Instrumentation

Beau Trepp

Electrical, Electronic and Computer Engineering

University Of Western Australia

A thesis submitted for the degree of

Bachelor of Computer Science/ Bachelor of Electronic Engineering

2011 November

Abstract

The topic of electric vehicles is becoming increasingly popular due to rising fuel costs and growing concern over emissions. Despite this attention, most electric vehicles have little or no telemetry systems, making many aspects of there operation and efficiency a mystery.

The aim of this project was to develop an extend-able system in order to capture various data-points that can be available in a vehicle, as well as an interface to display this data inside the car. The design developed differs from traditional embedded systems by being completely modular. It uses existing network protocols to allow the system to be distributed between various smaller embedded components. This will enable it to be easily extended, should the need for more data-points arise, and allows the use of many smaller systems to be implemented incrementally, rather than one expensive monolithic design.

By exposing and recording more data, deeper analysis can be done on the efficiency of the car, and help justify different technological improvements to the vehicle. The higher granularity of data acquired can also be used to analyse the economy of the vehicle in different conditions, and the affect that different accessories have on the range of the vehicle.

Acknowledgements

During work on this project, I recieved

TODO thank

ZeroMQ guys, Gumstix Guys

Ian Hooper Jonathan something?. Thomas

REV Team

Contents

List of Figures

LIST OF FIGURES

List of Tables

Glossary

BMS Battery Management System

CPU Central Processing Unit

GPS Global Positioning System

TCP Transmission Control Protocol

ZMQ Zero MQ

1

Introduction

1.1 Electric Vehicles

There are many motivating factors behind the development of electric cars. These vehicles utilze new technologies and are represent humanity moving forward in both imagination and respect for the environment.

1.1.1 Pollution

Electric vehicles are advantageous over traditional ICE vehicles as they operate with zero emissions. These vehicles have no exhaust, so therefore have no emissions. While this does not make them completely pollutant free, it does help limit and control the emissions being produced by the act of transport. It is important to remember when discussing electric vehicles that the components must be manufactured using industrial processes and the act of generation electricty. This does not making them truly carbon neutral, but helps limit the sources of pollution. It is much more easier to manage the pollution produced from one power plant, than that from thousands upon millions of vehicles. (1)

1.1.2 Rising Fuel Prices

1.2 Embedded Systems

1.2.1 History

1.2.2 Telemetry

1.2.3 Modularity

2

Aims of the project

2.1 Final Aim

The Ultimate goal of the project is to investigate the viability of distrubted systems in a automotive environment. This will culminate into a completed system, provided data logging functionality and a user interface to view the live data. (2)

2.2 Preliminary aims

Preliminary aims of the project are to.

a minimal embedded systems

2. Develop GPS capability

3. Develop BMS capabilty

this data into a user display

his data to be reviewed later

2. AIMS OF THE PROJECT

3

Literature Review

4

System Design

4.1 GPS Module

4.1.1 Hardware

A vital part of the data logging and user-interface of the software is finding the cars current location. This is done by the use of a off-the self GPS unit. Currently the system is using a Qstar MODEL NEEDED [CITATION NEEDED] usb equipped GPS receiver. This receiver operates at a rate of 10hz [CITATION NEEDED], though it can be set to operate at a slower frequency of 1hz. For the purposes of recording positional data, along with estimating the vehicles current speed, the unit should run as fast as possible. The extra precision is useful for the data-logging aspect, with no negative effects on the user-display aspect.

The GPS device can be enumerated as a standard serial port. This is beneficial as it can be used on any device that has the correct drivers and a available usb port. As it appears as a normal serial port, it can be queried using standard system routines. This allows the program that reads the device to operate any custom knowledge of the device it is connected to, aside from the serial parameters to make the connection.

4.1.2 Drivers

While the Eyebot M6 has hardware usb support, it was not immediately compatible with the GPS sensor. Various versions of usb-serial drivers where tried (see figure X.X) each with their own problems. The main cause of this difficultly was the out-dated Linux kernel being run in the system. This was kernel version number 2.6.17 and was

released in 2006, which is 5 years old as of writing [OSNEWS citation]. This was a major cause of incompatibilities, as the GPS receiver was manufactured a significant time after this kernel was written. The drivers had no clue as to what the usb product keys were, nor the specific quirks that the devices may have had.

The first driver attempted was the generic usb-serial driver, included as a kernel module in 2.6.17. This drivers success would mean that the sensor and program could be easily installed in just about any machine running Linux. The driver would have matured after the 2.6.17 kernel, and newer kernels would have support by default. This is beneficial to the system as it would require the least amount of configuration and setup if the GPS program was set to run in a different machine.

Sadly this driver did not perform correctly with the GPS device. While the driver was able to be loaded into the kernel without any errors, it caused problems when trying to associate with the GPS. The device appeared to use bulk endpoints [CITE/EXPLAIN], which were unsupported by the generic driver. This caused strange symptoms in the operating system. The main symptom of an incorrect driver was the generation of the /dev/ttyUSB0 device. This availability of this device implies that a tty is available to read/write from. Due to the incompatibility of the driver, this serial port would never report any bytes to be read, which is why it is unsuitable for use with this device. Customizing the generic driver to support this device would be unfeasible because it is unlikely that newer versions of this driver would support the device. This leads to the situation that if the GPS program is ported to a different machine, a custom version of the generic usb serial drivers would have to be ported as well.

As the GPS device did not work with the generic serial drivers, alternative drivers were investigated in order to support this device. Experiments indicated that this device was automatically detected and loaded in a newer kernel. This was kernel 2.X.X running on a x86 Intel machine. This functioned correctly and was able to communicate with the GPS device at the full rate of 10hz. The driver used by this kernel was called cdc-acm. Further investigation showed that this driver could be included as a kernel module for the gumstix platform.

This driver was not immediately compatible with the device. This was because the device was manufactured after the kernel [CITE ME]. As such the driver did not recognize the manufacturer ID and product ID of the GPS[see figure X.X]. The driver was then modified to include this information and re-deployed to the eye-bot. This was

successful in creating the virtual serial device inside /dev, and also in allowing data to be read from this device.

While the cdc-acm driver was able to be loaded and functioned, it still contained errors. If the system was under intense CPU load, the program may not run quick enough to remove all the data from the serial port buffer.[CITE TTY/SERIAL BUFFERS]. This would cause the operating system to throttle the port. Examination of the driver source code reveals that new information is dropped while this driver is throttled[cdc-acm/2.6.17-arm]. This is acceptable behaviour in this instance, however this driver had a race condition. If the TTY was throttled under certain conditions, it would be unable to un-throttle later on. This cause the TTY to drain its buffer and never accept any new data from the GPS even if its buffer was empty. [SOURCE CODE ANALYSIS SECTION HERE]. The driver was further modified to include spin-locks, a primitive kernel locking technique, in order to prevent this situation. The driver is now able to run for extended periods of time without locking up, enabling a reliable GPS reporting mechanism to be developed.

4.1.3 Design

Development of the GPS reporting component was done in C. This was chosen as it is a relatively low level language, with wide support. It is simpler to understand than more complicated object oriented style languages. This makes it a good choice for the GPS reporting mechanism, as it only has to do one task. In order to ensure that the code can be easily modified by future programmers, the structure of this program is simple. It runs in only one-thread, aside from the back ground ZeroMQ threads, and thus requires no concurrency management.

4.1.3.1 Initial Design

This first iteration of this code used blocking ports and read a single byte at a time. This was the style used to match existing examples [CITE PREVIOUS THESIS]. This was a functional design however it did lead to some problems. One problem with this approach was excessive throttling. The process would be woken up every time one character could be read, and would only remove one character from the buffer, even if there were hundreds waiting. This is a bad situation, as the process will continually be awoken to do trivial work. This steals cpu-time from another process, and caused the

program to appear sluggish. This design was also in-sufficient as blocked the process while attempting to read from the port. This would cause the program to appear to hang if no data was available. It made it difficult to diagnose errors with this program. Figure X.X shows the process logic of the first iteration of the GPS controller.

4.1.3.2 Final Design

The design was refined in order to support bulk-reads and non-blocking operation. This fixes the two problems with the first approach. Rather than reading one character at a time, the program now reads as many as possible and stores the information into its own circular buffer. This allows the serial port to be purged as quickly as possible. It also has the added feature of allowing the program to decide how to discard messages in the case that it cannot keep up with the GPS.

4.1.3.3 Network Protocol

Table X.X shows the protocol for the GPS message when transmitted over the network. The protocol uses a binary format instead of an ASCII based one. This reduces the space/data transmitted over the link, which helps reduce cost and improve speed. The motivation for ASCII based protocols is that control characters can be used to help synchronize the data. As this design uses ZeroMQ in order to manage the flow of data, such control characters are unnecessary. All values are transmitter in network order, this is big-endian order so that the most significant byte is transmitted first.

4.2 TBS Module

4.2.1 Hardware

The most important external device used in the user interface and data-logging aspect of the software is that of the battery monitoring module. The car has 45 Lithium Ion batteries installed, and it is useful to monitor the charge, current and voltage of the battery cells at all times. The system that the monitoring software runs on is not a highly reliable embedded system. It requires a few minutes to start up, and consumes too much power to leave running all the time. As such a different device is used to track the health and charge of the batteries. This device is a e-xpert pro battery monitor manufactured by TBS electronics. This is a commerical unit which increases

Property	Value
Baud Rate	2400
Data Bits	8
Stop Bits	1
Parity	Even
Flow Control	None

the reliability of the data that it produces. Unlike the eyebot, it is powered as long as the cells in the car remain energized, so it will always log and monitor the health of the batteries.

4.2.2 Expert protocol

The e-xpert device has a set protocol that it uses to communicate with other devices. It uses a RS232 connection over a 9 pin plug. This is a common way of communicating with external modules, and the eyebot has a serial port available to communicate with the e-xpert pro module. The module communicates using asynchronous communication. It automatically sends out updates at a rate of 1hz(**?**). These updates contain all the information that is recorded by the e-expert pro module. As the communication is asynchronous, this will happen automatically, even while the eyebot is not connected. This is not a problem as the e-xpert pro does not expect a response. This mode of operation is referred to as broadcast mode in the e-xpert documentation (**?**).

4.2.2.1 Destination and Start Byte

The message data that the module outputs is transmitted as shown in figure **??**. The first byte in the mesage is the start byte, in order to identify this start byte as the start byte it must be unique and never occur anywhere inside the payload. This is done by reserving the most significant bit (MSB) to be one only if it is a start or ending byte. The documentation refers to this bit as the IDHT (Identify Header Trailer) bit. This does mean that there are only 7 bits available in each byte for transmitting data, but garuentees that the start and end of messages can be synchronized. As the first bit is a one due to the start byte being the header, the value of this byte is greater than 0x80. The rest of the bits in this first byte are the destination address. While communicating

with a PC in broadcast mode, these bits can be ignored (**?**). The module also will not know where it is sending the byte, it is in broadcast mode, so there is no destination address. Thus the destination address bits will be 0, so the first header byte is always received as 0x80.

4.2.2.2 Source

The next byte transmitted is the source address. This byte is not a IDHT byte, so the MSB will always be 0. The device installed in the car, "e-xpert pro" will always set the source address as being 0. Combining this with the IDHT bit results in the second byte always being hex 0.

4.2.2.3 Device ID

The third byte in the message is the device ID. This is a unique number that identifies the type of equipment being used. This number is set by the manufacturer to distinguish different devices it it's product range. For the case of this product, the "e-xpert pro" the device id is 0x22.

4.2.2.4 Message Identifier

The e-xpert pro module transmits a variety of messages, which can be categorized into three groups. These different groups are handshake, commands and data. In broadcast mode, handshake and command messages are not required in order to extract information from the battery monitor. Table **??** shows the hexademical values for different messages.

4.2.2.5 Data

Following the message identifier is the actual payload of the message. This can take on various forms, but in the most simplistic sense is a number spread across a few bytes. Figure **??** shows the data layout for the battery voltage message. For more information about the different values, see appendix **??**. Due to the MSB of each byte being reserved, it is only possible to transmit 7 bits of data inside a byte, any value that requires more than 7 bits to be represented must be transmitted across multiple bytes.

Property	Value
Battery Voltage	0x60
Battery Current	0x61
Amphours	0x62
Charge	0x64
Time Remaining	0x65

COMBINE THEM AS AN EXAMPLE

4.2.2.6 Trailing Byte

The last byte in the message is the "end of transfer" byte (**?**). The purpose of this byte is to signal that the message has been sent. Like the starting byte, the MSB of this byte is set to one. The rest of the bits in this byte are also set to one, to signifiy that this is the end byte, rather than the start byte. The value transmitted is 0xFF, and this is the only location in which 0xFF can appear.

4.2.3 Design

Like the GPS module, the battery monitor module was developed using C. This keeps in line with the design principles of making each component as simple as possible. Figure 4.1 shows the program flow of this daemon.

4.3 Arduino Digital Input Module

4.3.1 Hardware

4.3.2 Drivers

4.3.3 Design

4.4 Accelerometer Module

4.4.1 Hardware

4.4.2 Drivers

4.4.3 Design

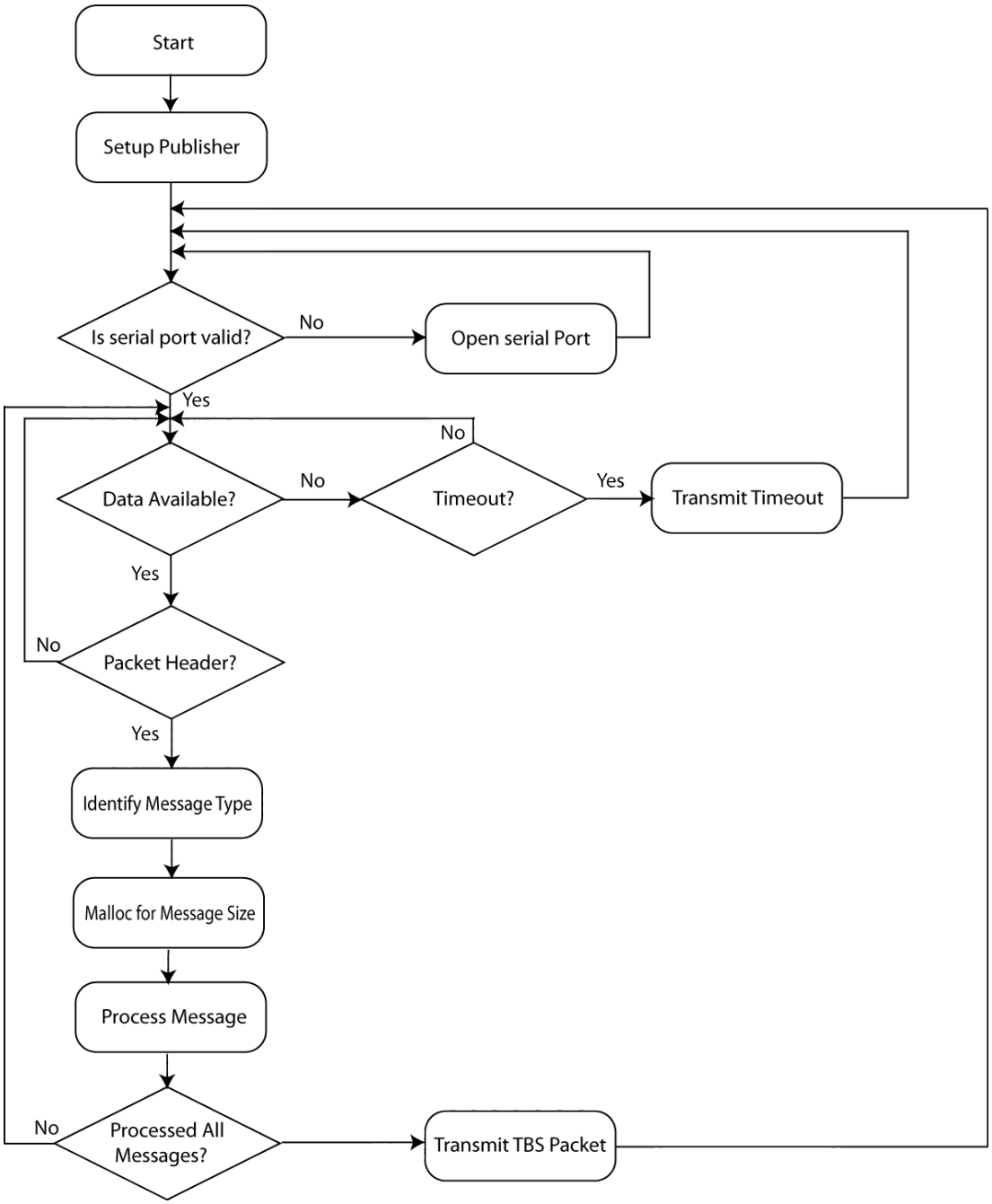

Figure 4.1: Flow chart of the battery monitor daemon -

5

Windowing Toolkit

5.1 Motivation

In order to improve stability of the system, a windowing toolkit was developed to display information to the user. Interactions with the user occur in only a few predefinable ways. The user will either view data that the system has produced, or press buttons on the screen in order transition the display to another screen. The currently developed software (EyeLin) provided very low level access to complete these actions. This allows for greater flexibility in developing using the software, but makes it more confusing to deal with, and contributes to bugs in the software. In order to alleviate this problem, a high-level abstraction was developed on top of the existing software. This abstraction allows the low level functionality, such as interacting with the touchscreen, to be hidden. This simplifies development when layout the user interface. This also allows for code to be written and debugged once, for example the toolkit completely removes interactions with the touchscreen, allowing buttons to be added in a much simpler fashion.

As other requirements of the system required C++ libraries, the windowing toolkit was implemented using C++. This is an object oriented language that provides a few distinct advantages over the C language that is traditionally used on embedded devices such as this. The first advantage is the use of object orientation. When constructing a user interface, it is much easier to understand the different elements of the interface as unique objects. By thinking of a text display as a single object, it becomes more intuitive to manipulate. The other advantage in the use of C++ is that it allows

for objects to inherit from other objects. This use of polymorphism allows elements to implement the functionality they require by inheriting from specifications in the toolkit. The toolkit is able to call these new functions due to use of virtual methods.

The toolkit has been written in a way that the programmer utilizing it does not need to understand anything about threads. While the mechanics inside the toolkit do use threads, the interaction with this threads is completely abstracted away. It is not possible to directly manipulate the underlying threads with the toolkit. This removes any problems with synchronization between threads, as they cannot be manipulated. There are three threads running inside the toolkit. One thread is responsible for manipulating the screen, one thread is responsible for responding too events occurring on the touchscreen and the last thread is responsible for reading and processing the network messages.

The final advantage of developing this toolkit, is that it is not limited to use in this project. As the toolkit is written in a generic way, with the exception of being able to receive the messages transmitted over the network, it is able to be deployed on future projects. This allows future projects to have a rich user interface, while keeping the high-level abstraction in place.

5.2 Elements

The windowing toolkit consists of pre-made classes that the are either used directly, such as the digit display element, or are inherited from, such as the base or runnable classes. Figure 5.1 shows the classes developed in the toolkit.

5.2.1 UIElement

The UIElement is the most basic class definition in the toolkit. It is an abstract class that can never be instantiated. Its purpose is to define methods for interacting with the screen and other objects. It also provides default functionality for most methods, this allows all the classes that inherit from it to function the same way.

5.2.1.1 addChild()

An important definition of the UIElement class, is that it may contain any number of other UIElement classes inside it. To add another UIElement class the method

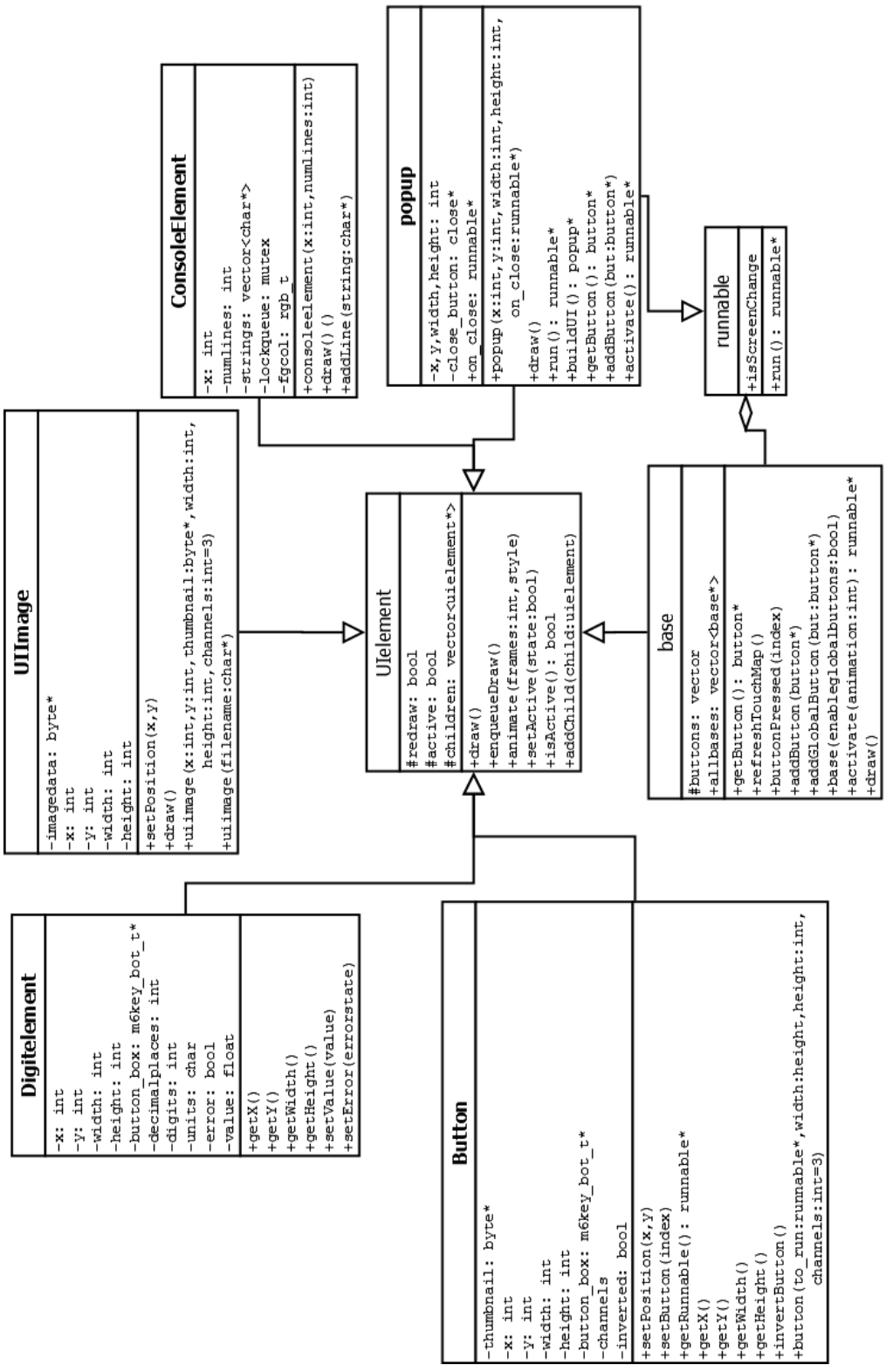

Figure 5.1: UML diagram of the window toolkit -

addChild() is called. This stores the child element inside a C++ vector whose length is only limited by the amount of ram inside the machine. The advantage of this is that actions can be performed on a UIElement and all it's children. If a element needs to be drawn, all it's children will be drawn too or if an element is disabled, all it's children will be disabled too.

5.2.1.2 draw()

The most important method of the UIElement class is the draw method. This method definition is not implemented in the UIElement class, it is defined a virtual abstract method. All classes that can be instantiated must implement this function call. The purpose of this function call is to allow the user to specify the low level commands that are used to display this element. This can include drawing lines, squares, or setting individual pixels. This method should be implemented by the programmer, but should only ever be called by the mechanics of the toolkit. The developed of the user interface should never directly call this method.

5.2.1.3 enqueueDraw()

enqueueDraw() is called whenever the system, or the programmer, wants to trigger a refresh of the screen. This will signal the toolkit that a redraw should be prepared. The purpose of this method is two-fold. Firstly it allows the toolkit to perform optimizations of the draw function in order to maximize speed (see **??**). Secondly it's implementation has O(1) complexity. This means that the call to enqueueDraw() completes in constant time. This is used for performance reasons, as whatever thread has called enqueueDraw(), will not need to wait for the screen to be redrawed, it will return instantly. The draw will be scheduled to occur some time after enqueueDraw() is called. By default this method will also call the enqueueDraw() method of all it's children, allowing entire sections of the display to be redrawn using one function call.

5.2.1.4 animate()

TODO

5.2.1.5 setActive()

A property that is required of any draw-able object inside the toolkit is whether it is current being displayed to the user. This property allows elements to exist in the machines memory, but only be draw if they are current being displayed. To manipulate this property, the method setActive() is called. This method allows the state to be set to either true or false, meaning that the object will be drawn or not drawn respectively. If the active state is false, calls to enqueueDraw() will be processed, but the call to the draw() function will be skipped. Thus individual elements of the display can be hidden at will. This method will also call the setActive() method of all the children of this element. Thus allowing sections of the display to be hidden with one function call.

5.2.1.6 isActive()

This method will return the current state of the element. This is used to check whether the current element is being drawn or not. This method is called internally by the screen drawing mechanics. It can also be used in order to check whether the element is being displayed, and perform different tasks if it is not being displayed.

5.2.2 base

The base element represents the panels or windows that are being displayed to the user. An important property of the base element is that it is defined to occupy the whole screen. This element will draw the entire width of the screen, which will clear any old draws that may still be present. Another extended property of the base element is that it maintains a list of all the other base elements that are present. This is used in order to allow for global navigation buttons. Rather than layout buttons in the same location on every screen, a button can be added as a global button. This will ensure that it appears on all the screens present in the list. This allows the buttons and their location on the screen to be defined once, making the final program more stable and simpler to understand. For further discussion on the button element see **??**. The base class itself is abstract, it cannot be instansiated. There is no way to display a "default" base element. In order to build a panel, the panel must inherit from the base element, and implement at minimum the abstract function getButton().

5.2.2.1 base()

The constructor for this class takes a single argument. This argument is a boolean value indiciating whether this panel will display global buttons or not. By default this option is set to true, though it can be easily overriden when a new class is inheriting from this one. This optional argument allows a panel to forego global buttons. This is used in the case when the developed only wishes to display a simpler panel, or when the current set of global buttons would appear in-appropriately on this panel.

5.2.2.2 draw()

This class implements a rudimentary version of the draw method. This specifies a default display for each fullscreen panel the user will view. The reasons for this functionality are two-fold. Firstly, it allows a common backdrop to be defined for each screen. This keeps a sense of consitency while navigating, as the background will always be similar. It is also useful in speeding up development, as having a default state for a panel is useful for prototyping and implementing new panels. The second reason a default is specified is to remove any elements that were present on the previous screen. Even though buttons and UIelements may not be active anymore, pixels may still be set corresponding to their images. These pixels may have last been refreshed a long time ago. They will remain in the framebuffer until they are overwritten by a new set of image data. As the a base panel is defined as occupying the whole screen, it will replace and UIElements that may already exist on the screen, thus removing them from being visible. As the instance of this base class with be enqueued onto the drawing queue first, all it's child elements will still be drawn correctly.

5.2.2.3 getButton()

In order to transition to a panel, an action must be undertaken by either the user or the system. The most common way of transisitioning would be when the user wants to display a different screen. Typically this would occur by the use of pressing buttons. This is why any base panel must implement the getButton() function. This function returns a button object that contains the image data to display for this button, and the action to undertake when the button is pressed. This action will typically be a call

to the activate() function of the panel, though other actions can be called before the call to activate().

5.2.2.4 addButton()

A common element that will be placed on any panel is a button element. This is the basic way in which the user navigates. A button element itself is a child element of the panel it is contained in, it needs to be drawn when the panel is drawn. A button will also have extra functionality than just being drawn, namely, it can be pressed by the user. As the existing addChild() function only adds the element to the list of child elements to be drawn, an extra method was developed to add buttons to a panel. This method is the addButton method. This method adds the button element to the list of children, but also registers the button element as an interactable object. Internally this is achieved by adding the button to a list that contains only buttons. This list is ordered in the order that the buttons were added to the panel. Maintain this list is important in order to translate the lower level screen interactions into finding which button was pressed see **??**.

5.2.2.5 addGlobalButton()

Mentioned earlier was the use of global buttons, which are buttons that will appear on every class that inherits from base. In order to distinguish between buttons that are added to every base panel and buttons that are added only to the current base panel, the addGlobalButton() function was developed. This method is declared statically and does not need to be called on an instance of a base class, however it does need at least one instance to have been created for it's effects to be observable. In it's simplest form, this method iterates over the list of base elements, and calls the addButton() method on each base using the supplied button argument. This allows the same button to appear on multiple screens, due to the fact that it is the exact same object, it will perform the exact same action and be laid out in the exact same place on each screen.

5.2.2.6 refreshTouchMap()

This method is an internal method to the framework, and should not need to be called by the developer. It is responsible for setting up the touch screen in the lower level

libraries. It takes the list of buttons that has been built by calling addButton() and registers each button and the region the button exists on with the touch drivers. This abstracts any interaction or understanding of how the touch screen mechanics away from the developer. This function will be called whenever activate is called, thus setting up the framework to respond to actions on the buttons that exist on the current panel. It will also clear the previously registered buttons, so they cannot be clicked while the system is displaying the new panel.

5.2.2.7 buttonPressed()

The buttonPressed() method defines what happens when the user presses a button. This method is implemented in the base class definition in a way that it should never need to be overwritten in any classes inheriting from the base class. This method takes the position in the list of buttons that is pressed and performs the actions that occur when the button is pressed. The runnable abstract class allows any action to be coded and run by this method see 5.2.4. This method also performs things like animating the buttons. In order to provide instant feedback to the user pressing a button, the framework will invert the colour of the button. After a short while, the button will turn back to it's previous state, and the buttons action will be performed. This small animation provides instant feedback that the user has pressed the button, and results in a much more enjoyable user experience. Delaying the action also has another advantage. By waiting a short time between running the action, which is usually the display of a different base panel, the framework is able to remove and duplicate keypresses that may occur. This prevents the user from pressing a button to transisition into a panel, and then immediately pressing a button in the new panel which they did not intend to press.

5.2.2.8 activate()

The activate method is defined in the base class. It can be overwritten in classes that inherit from the base class, though any class implementing different functionality should call the base classes activate() method as well. This method's main responsibility is to display or 'activate' the panel that it is called upon, and to translate the touch driver information into button presses. The default functionality of this method, is to call enqueueDraw() first. This will display the current panel on the screen. Next it will call

refreshTouchMap(), to register the active buttons to the framework. Once the panel has been setup, this method will then block and await input from the user via screen events. Thus this method should only ever be called in the thread that is responsible for controlling the button presses. A apparent downside to this is that there is no easy to for the developer to change the active panel inside a different thread, it is certainly possible to change the active panel though it is more difficult than changing in response to user interaction. On the other hand, this limitation is actually advantageous from a user friendliness standpoint. As the screen will only change from different base panels in response to actions performed by the user, it is easier for them to mentally link actions they have performed into re-actions displayed by the device. This specification means that the panels won't transisition by themselves, leading to a much simpler and easier to understand user interface.

This method also allows animations to be specified. By supplying an argument defining the type of screen transition to use, activate method will perform the actions necessary to animate the transisition between the previous panel and this one. These transisitions are implemented inside the screen driver, and include sliding the screen in and out. For performance reasons, the default transistion is an in-place swap, which is the most efficient in respect to CPU time.

When this method finishes it returns the next action to run. This could either be a class that runs the next panel, or some other action to be performed when this panel is not longer being displayed.

5.2.3 Popup

No user interface is complete without popup windows. These are smaller windows that overlay the existing panel. A popup can be positioned anywhere on the screen, and have any width and size, making it a flexible element for displaying information. The popup window will prevent any interaction with the panel they are sitting on top of. This is defined as a modal behaviour (**?**). The reason the popups will only implement modal interfaces is due to the fact that there is no way for the user to manipulate the position of the dialog box. This was done to simplify the user interface, and make it easier to understand. By default a popup will include a close button, so any new popup will always be able to close and return to the normal operation of the panel. Other buttons must be added by the developer.

5.2.3.1 popup()

The constructor to the popup takes arguments in the form of the size and position of the popup, and an action to be performed when the popup closes. The last argument is a runnable object that will be called when the popup is closed. This allows actions to be scheduled in between the popup dissapearing, and the previous panel taking focus. This flexibility can be used to trigger screen refresh events, or to flush a file to the disk, or even make another panel take focus. The popup class is a child of the base class, and the runnable class.

5.2.3.2 buildUI()

Due to the way C++ calls constructors, a method buildUI has been created in order to over-ride the default button layout. Any class that inherits from another class will call the superclasses constructor. This is a problem if the child class does not wish to use the default exit button provided. To overcome this, the elements inside the popup are laid out when buildUI() is called. This method is called after the popup has been fully constructed. Due to polymorphism, the method will always call the implementation of the child class. This allows the child class to completely override the default layout.

The method has returns a pointer to the class it was called on. This is done to allow the object to be constructed and the layout generated in one line of code, and still return a pointer to the popup object.

5.2.3.3 run()

The popup class implements the runnable abstract class by default. This allows the popup to be passed as a runnable object. This run action performs the neccesary steps in order to activate the popup and display it on the screen. The advantage of this is that a popup can be passed as an argument to the button class, meaning that when the button is pressed the popup will be shown. This results in very simple and easy to read code.

5.2.3.4 getButton()

This method exists because the popup inherits from the base class. It allows the popup to specify a button element that should be used to display the popup. This button

element will usually contain a runnable class that activates the popup. This method can return null if there is no button for this popup.

5.2.3.5 activate()

This method is inherited from the base class. See section 5.2.2.8 for more information.

5.2.4 runnable

The runnable class is an abstract class which is used to allow the developer using the toolkit to run actions inside the toolkit. By using polymorphism, the toolkit is able to call methods on classes which were not programmed or compiled with the toolkit. This runnable aspect is used in order to 'run' various aspects of the user interface. For instance, the base class's activate function must return a pointer to a runnable class. This is used to define what happens when the panel is no longer active. Runnable classes are also used extensively by the button system. A button must contain a runnable class. This class will be run when the button is pressed. Thus allowing any code to run whenever a button is pressed by the user.

5.2.4.1 isScreenChange

A property used in the runnable class is whether this action is triggering a screen change or not. This property is set to true by default. If this property is to, the previous base panel that called this action will have it's active property set to false. This suppresses the last screen from being changed, so the runnable action does not need to know where it was called from. If this property is set to false, the previous panel will not be flagged as inactive. This is used whenever an action is updating a element on the user interface, and does only wants to trigger a redraw of the element that changed. The previous panel will remain active, and will resume listening to touch events when the runnable action is finished.

5.2.4.2 run()

This is the abstract method that must be implemented in any class inheriting from the runnable class. It is called by the framework in order to run developer specified actions. Any code inside this method is run by the thread that controls the button interactions.

This means that whatever is inside this method will block the touchscreen until it has completed, though this method is able to interact with the touch screen and read of events itself. This is a useful property, as while the system is performing some action, the user will be prevented from transitioning the screen, or running another action.

5.2.5 Button

The button class is a fully implemented class in the framework. The developer using the framework does not need to inherit from this class in order to use it's functionality, though they are able to if they desire more control over the way buttons behave. This element is a object representation of a button, and contains all the information required to use the button in the framework. It stores information pertaining to the position, size, state, image and action to be performed when this button is pressed. This allows the framework to lay the button out and display it to the user, and perform actions when the button is pressed. The thumbnail is able to exist as a simple RGB based image, or an alpha enabled RGBA image see **??**. The information stored in this object is also used to correctly set up the touch drivers in order to correctly identify when the button is pressed.

5.2.5.1 button()

The constructor of this button requires two parts of information, the action to run when the button is pressed and the image data that will represent this button on the screen. The runnable action must be a class that inherits from the runnable class see 5.2.4. The image data must be an array of bytes formatted as either RGB or RGBA pixels. As the image data is a contigous array, the function also requires the knowledge of it's width and height. In order to differentiate the difference between RGB and RGBA, the constructor also needs to know how many channels are in the image data. 3 defines a RGB image and 4 defines a RGBA image. In order to neaten code, if the channels argument is ommitted the system will assume a RGB image is supplied. This will be programmaticallly safe, as if an RGBA image is supplied but the channels are 3, the framework will not read outside the array of the image. It will however display a corrupted looking image, so it is recommended to explicitly pass the number of channels inside the image. Note that the size of the image is the size of the area that responds to the users touch. If the areas were different sizes, image scaling would have to be

performed, which will lead to high levels of distortion on the very low resolution images displayed.

5.2.5.2 setPosition()

This method allows the developer to move the location of the button. It takes two arguments, the x and y co-ordinates where the top left corner of t he button will sit. The co-ordinate system used is that of pixels available on the screen. By default the button will be placed at (0,0) which places the top left corner of the button at the top left corner of the screen.

5.2.5.3 getX()

This method returns the current x position of the top left corner of the button. It is mainly used to find out the location of the button on the screen, in order to avoid laying another button over the same space.

5.2.5.4 getY()

Performs the same function as getX(), except it returns the y co-ordinate in pixels.

5.2.5.5 getWidth()

Returns the width of the image in pixels. This is used to find out how wide the image is on the screen.

5.2.5.6 getHeight()

Performs the same function as getWidth(), except returns the height instead of the width.

5.2.5.7 invertButton()

This method inverts all the colours on the button. This is used in the base class when a button is pressed (see 5.2.2.7). This method marks the button as inverted, and the next call to enqueueDraw() will draw the button with inverted colours. Calling this function a second time will restore the state to it's original condition.

5.2.5.8 getRunnable()

When the button object is created, it requires a runnable object to perform when the button is pressed. This method returns a pointer to that object so that it may be run, or inspected by the framework.

5.2.6 Digitelement

The digit element is a class that is repsonsible for displaying numbers to the screen. It is fully implemented in the framework, and does not need any methods to be implemented to be use-able. It renders white colour numbers from zero to nine, with a transparent surrounding so it is able to be overlaid on any image. Internally the digitelement is aware of the number that it has currently displaying and the new value it needs to display. It maintains this awareness in order to change only the information that differs from that present on the screen. This is done in order to maximize performance of the system. If a number, say zero, is displayed the system will not draw a zero over the top of it, it will skip to drawing the next digit.

One caveat of this element is that it records the current background when it is initialized. This snapshot is used to clear out the previous image data if it differs from the desired data. This is done to prevent the digits from being drawn on-top of each other continuously. This does make the digit element currently unsuitable for display over any moving image data, though moving image data is not used extensively in the interface.

The digitelement is highly configurable, allowing the developer to specify the position, size, integer places, decimal places and units to be displayed after the digit element. Common engineering units are defined in the image element, such as km/h, V and A. This allows relevant information to be laid out quickly. The class takes double length floating point values as an input, but will display the number according to the parameters specified during construction. This ensures the element has consistent sizing, as a change in the magnitude of the number will not cause it to shift about on the display. If decimal places is less than 1, the element will round the value to the nearest integer.

Finally the digitelement can also represent an error state. This is typically used when no information has been recieved, so the number to be displayed is undefined.

In this situation displaying any number would be incorrect. As such the element will display a horizontal dash. This indicates that an error has been triggered, and their is no valid data to display.

5.2.6.1 getX()

This method returns the current x position of the top left corner of the button. It is mainly used to find out the location of the button on the screen.

5.2.6.2 getY()

Performs the same function as getX(), except it returns the y co-ordinate in pixels.

5.2.6.3 getWidth()

Returns the width of the digitelement in pixels. This will be the width of a individual digit element multiplied by the number of digits that are being displayed.

5.2.6.4 getHeight()

Returns the height of the digitelement. This is the same as the height of any individual number in the element, as they are all rendered at the same size.

5.2.6.5 setValue()

This function is used to set the value to be displayed on the digitelement. The new value will be displayed on the screen after a call to enqueueDraw().

5.2.6.6 setError()

This function is used to set or reset the error state. It will cause the digitelement to display a horizontal line instead of numbers.

5.2.7 Console Element

The console element is a generic text element that is used to display full width text to the screen. The main purpose of this element is to provide an easy way to see debugging information on the screen itself, without requiring any remote connections

to the controller. This element takes any string and converts it to be correctly displayed on the screen. It is designed to be used with data that is dynamically changing.

The console element maintains an internal queue of the elements it is currently displayed. This allows a buffer to exist that will display the most recently added strings. The older strings will be removed as newer strings are added. To avoid any conflicts with memory, a string is copied into the internal buffer when it is added. This prevents any other section of the program from freeing or overwriting the memory where the string was stored. When a string is no longer being displayed, it is removed from the queue, and the memory used to store it is cleared.

As the queue has the possibility of being accessed from multiple threads, it must have security in place to ensure the queue does not become corrupted. If any strings memeory is cleared at the same it is being drawn, the system will have undefined behaviour. In order to prevent this, a mechanism calling locking is used. When a thread needs to manipulate or read the queue, it locks the queue. Any other thread that tries to lock the queue at this point will be forced to wait. When the thread is finally finished with the queue, it will unlock the queue, allowing the other thread to perform an operation on it. This will cause the second thread to stop until the first thread is finished. This simple method of synchronization will prevent the queue from becoming corrupt, and ensures the stability of the console element.

The console element is concerned with displaying the newest values first. In order to best illustrate this, the new values appear at the bottom of the screen, and move upwards until they are no longer displayed. This mimics the display of consoles in normal computers.

A problem that can occur with the display of the strings is the string length. The display only has a fixed width, and the desired string may be longer than the screen width. This could cause some of the data to not appear on the screen. In order to alleviate this problem, the string is split into multiple strings, each with a maximum length that is equal to the width of the screen. This allows the string to be rendered on multiple lines. When a string spills over to multiple lines, it means less space for the older lines that are to be rendered. The element takes into account multi-line strings, and may not always display the desired amount of strings if some strings are too long. This is done to enable all of the information of the newer strings to be read.

A problem with multiple-lined strings is that it can be hard to differentiated between the different lines. To solve this the console element uses alternating colours when displaying different lines. This provides strong visual indicators as to which lines are grouped together as one string, making it easy to identify what the element is trying to say.

5.2.7.1 consolelement()

The constructor to the consoleelement takes two arguments. The first is an offset. This offset specifies how many lines at the top of the screen the element should not print in. This is to prevent the element from drawing text over buttons or other elements that may appear at the top of the screen. The second element is the maximum number of lines that should be drawn. This argument is used to prevent the element from drawing over buttons or other elements at the bottom of the screen.

5.2.7.2 addLine()

This method adds a line to the queue. It will lock the queue, add the line, free any lines that are now too old and then finally unlock the queue. Note that this makes it's own copy of the supplied string, so it's argument may be destroyed after this method is called.

5.2.7.3 draw()

This method is implemented in the console element and does not need to be overridden. It will lock the queue, split any multiple line strings, draw the lines and then unlock the queue. This method is called by the framework, if the developed wants to refresh the console element, enqueueDraw() should be called instead.

5.3 Subscriber

5.4 Button Loop

5.5 Message Queue

5.6 Redraw Performance

Due to the limited resources of the system, care must be taken when attempting to update the screen. As as windowing system was already being developed, it was desirable to have this system absract these actions away from the programmer. By using the toolkit the programmer can trigger when an element should be draw, and specify the drawing code to draw the element. The absraction means that the developed code will never be called directly by the programmer, it is all taken care of by the underlying mechanisms.

5.6.1 Refresh on Arrival

Whenever a new message is recieved by the user interface, it would seem appropriate to process that message instantaneously. Figure 5.2 shows the flow of this methodology. A message is recieved, then the system processes the message, redraws the screen, and then starts over again. While this is a working solution, it does present problems in regards to performance and the overall usability of the system.

The performance penalty in such a design is not immediately obvious. Data that is recieved should be displayed instantanouelsy. Inspecting Figure 5.2 further does help highlight the problem that occurs in this situation. While the system is processing or displaying the message, it is unable to process anymore messages. This can be a problem when the source of the messages is generating messages faster than the device can process.

5.6.1.1 Message speed greater than redraw rate

In situations such as the BMS module, the program was able to sufficiently cope with the input. This lead to all the messages being instantly removed from the network layer when they arrrived. Thus the program was always in the state of "waiting for a message". However another important sensor caused problems with this method.

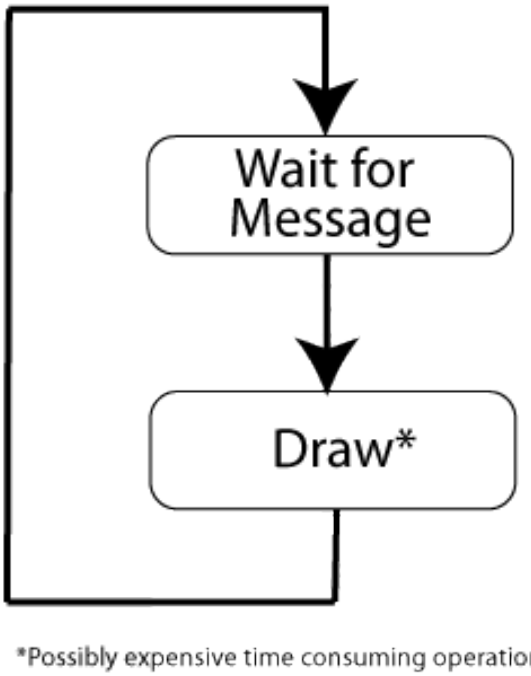

Figure 5.2: Processing message flow chart - This figure shows the naive approach to processing and displaying recieved message data

This sensor was the one designed to read the GPS signals. Messages containing GPS information where generated at a rate of 10 messages per second, or 10hz. This speed led to the situation where when a new message arrived, the process was still in the "Draw" state. This would cause the message to be delayed on the network layer.

The rate of input regarding the GPS module was constant, these messages would continually build up on the network layer. Any new messages that were transmitted would be dropped when they were attempted to be sent. While this is acceptable, eventually space would be cleared for new messages, it would lead to alot of new messages to be dropped. The other issue that occured here, was the new messages appeared at the back of the "Queue". Until all the older messages were processed, the new ones would not be seen. This is expected from how ZeroMQ functions (**?**)zeroMQ_internals. However, this meant that there was a significant delay until new data was seen.

The delay that occured was proportional to the size of the ZeroMQ Message Queue (**?**)zeroMQ_internals. This delay could be up to a couple of seconds, which is unacceptable for live feedback to the driver. Reducing the size of the Message Queue helped alleviate the problem, however the screen would still attempt to redraw as fast as the messages where recieved. This also led to the interface portion of the system hogging the CPU. Thus an alternate method of dealing with screen updates was developed.

5.6.2 Add to Queue

5.6.3 Redrawing the Screen

The previous section discussed the problems that occured with allowing the screen to update whenever a message was recieved. The biggest performance penalty that occured was not from the actual processing of the data, but from having to display it on the screen. All of the processing is relatively trivial computation wise. It is the transfer of variables into various memory locations so they can be accessed when the screen is displayed. Copying a variable itself is trivial, however processing that variable for display on screen is incredibly intense.

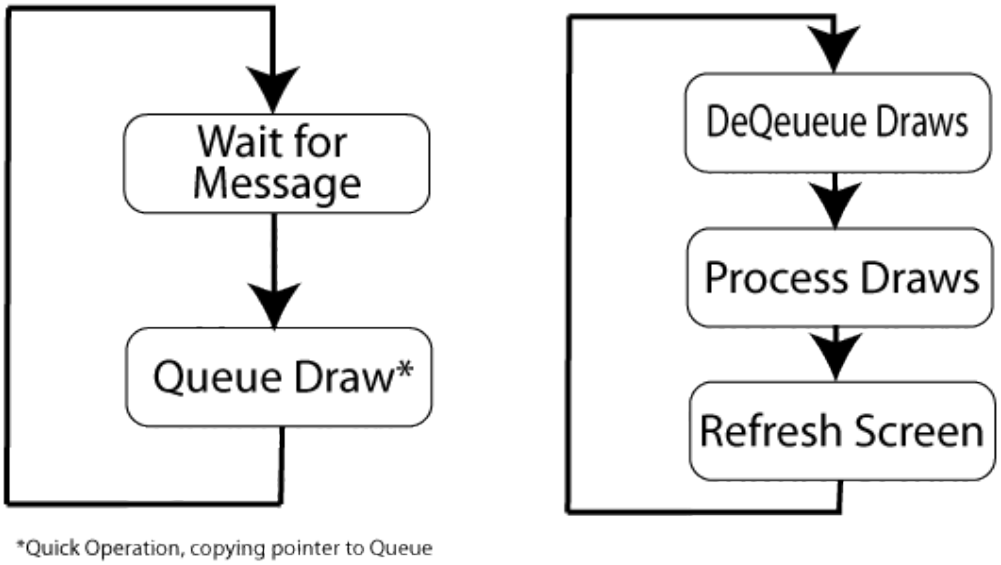

Figure 5.3: Appending to Queue - This figure shows the flow of appending screen refreshes to a queue

5.6.4 Redraw rate

Inspection of the EyeLin library source code showed that the library was using a simple framebuffer in order to interact with the screen (**?**)frame_buffer. This framebuffer method stored the entire screen state as an 24bit image. By default, whenever one pixel was changed in this image, all the data was copied to the framebuffer again. This was incredibly wasteful, and contributed to the large delays in redrawing the screen. Many items on the screen need to be redrawn together, for instance, a digit display has three or more digits that may change from it's last appearance. By default the library would redraw three times if the number changed by a large amount. This caused alot of performance issues in previous projects using these libraries see

5.6.5 Batch Redraw

In order to offset this problem the way in which the queue processed draws was modified. The thread that processed the queue would attempting to dequeue as much items as possible and process them as a batch. This gives more performance than attempting to redraw the screen for every change.

5.6.5.1 Maximum batch size

Attempting to remove as many items as possible is a problem in a multi-threaded system. Consider the case where a thread is adding elements, the producer thread, at the same time the drawing thread is removing them, the consumer thread. If the producer thread is operating faster than the consumer thread, the consumer thread will always be removing elements from the queue. Thus the consumer thread has a maximum amount of elements it will process at a time. This garuntees that the consumer thread will always refresh the screen.

5.6.5.2 Incomplete batch

Another issue that can occur with processing drawing events in a batch format is that no new draw events may be generated. Consider that the thread is attempting to remove a certain number of redraw events, however there may only be half present inside the queue. If , for whatever reason, no more redraw events are added, the queue will wait forever attempting to remove them. This is undesirable, as some transitions may only trigger a few redraw events and do nothing more. A good example of this is a static page, like the sponsor page. It only has a few elements, the buttons and the sponsor logos, and is not dynamically updated in response to any data. If the thread was waiting for more draw events, they would never be recieved. The solution to this problem is to continue on with the drawing actions if the queue is ever empty. This prevents the thread for waiting for more events, and helps garuntee the constant refreshing of the screen.

6

Interface

6.1 Layout

An important aspect of user interfaces is that they must be visually impressive [GO CITE?]. This keeps the user interested in the product, and helps new users want to learn how to operate the device. The work done in the windowing toolkit allowed a much more visually impressive layout to be developed. This was achieved through the use of transparencies in order to overlay different elements on top of each other.

6.1.1 Background

As the device does not have enough power to dynamically render any sort of complex images or motions, a pre-rendered static background was used. This background was designed on another pc, and converted for use on the embedded system. It consists of two main regions. The first area is the main display region. This section is the largest of the space and exists so the current data being displayed can be laid over it. The second region is the navigation bar region. This area is at the bottom of the image, it uses a more uniform texture in order to contrast the main data display area above it. The background itself is rendered in black and white. The black and white styling allows important information to contrast against it easily. This helps highlight the information and user interface actions present to the user, and avoids a simple single-colour background on the user interface.

6.1.2 Navigation Model

Uses a tree model that can throw back any other blah blah.. FIXME

6.2 Overview Panel

To facilitate easy navigation a panel was designed that shows all the different panels available. This panel is shown in figure 6.1. It contains 8 different aspects of the program, with room available for 12. This will allow the system to be extended in the future.

Figure 6.1: Panel showing other panels - Shortcuts to different aspects of the program. (Battery, Maps, Trip Meter, Accelerometer, Arduino, Savings, About, Options)

6.3 Battery

The battery panel is the main display panel used in the user interface. It is the first panel displayed to the user when the system turns on. It displays five important pieces of information to the user operating the vehicle. This panel listens to both the battery (TBS) and gps messages. It requires the battery messages to display the battery voltage, current and charge to the user. These are displayed using the digitelements mentioned

in **??**. It also uses a custom charge element class that exists outside of the framework to draw a battery on the center of the screen. This green bar on this battery will decrease in proportion to the charge remaining in the car, providing a quick visual indicator for how much charge is still stored.

This panel also displays the speed in km/h, which is why it has to register to the gps module. The speed is included so the user does not have to interact with the screen while driving. This panels main purpose was to provide a quick overview to data points that are immediately of concern to the driver.

Displayed in the top left corner is a rudimentary calculation of how much distance is remaining in the car. Range tests conducted by the REV team indicated that this distance was 80km from a full charge. To provide some security while driving, the interface assumes that the max distance the car will travel on a full charge is 70km. It uses this to calculate the distance remaining according to the formula 6.1

$$70 = 10 \tag{6.1}$$

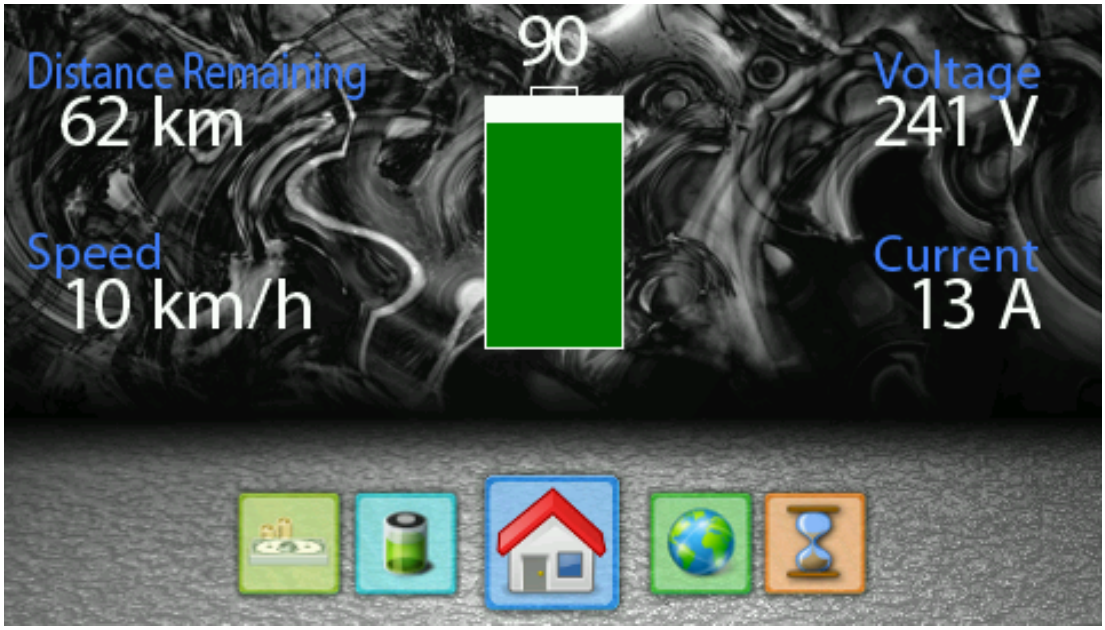

Figure 6.2: Battery state panel - Screenshot of the battery state panel, showing the voltage, current, charge, speed (via gps) and distance remaining

6.4 Maps

A common, yet useful driver aid is displaying the map of the current location. This is what the maps panel does. It listens to the GPS messages to determine the position of the car. A screen-shot of this panel is shown in Figure 6.3. This panel uses the full area to display the current location of the vehicle on the screen. The actual position of the vehicle is centered on the middle of the screen, allowing the driver to view streets and landmarks relative to his current position. The panel features several levels of zoom, controlled by the bar on the top right corner on the screen. The plus button will zoom in, and move the slider to the right, while the minus button will zoom out, and move the slider to the left. The system supports various levels of zoom, being able to display a few buildings relative to the car or being able to display the surrounding suburbs.

Figure 6.3: Map display panel - Screen-shot of the map display panel, showing the map of the current location and the map controls visible

The controls visible on Figure 6.3 are not always required by the operator. They can obscure parts of the map in which the driver may be interested in. This can be a problem as the driver should not be expected to adjust the position of the car, in order to see a part of the map that is hidden by the UI controls. In order to prevent this problem, the controls can be hidden by tapping any area of the screen that is not

occupied by a button. Doing this will hide all the navigation elements and just display the map as shown in 6.4.

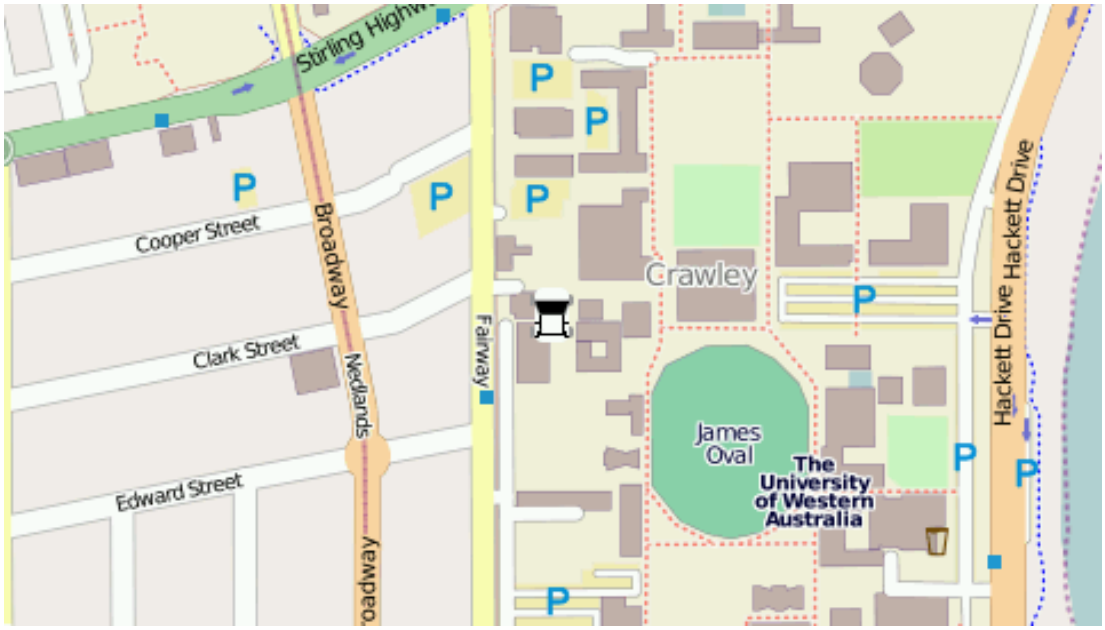

Figure 6.4: Map display panel with hidden controls - Screen-shot of the map display panel, showing the map of the current location and the map controls hidden

.

6.4.1 Map Data

In order to display the map images, the map data must first be obtained and stored. There are many possible methods for doing this, ranging from creating the maps as needed, or downloading them from external services and storing them in a cache. The relatively low CPU power of the eye-bot m6 makes rendering the maps on-the-fly a undesirable prospect. Open source map data for the entire planet results in a file that is 18GB in size (**?**). This file is much too large to store locally, and this file is actually storing a compressed version of the data. Even if it were possible for the eye-bot to store this file, via the use of external storage, it would be a large strain on the CPU to convert the street level data into viewable maps. It would also require many other pieces of software to be installed on the device, making it much more complicated to manage.

Another possibility is to use an existing Internet based map server and download the map imagery as needed. The has several downsides. Foremost it requires an Internet connection whenever to display the maps. The system does have a 3G connection installed, but this cannot be considered a dependable communication channel. It is highly likely to drop out, and is limited in coverage to the areas in which it has reception. Even without these issues, most map-servers do not allow you to download maps in bulk, as this violates their usage policies (**?**). This makes this method undesirable as it is not suit-able for downloading maps in bulk, or as needed. Attempting to pre-download all these maps using a non-3g link would result in a violation of the usage policy.

The method chosen to obtain the map data was to pre-create create the map data using a more powerful machine. A map-server was setup and loaded with all the street data for the oceania region. For more information on the map-server setup see appendix **??**. This method overcomes the problems of the previously mentioned methods. All the processing is done on the much faster machine in advance. The area processed in advance is defined by the properties in table 6.1. This area is depicted by the image shown in Figure 6.5. The expanse of these maps covers all of metropolitan Perth. In future more maps can easily be processed, however this will result in more storage space being required. The current settings are a good balance between storage and expanse of data. This is because the map data will be less useful outside of the city, and the car is typically not driven any further than the pre-rendered maps. The current space usage of the map data is given in table 6.2. As this is much much larger than the internal 16MB of flash storage the eyebot has, it must be stored on external storage. Luckily, storage is now cheap, and 8GB usb thumb drive has enough performance and space to store and serve the image data.

Property	Value
Minimum Zoom Level	11
Maximum Zoom Level	18
Top Left	115.687,-31.71
Bottom Right	116.508,-32.253

Table 6.1: The properties of the pre-rendered map data

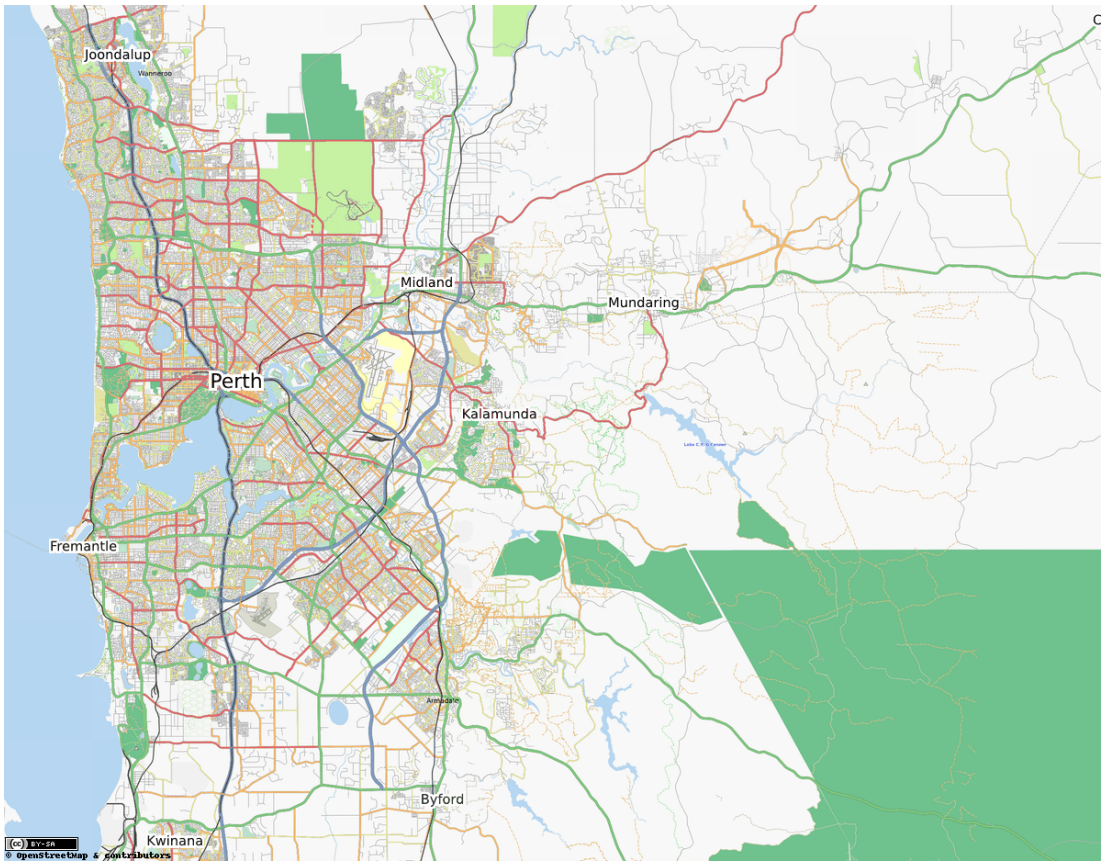

Figure 6.5: Pre-rendered map size - This figure shows the area that has been pre-rendered for use in the map panel display

Property	Value
Zoom Levels	7
Number of Files	435,456
Total Size	580.2 MB
File Resolution	256*256 pixels
File Format	Palette PNG
Time taken to process	Approximately 1 Day

Table 6.2: File statistics of map data

6.4.2 Tiling

As mentioned previously, the map data is much larger than the internal storage of the eye-bot. It is also much much larger than the operating systems 64MB of ram. This makes it impossible for the entire map data to be loaded inside any program to be displayed to the user. In order to overcome this limitation, the map panel only loads a small subset of the map data at a time, and displays it using a method called tiling. Rather than have one giant map that displays all of the information, and sliding this around to view the correct part, this method divides the map into many smaller maps. These smaller maps are referred to as tiles. Each tile consists of a 256 pixels wide by 256 pixels high image. These images are combined in order to display the current location, as seen in figure 6.6. By loading the adjacent tiles in all directions the screen will always have data to display.

6.4.2.1 Converting GPS Co-ordinates

The GPS outputs the current position in latitude/longitude format. This format must be converted so that the correct tiles may be loaded. To do this the GPS co-ordinates are converted to grid based co-ordinates using a mercator projection (**?**). The equations to convert to this format are given by 6.2. It is possible to convert from a grid co-ordinate back to a GPS co-ordinate using the equations given by **??**. This of course will only be able to return the top left corner of the tile in GPS co-ordinates, and not the original position.

$$70 = 10 \tag{6.2}$$

$$70 = 10 \tag{6.3}$$

With the GPS co-ordinate converted it is possible to locate the tile to display to the user. Rather than utilize lookup files to locate the tile, a specific folder structure is used to instantaneously load the file. This is done by storing the files in the format */zoom/tilex/tiley.png*. This allows the system to directly open the file and display it. Thus even if the entire world was loaded in the maps folder, for many different zoom levels, the display time of the current location would not change.

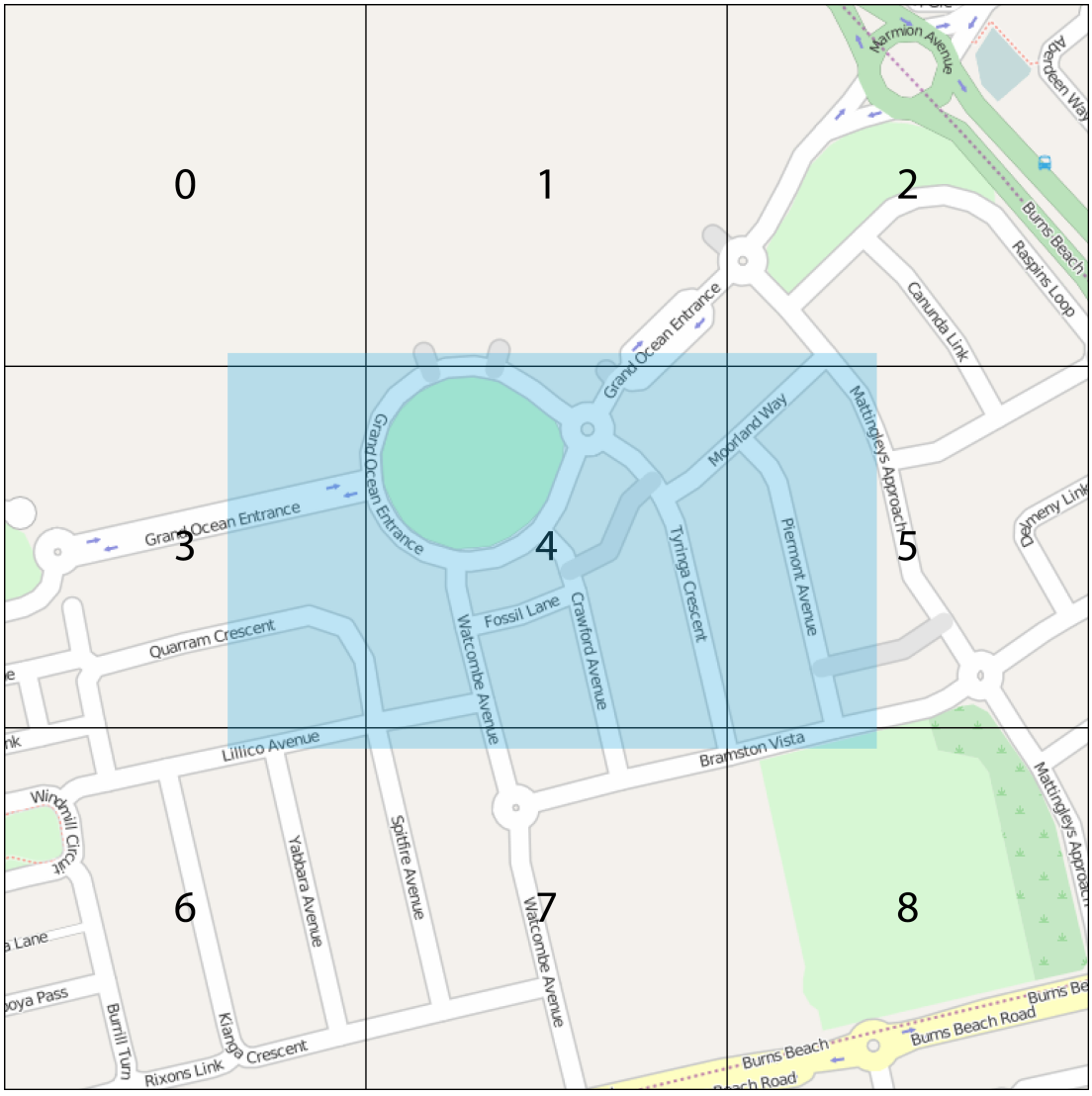

Figure 6.6: Tiling Maps - This figure shows the tiles that are loaded, labeled 0-8, and the area of the screen that is able to view them.

6.4.3 Palleted File Format

The amount of storage required for all these tiles in an uncompressed format is shown in equation 6.4 using the data obtained in table 6.2. This amounts to approximately 81 gigabytes of information at all the various zoom levels. This size is much too large to store and a small usb drive. It would be possible to store this data on a mechanical disk, but this is much more costly than a cheaper flash based drive, and would require much more power to run. In order to solve this problem, a compressed file format is used. The PNG file format results in a much more smaller filesize per tile. The average filesize of the final compresed files is approximately 1.3 kB. This is a large contrast to the size of one file in an uncompressed format, being 192 kB. PNG uses various compression techniques to greatly reduce the filesize of the tiles.

$$\begin{aligned} TotalSpaceRequired &= \frac{\text{height} * \text{width} * \text{image} * \text{number of tiles}}{\text{bytes in a megabyte}} \\ &= \frac{256 * 256 * 3 * 435456}{1048676} \\ &= 81648\ MB \end{aligned} \tag{6.4}$$

The main technique that is utilized in this system is the use of pallettes. Palleting restricts the colour space in the image, in order to save space. The map images do not utilized the full 24bit colour space that the screen is capable of displaying as the maps need to be visually clean, they only use a handful of colours to represent the area. Palleting works by defining the sets of colours that will be used in the image beforehand, or the pallette of this image. Consider an image with only two colours. If each colour in the pallette is given a value, each pixel in the image can be represented by that value. Hence each pixel can be represented by 1 bit, rather than 3 bytes (24 bits). This is a simple example, but it highlights how pallettes can be used to save space. The downside to using this technique, is that it is slightly slower than using raw image data. As the pallette is represented in a different format, it must be loaded and converted to the unprocessed data when needed. This takes some time compared to just reading the raw data. This trade-off is worthwhile however, as the space saving more than makes up for the extra CPU time to load the images. The method also has the advantage that the tile images can be easily read or modified by consumer

software on other machines, PNG files are very common and can be natively viewed on all full-blown operating systems.

6.4.4 Sliding Maps

The last feature that the map panel implements is centering the vehicles position on the screen. This means that the screen will slide around in response to changes in the vehicles position. It does this by using a modified version of the equations in 6.2. These modified equation works takes the same inputs, but returns the position of the car inside the grid co-ordinate as a floating point number. This return result has the range $0 \leq result < 1$. A return value of 0 indicates that the top right corner of the image should be placed in the center of the screen. A return value of 1 is never possible, as this would be 0 on a different tile. This allows the tiles to be moved around as the car moves, providing sleek graphical feedback to the operator of the vehicle.

6.5 Trip Meter

A useful driver aid that is common on vehicles is that of a trip meter. Traditionally this component records the distance the car has traveled since the trip meter was set. This functionality is usually a result of the simple systems in place, and is tied to the revolutions of the wheels on the vehicle. As this system has more hardware at it's disposal, the trip meter can implement more functionality than a standard trip meter, making it much more useful in examining the performance of the car. Figure 6.7 shows the trip meter panel being displayed on the screen. An important note of this panel is that all calculations are done whether the panel is being displayed or not.

The first unique point of this trip meter, is that it contains two independent meters. This is useful as it allows the driver to evaluate the statistics of two overlapping trips. One trip meter can be used to record the distance traveled since the car was last charged, while the other can be used to record the distance traveled in the last week or month. This independence allows the operator to decide how best to use the trip meter data, resulting in a high level of flexibility. In figure 6.7 the trip meters are located on the left, sitting above each other.

Each trip meter records the distance traveled, the time elapsed since the meter was started, the time the car has been moving since the meter was started and calculations

Figure 6.7: Trip Meter Panel - This figure shows two independent trip meters and the best record speed data

based on the elapsed and moving time. These statistics are displayed live to the user, but are not logged, as the logging functionality is taken car of by a different component in the system. The trip meter panel also displays the current moving speed in the top right. Below the moving speed are the best run records in seconds. This allows the driver to have quick feedback as to how the car is performing, without having to do lots of processing on logged data.

6.5.1 Distance Driven

The most important part of a trip meter, is the distance that the meter has recorded. This is shown on figure 6.7 at the bottom of each trip meter. The meter stores the distance driven internally as a double length floating point number, but displays it on the screen as a rounded integer. This is done in order improve precision for later calculations as other values will depend on the distance that has been driven. Equation 6.5 shows the the distance is calculated based on the GPS position.

$$\text{Distance} = \text{Distance}_{\text{lastrun}} + \text{Speed}(\text{Time}_{\text{now}} - \text{Time}_{\text{lastrun}}) \tag{6.5}$$

The method for working out is a continuous function that is based on the last known distance the car has traveled. This method was chosen as it does not require any information other than the last time the formula was run, and the last distance calculated. This also makes the trip meter flexible in that the distance calculation does not need to be processed at exact intervals. If messages are dropped for whatever reason, the calculation will still take place, though it will not be as accurate as it could be. The calculation will be able to cope with fluctuations in the message timing, and can adapt to the speed of the GPS being used.

The downside of this method, is that big changes in time can cause problems with the calculation. If the signal drops out for an extended period of time, such as going through a tunnel, the calculation in 6.5 would have a big margin for error. In order to prevent this, the trip meter will ignore large time differences. If two calculations are over 10 seconds apart, the result will not be trusted, and not be used in the calculation. This prevents GPS signal loss from having an adverse effect on the trip meter calculations, but does impose a limit on the trip meter.

The limitation of this method of calculating the distance driven is that it relies on the GPS messages being sent to it. If the GPS signal is lost, the distance driven will not be increased. This is a limitation imposed by the use of the GPS sensor, and cannot be avoided, as attempting to guess the distance driven while the GPS signal has been lost has a very high probability of being incorrect.

6.5.2 Time Elapsed

Another variable displayed on the trip panel is the time elapsed. This is simply the time elapsed since the trip meter was last reset. This time increments even while the GPS signal has been lost, performing a stop-watch like action on the trip. While it may seem natural to just record the time that the trip meter was started and subtract it from the current system time, this would lead to problems during the system startup. The time needs to increment even while the GPS is connecting, and must be resistant to changes in the systems internal clock. Thus the time is calculated similar to section 6.5.1. The formula used to calculate the elapsed time is given in equation 6.6. The time elapsed is displayed as the highest element of the trip meter in Figure 6.7

$$\text{TimeElapsed} = \text{TimeElapsed}_{\text{lastrun}} + (\text{Time}_{\text{now}} - \text{Time}_{\text{lastrun}}) \tag{6.6}$$

Much like the distance, this method of calculation depends on the last known values. This means it does not matter at the actual time the system trip started once the timer has been running. This makes it resistant to changes in the system time, and thus makes the timer more robust. This timer records the time elapsed on the nanosecond level, as it is used elsewhere in calculations. For display, the timer converts these values into the traditional hours, minutes, seconds format that is easy for the operator to read.

6.5.3 Moving Time

An aspect of the cars telemetry that would be interesting to the driver is the cars moving time. This is defined as the time in which the car has spent in motion. The main use of this data point is to contrast it against the elapsed time, to highlight how long the car has spent sitting still in traffic. This variable is also useful to record for future calculations, such as working out the average speed of the trip. This element is calculated according to equation 6.6, except that it will not update if the current speed of the car is 0 km/h. As such this element requires the speed of the car to be processed, so it cannot be calculated when the GPS signal is lost. This fits in with the functionality defined in section 6.5.1, as the trip meter will not update the moving time or distance driven if the GPS signal is lost. The moving time is displayed below the elasped time and above the distance driven in Figure 6.7

6.5.4 Average Speed

When reviewing the trip meter data, it is useful to know the average speed the car was traveling during the trip. Having this information allows the driver to better understand the characteristics of the drive. This element is also easy to calculate, as the time elapsed and the distance driven are already available. Equation 6.7 shows the formula used to calculate the average speed. The calculated value is displayed to the right of the elapsed time in figure 6.7.

$$\text{AverageSpeed} = \frac{\text{DistanceDriven}}{\text{TimeElapsed}} \tag{6.7}$$

This value is calculated whenever the distance driven or elapsed time values are updated. As this value is using two calculated values, it does not need to worry about discrepancies in time or the loss of the GPS signal.

6.5.5 Average Moving Speed

The average moving speed is like the average speed. The only difference is that it uses the moving time to calculate the speed, rather than the elapsed time. This is done using the same equation 6.7, only substituting "Time Elapsed" for "Moving Time". This value provides the average speed of the car when it was actually being driven, thus ignoring time spent waiting in traffic.

6.5.6 Reset

The final functionality of each independent trip meter is the reset button. This button resets the trip meter it is attached to. The distance driven, moving and elapsed time counters will all display zero, and the average speed calculators will display zero. As each trip meter is independent, one can be reset without affecting the other. To reset the trip meters, the driver just has to press the reset button, located below the average moving time and to the right of the distance driven in Figure 6.7.

6.5.7 Current Speed

The trip meters provide statistics on where the car has been driven, but do not provide much insight into the instantaneous speed of the vehicle. As this variable is already being used in calculations it is trivial to display it to the driver. This is displayed using a simple digit element, and appears in the top right corner of Figure 6.7.

6.5.8 Time Trial Data

A common metric in measuring the performance of cars is to measure how long the car takes to achieve a certain speed. Usually this requires expensive equipment in order to accurately measure the time and speed data. As the information exists inside the trip meter in some form already, it is useful to display a less accurate version of this time trial data. By recording the time it takes to reach 50 or 100 km/h the operator is able to have quick feedback on the performance, without having to setup lots of

equipment. The flow chart of this calculation is given by Figure 6.8. The results of this are displayed below the current speed in Figure 6.7

An important condition on this flow chart is that the zero time must be set before any calculations are performed. The zero time is the time at which the car was last traveling 0 km/h. This time will be reset whenever the car is stopped, so the calculation requires no input from the user. Also present in the flow diagram is that the system will display the best time recorded. If there is a previous best time, and a new one is achieved, the new one will automatically be displayed. This allows the user to ignore the trip meter panel entirely, and be able to trigger and record some performance data on normal drives.

6.5.9 Persistance

TODO: talk about file format and persistance

6.6 Interial Measurement Unit Display Panel

Developed in section

6.7 Arduino Inputs

6.8 Economy Panel

Reduced running costs are one of the most cited reasons for the interest in electric vehicles. As such the system tries to quantify the savings that occur by providing a screen which shows an approximation of the running costs of the vehicle. It does this by making some assumptions as to the cost that the petrol version of the car would consume. This is constrasted against the power that the car has consumed over the same duration. Using these values, the cost of running the car on electricity and on petrol can be calculated and the difference can be displayed. Figure 6.11 shows this panel displayed on the device. The values displayed here are persitant through multiple drives, allowing for the differences to be calculated over a long period.

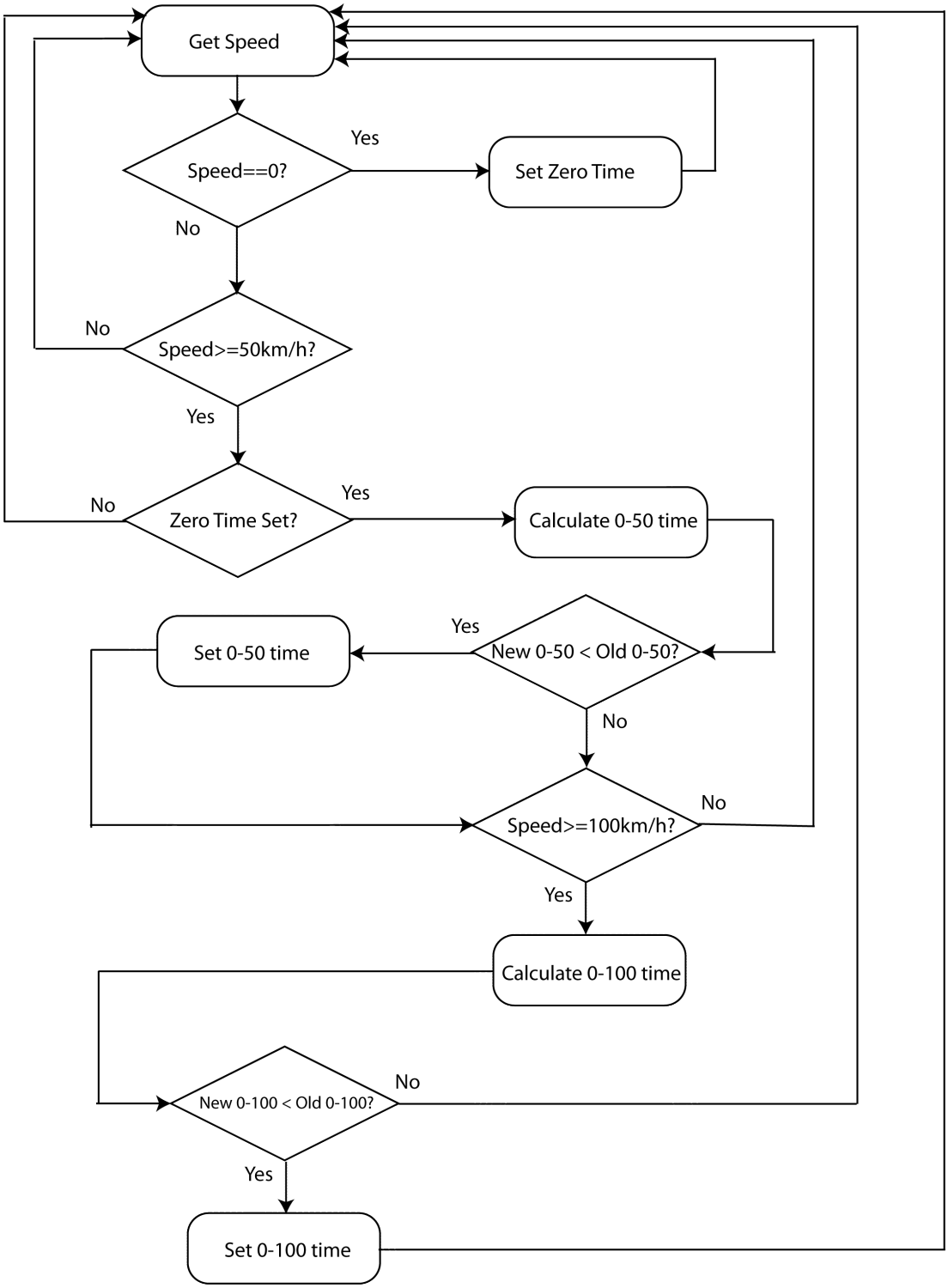

Figure 6.8: Time Trial Data flow chart - The program flow used to calculate the 0-50 km/h and 0-100 km/h time trial data

Figure 6.9: IMU display panel - This figure shows the information output by the IMU daemon

Figure 6.10: Arduino display panel - This figure shows the information output by the Arduino based IO board

Figure 6.11: Savings Panel - This figure shows the savings panel, it calculates the cost of running the car on electricity and approximates the cost of running the car on petrol

6.8.1 Petrol approximation calculation

The car does not consume any petrol, so it is not able to provide data in relation to the amount of fuel consumed. As such the system attempts to approximate the amount of fuel that would be consumed. It does this by using the cars advertised fuel consumption, at 6.1 L per 100 km (**?**). It uses the same method to calculate the distance driven as seen in section 6.5.1. This value is calculated independently of the values calculated in the trip panel, this was done to ensure that each panel cannnot affect the other panel. It is not possible to reset the a trip meter and change the economy panel. The last variable required to calculate the cost is the price of the fuel. While this is highly variable, a single value is used to provide an approximation to the cost.

$$Petrol\ Cost = \text{Distance Driven} * \text{Fuel Economy} * \text{Price per Litre} \tag{6.8}$$

Equation 6.8 shows the formula used to calculate the petrol cost. The distance driven, and cost per litre of fuel are displayed on the display providing insight into how the calculation is performed. The static price of petrol used works with this display, as the user is able to see the links between the numbers.

This is not an incredibly precise calculation, as the GPS can drop out or be slightly off with the distance travelled. It also will not reflect the real cost, due to a static fuel consumption being assumed. It is also unable to take into account the fluctuation of fuel prices over a long period of time. The calculations purpose is to provide a rough ballpark figure, so the user is able to have some indication of the cost the petrol car might have incurred.

6.8.2 Electricity calculation

Given the voltage and current, the instananeous power can be calculated according to Joule's Law (**?**). The formula for calculating the power is given in equation 6.9.

$$P = VI \tag{6.9}$$

The battery monitor module outputs both the current flowing out of, and the voltage level of the battery at a rate of 1hz. With this information it is possible to calculate the power that the vehicle has consumed. By dividing equation 6.9 by the number of seconds in an hour, the power consumed in units of kwh is obtained for the last second. Continually summing this value will lead to the total kwh consumed by the vehicle.

By multiplying the result by the cost in per kwh hour, the total cost can be obtained. Like the petrol calculation, the units of this calculation are displayed on the screen, so the user is able to intuitively understand the calculations being performed. This calculation does have some drawbacks, as it does not take into account fluctuations in electricity costs, such as on and off-peak charging. It does also not take into account in-efficiencies in the charging equipment. The power present in the batteries, and consumed by system while running will not be equal to the power that was supplied to charge the batteries. This is acceptable however, as the purpose of the calculation is to provide rough estimates, in order to provide feedback as to how much the car is costing to drive.

6.8.3 Resetting

Over time, the operator may want to reset the settings back to an initial zero state. This could possibly after months of driving. To faciliate this, the panel has a reset

button present. This button will reset the distance driven and kwh consumed values. As these values are now both zero, the calculated cost values will also become zero.

6.8.4 Persistance

A required feature of this panel, is persistance of the calculated results. If the system is offline, it should remember the calculated values when it next starts up. It achieves this functionality by storing the values needed in a simple text file, that is present with the program's executable. If this file is not present when the program is run, it will be created with zero values. The system constantly

6.9 About

7

Discussion

8

Materials & methods

References

[1] Thom Holwerda. **Linux 2.6.17 Released**, June 2006. 1

[2] Kathleen Potosnak. **Modular implementation benefits developers, users. (separating user interface from rest of computer program)**. *IEEE Software*, **6**(3):91+, 1989. 3

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other German or foreign examination board.

The thesis work was conducted from XXX to YYY under the supervision of PI at ZZZ.

CITY,