parsed and, if the packet is a transmission request, wrapped in a different 802.15.4 packet for transmission over the air, and re-packaged in a serial data frame on the other end.

Re-Transmission if checksums fail or no ACKs are received for these over-the-air packets is handled at the ZigBee layer that is equivalent to the Network layer in the OSI model, and in this implementation is completely independent from even the application transport layer. This was taken as a given as part of the decision to use the ZigBee devices and protocol, therefore the following discussion does not take this lower layer into account.

Parsing of the incoming character stream is achieved through the `zb_parse` method which must be called on every character received. It returns a status code to indicate when a complete packet has been received. At this point, globally available variables are filled with the relevant information (op code, device id, data, length) from the packet.

This method is implemented as a finite-state-machine with the state maintained in static local variables.

3.1.7 Application Layer

Figure **??** shows the interaction between the various components for servicing a user request for updating the measurements. One scale unit is shown exemplary.

Master Unit and User Interface

The user interface is implemented as an HTML web page styled with CSS and all client side logic is based on the jQuery Javascript library [**?**] which provides abstraction for sending remote HTTP requests and handling responses to these.

Based on the use cases established in the requirements engineering section, the interface is kept as simple as possible. Four Sensor display elements are included, which show the raw value received from each sensor as well as the conversion to kilograms.

This conversion is performed on the server, as the method used for this is expected to change once specifications for the actual load cells are available.

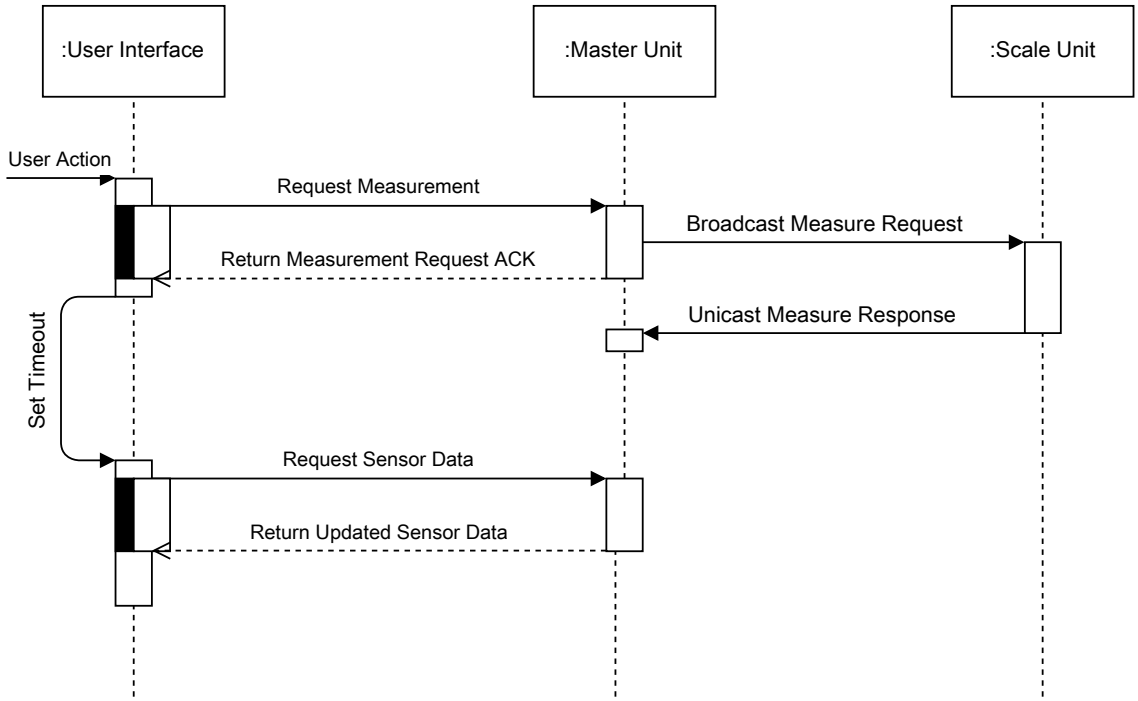

Figure 3.1: Application Level Communications Diagram

Scale Unit

scale unit is an infinite loop: wait for packet, send response, wait for next. ADC set up for continuous DMA conversions.

3.2 Hardware Implementation

3.2.1 Strain Gauge and Wheatstone Bridge

A strain gauge is a component that changes its resistance value based on how stretched or compressed it is, it is therefore good for use in measuring strain which can in turn be used to calculate the weight of an object e.g. a car. When the client UGR commissioned the product they had already selected a strain gauge for use in the system this would be attached to a machanical device creating a load cell, in particular the N11-MA-5-120-11 mild steel foil strain gauge. The relevant specifications of this are that it is 5mm long, it has a gauge factor of 2.1 and has a base resistance of 120.

If a strain gauge changes resistance based on the force exerted on it, one can determin how much force by applying a voltage acrossthe gauge and appreciating the changing behaviour of the circuit when a force is applied to the gauge. To fully explain the relationship between force applied and resistance the datasheet of the gauge was consulted, which revealed the equation K*L/L = R/R (where L is length of the device, K is the gauge factor and R is the resistance).

The datasheet also recommends using a wheatstone bridge configeration on the output of the gauge as seen in fig(**??**). This configuration allows the output voltage to be 0 when there is no force applied and using the right arm as one might use a control group in an experiment it allows for obvious observation of change. This change in resistance is normally tiny most gauges having a maximum L of between 2% and 4%, this being the case the maximum output voltage from the wheatstone bridge is also small and therefore requires an instrumentation amplifier. This instrumentation amplifier helps to bring the voltage difference on the output to a manageable level for the microcontroller so that small changes in resistance can have a bigger and therefore more noticeable change in voltage.

The wheatstone bridge is a simple system relying on the principle of voltage dividers. When looking at fig **??** we can see that when there is no force applied the voltage across the output will be 0 as the voltages at both point A and B will be 5*1000/1120. Now if instead there were a force applied to the strain gauge changing its resistance by 10 then the voltage at point A would be 5*1000/1130 and at point B it would still be at 5*1000/1120 therefore the output voltage would be the difference between these two voltages. The voltage at both points is fed to an instrumentation amplifier, which then magnifies the voltage difference to a more substantial level for the microcontroller to process.

Given that the client is creating their own load cells that have not started the production process yet it is impossible to say what the maximum strain applied to the strain gauge might be. This being the case calculating the required gain of the instrumentation amplifier is also impossible, the decision has therefore been made to model a strain gauge using a simple potentiometer. This allows us to prototype the load cell, enabeling the testing and demonstation of the system, once the load cell is completed it is a simple calculation to find the required gain for the instrumentation amplifier.

Using a potentiometer that has resistance between 0 and 1000 and a resistor in parallel of 430 the model of the load cells's resistance varies between 0 and 430*1000/1430 which equals 300. As we know that the strain gauge is at least 120 the lowest resistance that the potentiometer should be reduced to is 166 as this gives a total 120 when in parallel with 430 static resistor. Using the equations stated previously this means that the output voltage varies between 0V and 0.62V, this model is obviously not perfect as the real strain gauge should only be varying in the milliohm range and the voltage would also therefore be varying in more like millivolts, but it suits the purpose of showing that everything works.

//TODO make this more presentable and shit plus find a way to get the OHM symbol in there as apparently latex doesnt like it which i just type it.

3.2.2 Instrumentation Amplifier

As stated an Instrumentation amplifier is required to increase the range of output voltage from the wheatstone bridge to a more manageable level for the microcontroller. Given that a microcontroller that has a 3 volt ADC is being used, the best thing to do is amplify the largest possible output from the bridge up to 3 volts that way the output should range from 0 to 3 volts.

Given that the maximum voltage on the output of the wheatstone bridge of 0.62V this would make the required gain of the INA 3V/0.62V which is 4.8. So using the equation for gain obtained from the datasheet of the INA 3.2.2 to calculate the resistor required which is Rg = 1 + 2Ra/Av therefore in this case Rg = 1 + 49.4K/4.8 so Rg is 10.3K resistor.

Chapter 4

Evaluation

Things that should be mentioned in here:

Integrate with strain gauges, update conversion method, test accuracy and calibration with known weights.

Scale units should be ported to M0 based system which will lead to smaller PCB. Need to be placed in IP-65 boxes to meet client's requirements.

Power usage at 250mA for scale units is too high. Minimise power usage through power supply redesign. Also, the ZigBee devices on scale units should be configured as end-devices, and transport layer logic should support pin sleep mode to turn the radio off when it is not required.

Test for robustness by modifying units to send invalid packets, or provide artificially generated RF interference.

Incorporate visual feedback in user interface if a unit does not respond to a measurement request.

Run a user test with mechanical engineers who will be using the system, and confirm that it is simple enough for them to set-up and take down for redeployment in different locations. Provide a user manual.

Integrate the Raspberry Pi master unit with a USB wifi dongle, configure hostapd and dnsmasq to create a hot-spot without hte need for an external router.

Analogue Design tested by replacing strain gauge with potentiometer of same resistance range, but it still does not work with the calculated gain resistance for the differential amplifier.

4.0.3 Testing Strategy and Results

4.0.4 Status Report

4.0.5 Future Work

Chapter 5

Conclusion

A great project!

5.1 Contributions

Here we explain that Lewis Carroll wrote chapter 1. John Wayne was out riding his horse every day and didn't do anything. Marilyn Monroe was great at getting the requirements specification and coordinating the writing of the report. Betty Davis did the coding of the kernel of the project, described in Chapter 3. James Dean handled the multimedia content of the project.