

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 their 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	vii
List of Tables	ix
Glossary	xi
1 Introduction	1
1.1 Electric Vehicles	1
1.1.1 Pollution	1
1.1.2 Rising Fuel Prices	2
1.2 Embedded Systems	2
1.2.1 History	2
1.2.2 Telemetry	2
1.2.3 Modularity	2
2 Aims of the project	3
2.1 Final Aim	3
2.2 Preliminary aims	3
3 Literature Review	5
4 System Design	7
4.1 GPS Module	7
4.1.1 Hardware	7
4.1.2 Drivers	7
4.1.3 Design	9
4.1.3.1 Initial Design	9

CONTENTS

4.1.3.2	Final Design	10
4.1.3.3	Network Protocol	10
4.2	TBS Module	10
4.2.1	Hardware	10
4.2.2	Expert protocol	11
4.2.2.1	Destination and Start Byte	11
4.2.2.2	Source	12
4.2.2.3	Device ID	12
4.2.2.4	Message Identifier	12
4.2.2.5	Data	12
4.2.2.6	Trailing Byte	13
4.2.3	Design	13
4.3	Arduino Digital Input Module	13
4.3.1	Hardware	13
4.3.2	Drivers	13
4.3.3	Design	13
4.4	Accelerometer Module	13
4.4.1	Hardware	13
4.4.2	Drivers	13
4.4.3	Design	13
5	Windowing Toolkit	15
5.1	Motivation	15
5.2	Elements	16
5.2.1	UIElement	16
5.2.1.1	addChild()	16
5.2.1.2	draw()	18
5.2.1.3	enqueueDraw()	18
5.2.1.4	animate()	18
5.2.1.5	setActive()	19
5.2.1.6	isActive()	19
5.2.2	base	19
5.2.2.1	base()	20

5.2.2.2	draw()	20
5.2.2.3	getButton()	20
5.2.2.4	addButton()	21
5.2.2.5	addGlobalButton()	21
5.2.2.6	refreshTouchMap()	21
5.2.2.7	buttonPressed()	22
5.2.2.8	activate()	22
5.2.3	runnable	23
5.2.3.1	isScreenChange	24
5.2.3.2	run()	24
5.2.4	Button	24
5.2.4.1	button()	25
5.2.4.2	setPosition()	25
5.2.4.3	getX()	25
5.2.4.4	getY()	25
5.2.4.5	getWidth()	26
5.2.4.6	getHeight()	26
5.2.4.7	invertButton()	26
5.3	Subscriber	26
5.4	Button Loop	26
5.5	Message Queue	26
5.6	Redraw Performance	26
5.6.1	Refresh on Arrival	26
5.6.1.1	Message speed greater than redraw rate	27
5.6.2	Add to Queue	28
5.6.3	Redrawing the Screen	28
5.6.4	Redraw rate	29
5.6.5	Batch Redraw	29
5.6.5.1	Maximum batch size	30
5.6.5.2	Incomplete batch	30

CONTENTS

6	Interface	31
6.1	Layout	31
6.1.1	Background	31
6.2	Overview Panel	32
6.3	Battery	32
6.4	Maps	33
6.4.1	Map Data	34
6.5	Trip Meter	37
6.6	Arduino Inputs	37
6.7	Cost Panel	37
6.8	About	37
7	Discussion	39
8	Materials & methods	41
	References	43

List of Figures

4.1	Flow chart of the battery monitor daemon	14
5.1	UML diagram of the window toolkit	17
5.2	Processing message flow chart	27
5.3	Appending to Queue	29
6.1	Panel showing other panels	32
6.2	Battery state panel	33
6.3	Map display panel	34
6.4	Pre-rendered map size	36
6.5	Panel showing other panels	37

LIST OF FIGURES

List of Tables

GLOSSARY

Glossary

BMS Battery Management System

CPU Central Processing Unit
GPS Global Positioning System
TCP Transmission Control Protocol
ZMQ Zero MQ

GLOSSARY

1

Introduction

1.1 Electric Vehicles

There are many motivating factors behind the development of electric cars. These vehicles utilize new technologies and 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 electricity. This does not make 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. INTRODUCTION

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

this data to be reviewed later

2. AIMS OF THE PROJECT

3

Literature Review

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 difficulty was the out-dated Linux kernel being run in the system. This was kernel version number 2.6.17 and was

4. SYSTEM DESIGN

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

4. SYSTEM DESIGN

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 message 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 guarantees 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

4. SYSTEM DESIGN

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 in its 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 hexadecimal 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 signify 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

4. SYSTEM DESIGN

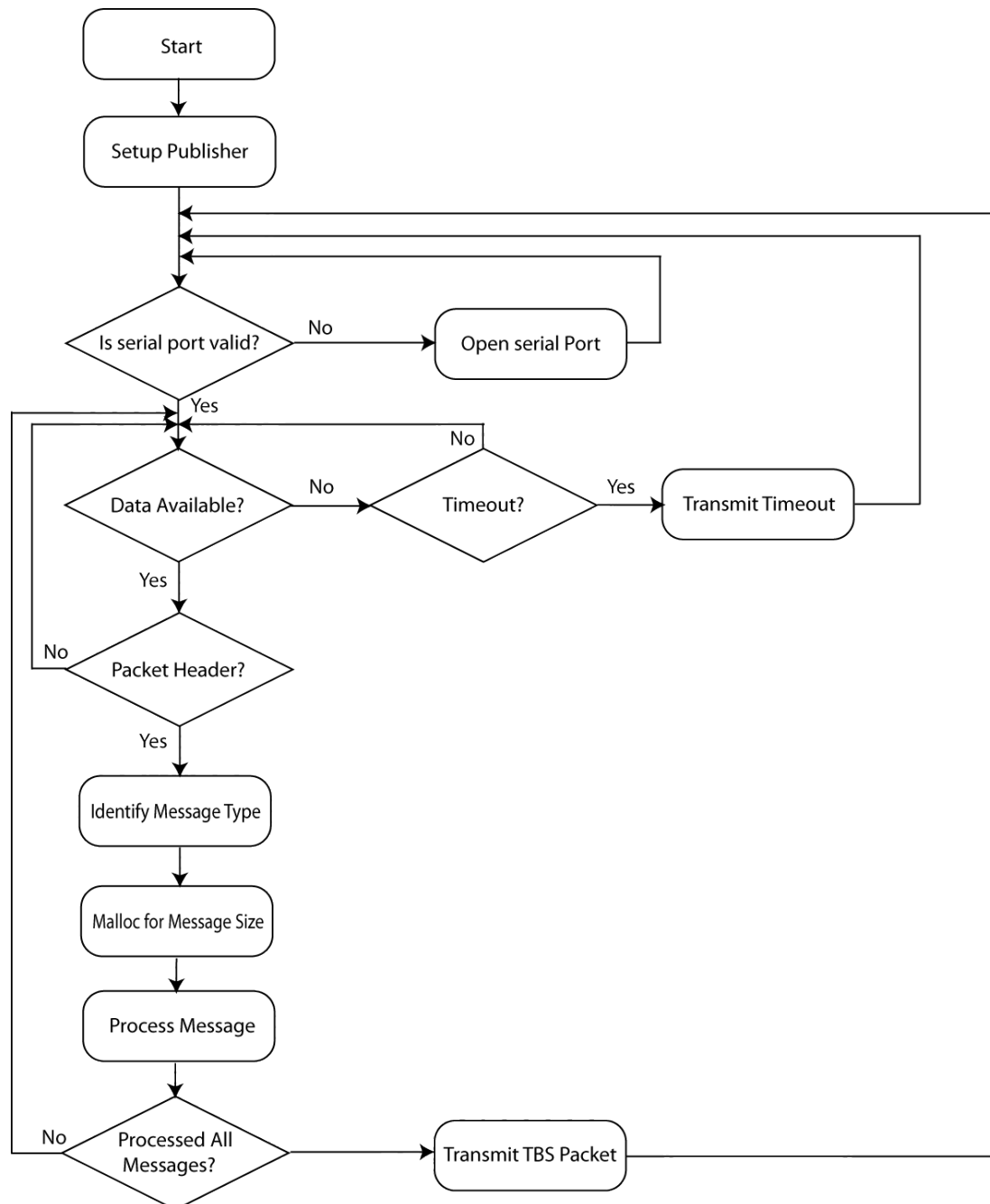


Figure 4.1: Flow chart of the battery monitor daemon -

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 pre-definable 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

5. WINDOWING TOOLKIT

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

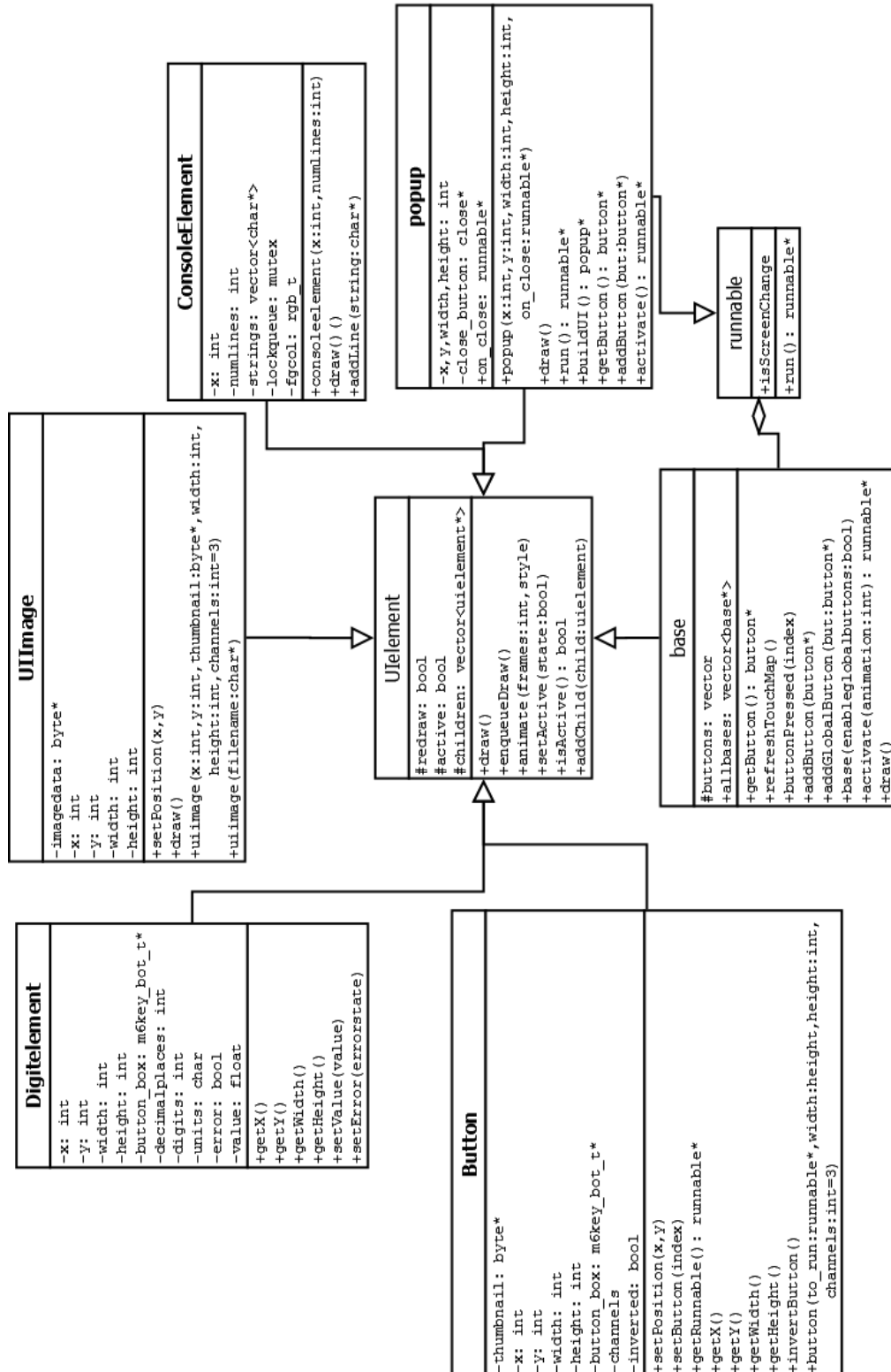


Figure 5.1: UML diagram of the window toolkit -

5. WINDOWING 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 redrawn, 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. WINDOWING TOOLKIT

5.2.2.1 `base()`

The constructor for this class takes a single argument. This argument is a boolean value indicating whether this panel will display global buttons or not. By default this option is set to true, though it can be easily overridden 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 developer only wishes to display a simpler panel, or when the current set of global buttons would appear inappropriately 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 consistency 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 will be enqueued onto the drawing queue first, all its 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 transitioning 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

5. WINDOWING TOOLKIT

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 5.2.3. 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 transision 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 transisition 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 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. WINDOWING TOOLKIT

5.2.3.1 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 its 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.

5.2.3.2 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.

5.2.4 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 its 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.

5.2.4.1 `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 5.2.3. The image data must be an array of bytes formatted as either RGB or RGBA pixels. As the image data is a contiguous 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 omitted the system will assume a RGB image is supplied. This will be programmatically 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.

5.2.4.2 `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 the 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.

5.2.4.3 `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.

5.2.4.4 `getY()`

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

5. WINDOWING TOOLKIT

5.2.4.5 `getWidth()`

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

5.2.4.6 `getHeight()`

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

5.2.4.7 `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.

5.3 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 a windowing system was already being developed, it was desirable to have this system abstract these actions away from the programmer. By using the toolkit the programmer can trigger when an element should be drawn, and specify the drawing code to draw the element. The abstraction 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 received 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.

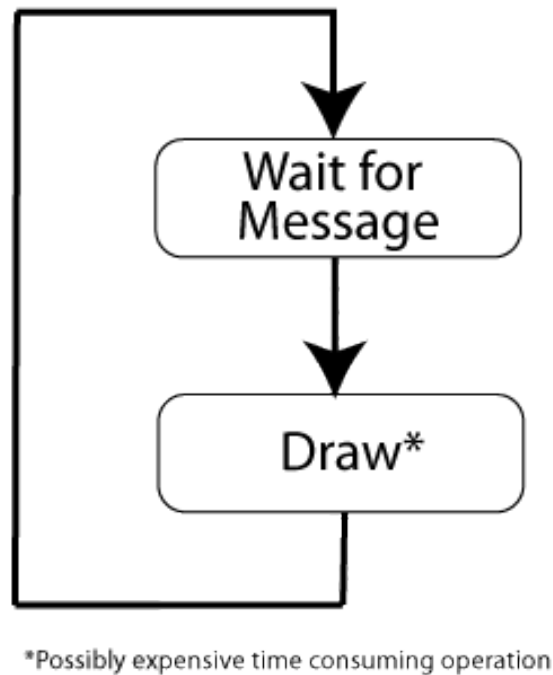


Figure 5.2: Processing message flow chart - This figure shows the naive approach to processing and displaying recieved message data

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

5. WINDOWING TOOLKIT

a message”. However another important sensor caused problems with this method. This sensor was the one designed to read the GPS signals. Messages containing GPS information were 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 a lot of new messages to be dropped. The other issue that occurred 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 occurred 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 were received. 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 occurred with allowing the screen to update whenever a message was received. The biggest performance penalty that occurred 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.

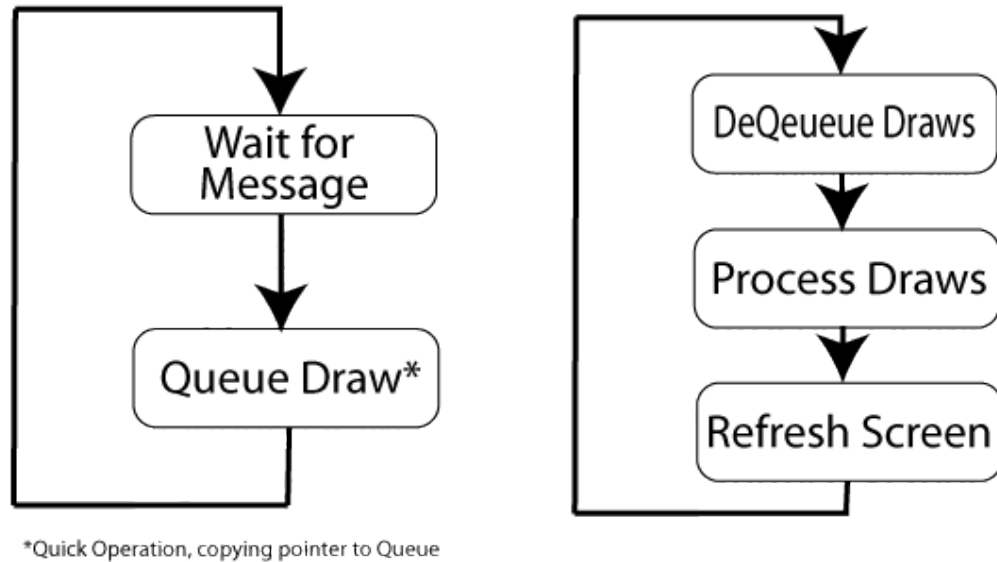


Figure 5.3: Appending to Queue - This figure shows the flow of appending screen refreshes to a queue

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. WINDOWING TOOLKIT

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 guarantees 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 received. The solution to this problem is to continue on with the drawing actions if the queue is ever empty. This prevents the thread from waiting for more events, and helps guarantee 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. INTERFACE

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 panel's 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

$$70 = 10 \quad (6.1)$$



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

6. INTERFACE

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

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 an 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

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

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. This 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 suitable 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 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 ??. This area is depicted by the image shown in Figure 6.4. 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.

6. INTERFACE

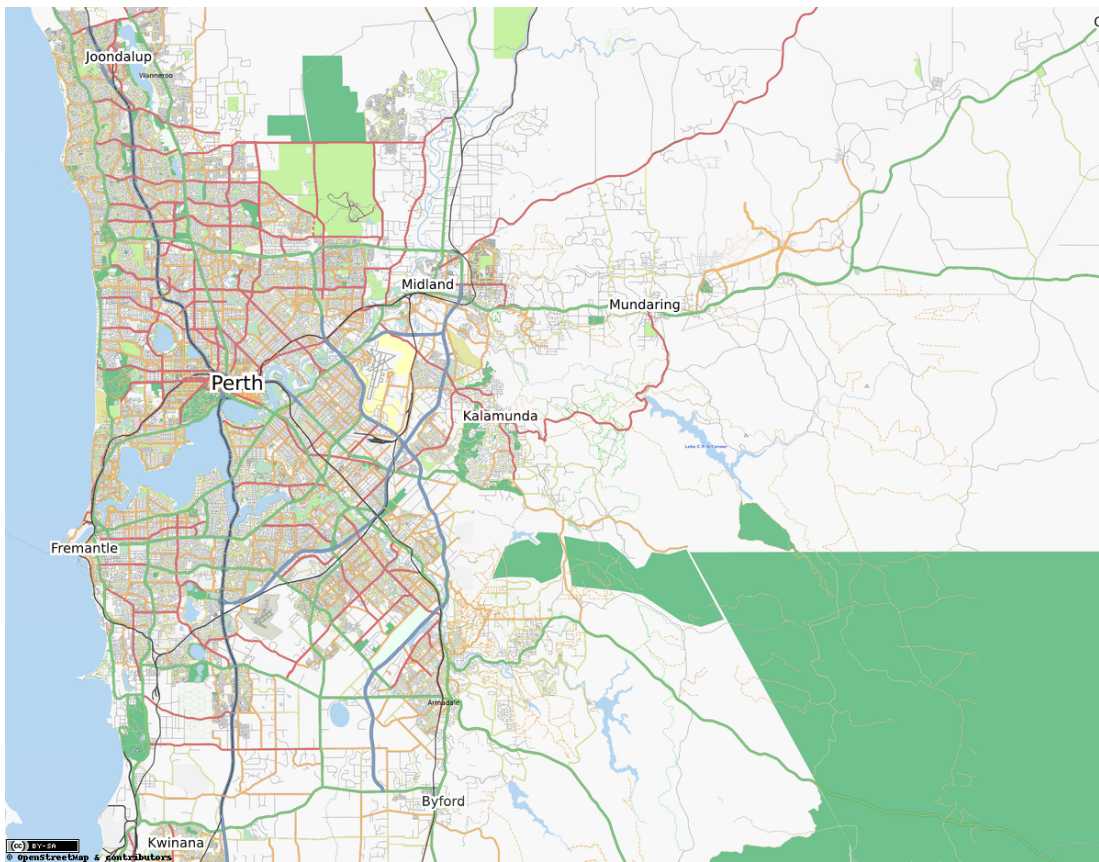


Figure 6.4: Pre-rendered map size - This figure shows the area that has been pre-rendered for use in the map panel display

6.5 Trip Meter

6.6 Arduino Inputs

6.7 Cost Panel



Figure 6.5: Panel showing other panels - Shortcuts to different aspects of the program.
(Battery, Maps, Trip Meter, Accelerometer, Arduino, Savings, About, Options)

6.8 About

6. INTERFACE

7

Discussion

7. DISCUSSION

8

Materials & methods

8. MATERIALS & METHODS

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

References

- [1] THOM HOLWERDA. **Linux 2.6.17 Released**, June 2006. 1

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,