Computer control of the amateur radio "Yaesu FT8900"

Final Report for CS39440 Major Project

Author: Cormac Brady (cob16@aber.ac.uk, cormacbrady@hotmail.co.uk)

Supervisor: Dave Price BSc, MSc (Wales), MBCS (dap@aber.ac.uk)

24th June, 2021

Version: 1.0 (Release)

This report was submitted as partial fulfilment of a BSc degree in Computer Science (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name : Cormac Brady

Date : 24th June, 2021

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name : Cormac Brady

Date : 24th June, 2021

Acknowledgements

I would like to thank everyone for there support in helping me to produce this report. In particular, I would like to thank my support tutor, Heather Philips for helping me stay organised for the last three years of my degree. I would also like to thank my supervisor, Dave Price for his help in learning the details of amateur radio and its terminology. Many thanks also to Peter Todd, for his help with the physical side of the project as well as for putting up with my barrage of questions about serial and electronics.

I would also like to thank my friends Owen Garland, Xander Barnes and Nicholas Dart for helping my studies at Aberystwyth University stay both focused and entertaining.

Of course I would never have even been able to even write this acknowledgement section without the enduring love and support of my parents, Mark and Christina. They helped me through my childhood and gave me the opportunity to find and pursue my interest in computing.

Finally I would like to thank Laura Pugh for all her love and support that she has given me throughout my undergraduate degree. She is the most kind person I know and has put up with me moving over 600km away to Germany for my industrial year.

Abstract

In amateur radio, C.A.T (Computer Aided Transceiving) is the name for a radio's ability to interact with a computer, often via serial communication. This is often done so that the radio can behave like a computer peripheral. There are many applications that rely on this premise to provide a user interface on a computer screen to fully control a radio.

The Yaesu FT-8900R (FT-8900) is a popular budget amateur radio. However, the the FT-8900 it is missing a significant feature, C.A.T (Computer Aided Transceiving). The design of the The FT-8900 allows users the potential to add this feature to the radio. This is due to its ability to split off the front control surface from the main body. The detachment is made via a connecting serial line and is intended to be used to mount the controls of the radio in the console of a car.

This report details the process of research, design and development of an application that is capable of controlling the radio, by tapping into this serial connection. The application was developed as an alternative to existing commercial solutions, with further consideration given to cost constraints. The typical conditions expected of CAT capable radios gave rise to strong efficiency and portability requirements.

The end result was an application written in C that met it's objectives to produce a minimum viable product. The produced application has been evaluated for its quality and suitability as a solution.

CONTENTS

1 Background **1**

2 Problem analysis & Objectives **3**
2.1 Analysis . . . 3
2.1.1 Protocol . . . 3
2.2 Aim & Deliverables . . . 5
2.3 Security . . . 6
2.4 Process . . . 6

3 Design **10**
3.1 Hardware . . . 10
3.2 Overall Architecture . . . 13
3.3 Language choice . . . 15

4 Implementation **18**
4.1 Iteration 0 . . . 18
4.2 Iteration 1 . . . 19
4.3 Iteration 2 . . . 19
4.4 Iteration 3 . . . 20
4.5 Iteration 4 . . . 21

5 Testing **23**
5.1 Approach . . . 23
5.2 Automated Testing . . . 23
5.2.1 Unit Tests . . . 23
5.3 Manual Testing . . . 25
5.3.1 Memory profiling . . . 27
5.4 Known issues . . . 28

6 Evaluation **29**
6.1 Design Reflection . . . 29
6.2 Suitability of tools . . . 30
6.3 Improvements . . . 31
6.4 Objectives . . . 31

Glossary **33**

Appendices **34**

A Third-Party Code and Libraries **35**

B Ethics Submission **36**

C Code Examples **39**
3.1 Python Bitfield . . . 39
3.2 Unit test output . . . 41

D Code Documentation **43**

Annotated Bibliography **84**

LIST OF FIGURES

1.1 Picture of RT-8900r radio 2

2.1 Control packet 4
2.2 CRC front 7
2.3 CRC back 7

3.1 Hardware Overview 10
3.2 Radio locker 11
3.3 Prototype breakout board 11
3.4 Circuit diagram 12
3.5 basic architecture 13
3.6 Sender thread 14
3.7 Overall architecture 14

4.1 Packet mapping 21

5.1 A screenshot of the CI dashboard. This can be left open on a screen to track the build status. On failed builds this would turn red with a reason specified. 24
5.2 Jenkins build trends 24
5.3 Tests on github 25
5.4 A Raspberry Pi running the application 26
5.5 Memory usage graph 27

6.1 Project statistics 29
6.2 Dependency graph 30

LIST OF TABLES

3.1 Summary of languages comparison 17

4.1 13 Segment Truth table 22

5.1 Acceptance testing 26

6.1 List of objectives/deliverables 31

Chapter 1

Background

> "Amateur radio is a popular technical hobby and volunteer public service that uses designated radio frequencies for non-commercial exchange of messages, wireless experimentation, self-training, and emergency communications. Amateur Radio is the only hobby governed by international treaty." [1]

Amateur radio operators do not always sit in front of their radios. Often for operators it is more convenient to control their radios remotely, known more commonly as having a "remote shack". In this scenario as much remote control as possible is desired. This is often achieved using existing commercial or home made solutions that rely on simply repeating control signals. For example the "RRC 1258Mk" is a product designed to do just. However it is priced at $708 [2] making it often prohibitively expensive as it is two times the price of budget consumer radios. Instead it would be preferable to understand and/or emulate these signals. The end result is that the radio gains Computer Aided Transceiver (CAT) [3] features.

Remote shacks are geographically positioned for their prime transceiving properties, for example on the top of a hill so the signal is less affected by the geometry of the surrounding area. They are often limited in their connectivity to utilities, such as power or Internet connection. This is especially in extremely remote areas where communication is sometimes done over telephone networks. This adds some constraints to the project and is discussed in the next section.

An ideal scenario for these kind of users is where all functions can be controlled from the radio remotely, using as little data traffic as possible. This use-case also extends to users that prefer to use a PC to control their radios. This may be for more features, better user experience (given by software), or possibly for accessibility reasons (i.e where reading the screen or using the small dials would be a problem for the operator).

The FT-8900 (Seen in figure 1.1) is a popular, budget amateur radio, it is marketed for mobile applications such as being mounted on a car dashboard. This is achieved using a control surface (hereby referred to as the head) that is detachable from the rest of the body of the radio. This allows the body to be mounted in a more practical location by running a control cable between the two. The FT-8900 is capable of monitoring two separate frequencies at the same time, therefore there are duplicate controls on ether side of the radio to manipulate each receiver. The microphone connects into head and contains additional buttons such as a keypad that is able to dial a exact frequency. Conversely the loudspeaker is located in the body of the radio so a second cable must be run from the line out jack at the back of the radio to the head in order to hear audio clearly.

Figure 1.1: A picture of the face of the RT-8900r radio.

The main goal of this project is to allow the functions of the radio to be available from a computer. This will be achieved by adapting the FT-8900 to have CAT features. Radios that come with this feature typically have a serial port or some kind. My project relies on the assumption that it is possible that the control line can be intercepted and emulated by a computer. This means that functionality could be added in a similar way to radios that advertise the C.A.T feature. This must be achieved cost effectively and in a simple way for the user to create and setup by themselves. Any designs and software will then be released under the GNU Public Licence (v3) for the benefit of the community.

Chapter 2

Problem analysis & Objectives

2.1 Analysis

After confirming the data port on the back of the device did not provide enough control [1], it became clear that the only way to interface with the radio without opening up the case (and invalidating the warranty) was to utilise the serial line that connects the head and body of the radio.

Previous work on understanding the serial protocol between the body and head of the radio was undertaken and compiled by Ben Cooper [4] [5]. This work helped to ascertain the basics of the serial communication. Ben Cooper's project was focused on developing software aimed to run on a Arduino microcontroller. He concludes that the severe complexity of sending, receiving and processing on a single thread was perhaps reason to consider other options.

Considering the application may need to be run in these remote shacks with limited power and Internet connectivity. The project will aim to be as power and memory efficient as possible, so that it can be run on low specification hardware. This gives the benefits of a lower cost. Additionally for this reason the application must be portable for at least the x86 and ARM architectures so that it is actually possible to run on a low spec machine. This will give the user as many options as possible, for example low power PCs such as the Raspberry Pi or to just control the radio straight from a laptop that has serial capabilities. Development of the project will also benefit as common laptop/desktop hardware can be used with the use of a USB Transistor to transistor logic (TTL) dongle. These dongles are cheap and ubiquitous, often costing less than £1 each. This gives a low barrier to entry as it is the only dedicated hardware required for most users.

2.1.1 Protocol

Most of the protocol was already reverse engineered by other authors such as Ben Cooper [4], these assumptions were confirmed using a digital oscilloscope by being able to decode both serial transmission to the radio (TX) and serial transmission from the radio (RX). Figure 3.4 shows a circuit diagram, the connectors at each end of the circuit are RJ25 ports (aka telephone jacks) with 6 active lines. In the order of left to right from the perspective of looking into the socket, the first 2 lines are for RX and TX of serial, ground, 9v power, power switch and the last line is for

[1] The data port only allows input and output of audio, transfer of station memory when the radio is in a special mode and control of squelch level.

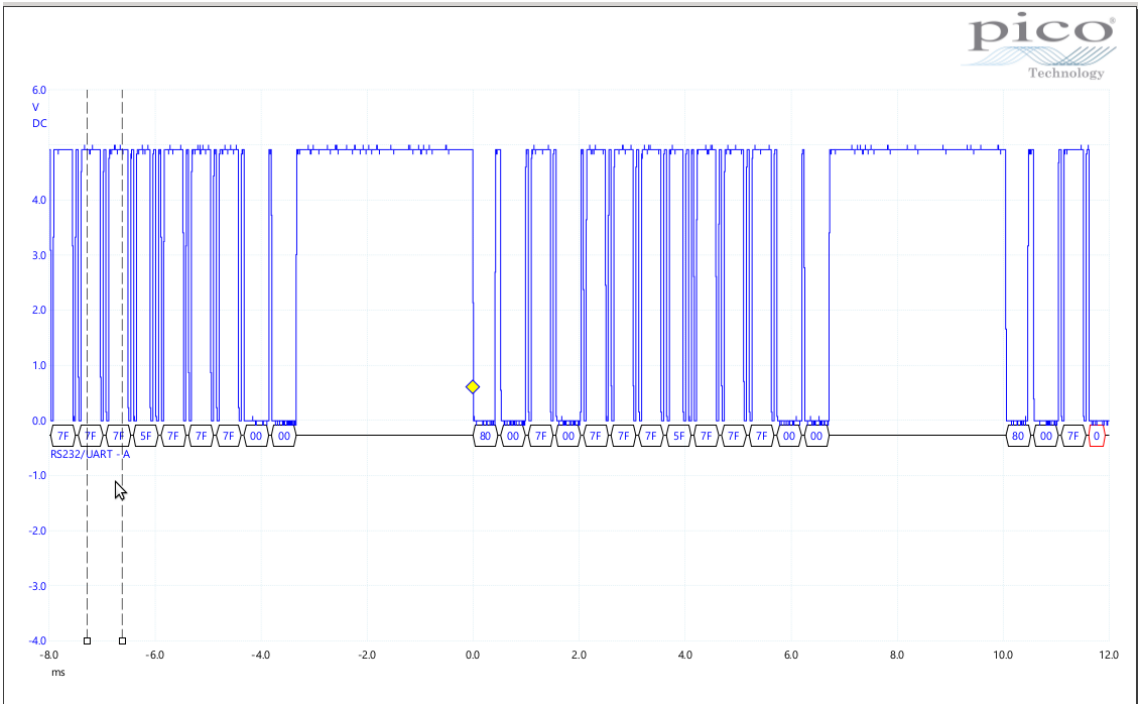


Figure 2.1: A capture of control packets. The 2 long peeks are the begging and end of the middle packet

the microphone. The radio does not use the common RS232 standard[2] instead it uses the older TTL standard. In this case this means that an idle line is at 5V which represents a 1 bit and 0V representing 0 bit. The beginning of a transmission is marked with a single start bit, followed by 8 bits of data and then a single stop bit. There is no error checking through parity or otherwise.

So to summarise the communication:

- 0-5V
- TTL protocol
- 8 data bits with 1 stop bit
- The most significant bit is transferred first
- 19,200 baud rate
- Automatic shutoff after 80ms

TX from the head to the body of the radio does two things. For analogue buttons it transmits the current state of those buttons and for digital encoders, such as the dials, it transmits the number of rotations since the last TX. The packet of data that is sent consists of 13 bytes (mapped previously [4]). This could be utilised as a method to provide emulated input to the radio. Actions could be undertaken by the program to replicate any input that a human was capable of.

[2]RS232 sends binary with -13V to represent a one bit and +13v for a 0 bit, allowing for easy knowledge of when the line is idle at 0V

RX (from the body to the head) is a packet of 42 bytes. Each bit in the packet represents whether a segment of the screen should be illuminated or not, for example what frequency should be shown. The program can listen to this to learn the internal state of the radio. The state of the radio that is not represented on the screen will have to be discovered by navigating the menus and then reading the screen.

During the project there where some new observations made to the transmission protocol. The most significant bit (MSB) is transmitted first between the radio and head with a 20ms gap between both RX and TX packets. The head will transmit its TX packets as soon as it is given power if the body is given the boot signal (setting the 5th line to low) but without receiving head packets it will shutdown after 80ms. The only indication of this boot failure is is a small audible click sound from the loudspeaker.

2.2 Aim & Deliverables

The scope of development efforts will be to create a way to control the body of the radio, with the aim to achieve as much functionality as possible (controllable by a computer). Communication to the head, for the purposes of using it as a generic control surface will remain out of scope. This non-critical feature would be at a high cost in time spent, when instead more important control features could be implemented. However work done that is in scope will be foundational to this feature if desired later. Dealing with the audio from the radio was also deemed out of scope, as the data-port of the back of the radio can already achieve this. Users can use a cable speaker line out and microphone line out from the back data port.

Below is a list of deliverables. Necessary functions were derived by asking a number of "Customers" what they would consider their minimum set of features (aka the planning game) and there expected difficulty. They are also in order of priority. Therefore this became the priority backlog.

- Final report
- Software functions that have been deemed necessary for radio control:
  - Frequency
  - Push to talk
  - Volume
  - Squelch
  - Power
  - Powering on the radio
- Documentation
  - Code documentation (Doxygen)
  - Schematic of prototype serial controller

Futures such as a more permanent printed circuit board and control of all functions of the radio were deemed not possible in the allotted semester of work[3]. Originally there was expressed intent

[3]From 30th January, 2017 to 8th April, 2017.

on providing a patch to the open source Ham Radio Control Libraries, Hamlib [6]. The Initial report outline submitted reflected this. However after discussion of this intent on the developer mailing list it became clear that a multi-threaded application was unacceptable within the Hamlib codebase, due to concerns with portability and stability. Instead future work will likely include some sort of inter process communication through a socket.

2.3 Security

The envisioned scope of the project means that the software will not be networked in the modern sense of using IP communication. If the project did so there would be many security issues raised in how connections are handed, what kind of authentication is used and how will traffic be encrypted. Major operating systems manage access to serial communications often by requiring special privileges or groups. Therefore in this case, loss of access control can be prevented using just the default system configuration of most operating systems. The application will endeavour to not manipulate the system as much as is possible so as to minimise the attack surface given. Attention will be given to proper copying in memory and validation of incoming received serial communications to mitigate fuzzing based attacks.

Considerations must be made on who has physical access to the computer connected to the radio. As another user could plug the serial connection into their computer in order to interface with it, thereby forgoing the access control that was in place on the system. Users will also have to consult their local laws on whether it is legal for them to operate their radio remotely in their respective countries.

2.4 Process

Due to the large potential scope of the project it was necessary to choose an agile methodology so that the large set of possible requirements could be better managed and prioritised. Therefore agile software development methodologies had to be constituted and chosen on their merit for the project. Methods considered were Scrum, extreme programming (XP) and feature driven development (FDD). FDD was quickly ruled out due to its major benefits only seen with larger teams that need to be able to produce regular status reports. For this project it was feared this would slow down the rate of development of features. Scrum was seen as less useful than XP as it does not talk about the actual engineering steps to take in programming unlike XP. Instead it focuses more on general management for a team. Therefore XP [7] was chosen, however it still had to adapted for a single person project.

XP guidelines for design can be summarised to the following [8]:

- The Planning Game
- Small Releases (Iterations)
- System Metaphors
- Simple Design

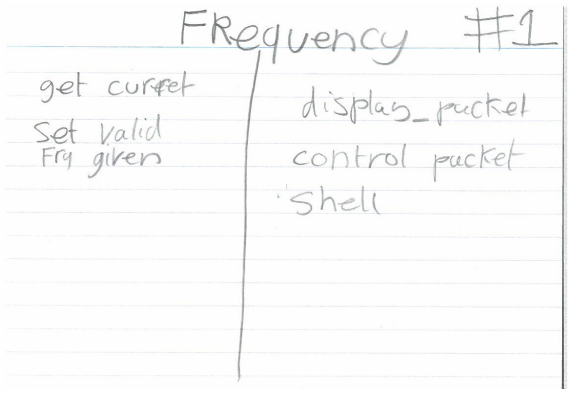


Figure 2.2: The front of the CRC cards contained on the left the required functions of the feature and what it will deal with on the right.

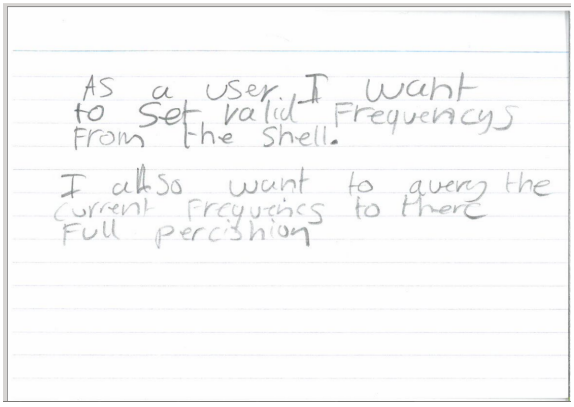


Figure 2.3: The back of the CRC cards contained the acceptance tests for the feature.

The above points are very compatible for a lone developer project. The planning game was used with potential "customers" (Customers were any potential user of the radio i.e with the project supervisor), by requesting features to be prioritised. This was then used to produce and sort Class, Responsibilities, and Collaboration (CRC) cards (See figures 2.2 and 2.3 for the front and back of a CRC card respectively). Use of system metaphors was fairly common when discussing the project, this can been seen in this report with the "head" and "body" metaphors and standardisation of point of reference to serial communication with the RX/TX terms. Small releases at the end of each iteration were make allowing users give regular feedback. The initial design of the application was made to be just large enough to start development without the need for heavy refactoring later.

The actual development methods used in XP will need to be modified in order to accommodate one developer. Thankfully adaption to a single developer as been discussed openly, including from the creator of XP, Kent Beck in a number of mailing list discussions [9] [10].

The following sections will address the adaptations made:

Unit Tests

The described use of unit tests do not need to be changed for the project. The test suite "Google Tests" [11] was chosen despite the fact that this is implemented in C++, as it is often used to test C code and better support and features than native suites. Its test macros are familiar to programmers that have used suites such as JUnit etc. Unit tests were especially useful for the project as the real hardware was not always available for testing the radio was stored in a university lab that was shut on the evening, Fridays and weekends.

Acceptance Tests

Acceptance Tests are created on the back of CRC cards. A successful test is likely to be the ability for the user to preform said action on the real radio without any input from the developer. A passing acceptance test is the only verification of a completed feature.

Refactoring

Refactoring should occur naturally throughout development as well as during the refactoring stage of creating a function with the Red, Green, Refactor (RGR) method. Good use of unit tests will make refactoring easier as developers can make changes with confidence of not braking the system.

Pair Programming

XP has a heavy focus on pair programming. This project work is not permitted with other developers in order for a full assessment to take place of work done. Instead other methods that give some of the benefits of group collaboration such as"rubber ducking" will be used. Rubber ducking is the process of talking out or explaining code to oneself in order to check that there are not better ways to do something. Throughout the project conversations with other students about the project, for example when walking up to campus was used to provide reflection. Questions from others

asking why something was done in a particular way can often promote further analysis and better ideas.

Continuous Integration

Git will be used as a VCS for the project using branches for new features before merging them, this was chosen over over VCS due to its better branching and mergeing as well as ability to still commit when disconnected from the Internet. Github will be used to host the central repository, due to its good user interface as well as its large popularity (meaning new user discovery is possible). A self hosted version of Jenkins [12] will used automatically test commits by running builds. Jenkins CI was chosen due to previous positive experiences with it as well as its high level of customisation owing to its large plugin library.

Coding Standards

The "Linux kernel coding style" [13] was chosen as it is a widely used standard that has a good rationale and format. To enforce a consistent style checking to was made in the text editor, as well as a step in continuous integration checks to look for style violations.

Iterations

The project will run using two week iteration/release cycle. It is hoped that this will give enough time to plan, develop, and physically test the radio functions. In this time the current most needed feature from the priority backlog will be worked on. To do this a number of engineering tasks are made from the feature and placed as "TODO" items in the source code. This is the most appropriate place as tasks remain very visible until removed, unlike when tasks are recorded on a webpage elsewhere. At the end of an iteration a release is tagged in git with a version number that conforms to the semantic versioning convention [14]. Additionally a changelog is made on the developer blog. This means there are regular versions of the software that can be tested and reflections can be made about the progress of the project.

Chapter 3

Design

3.1 Hardware

In order to develop software for the radio, it had to be ensured during development that an error in the program did not lead to rouge transmissions. To do this the radio was suspended on a wooden shelf placed in a grounded metal locker (See Figure 3.2). Connected to the radio was a dummy load in place of aerial, with power to the radio coming through a whole in the side of the locker so that the door could remain closed. This helped to reduce the strength of any transmissions significantly but not entirely. As a final precaution a second hand-held radio was placed next to the FT-8900 to monitor for any transmissions. If this was to occur the developer could then shutdown the radio quickly so as to avoid prolonged transmission.

The serial brake-out board (shown in figures 3.3 and 3.4) passes though the required connections to the head of the radio. Allowing the head to still be used, even when the body is controlled by the radio. The ground, RX and TX serial connections are tapped by the serial dongle with the TX being the only disconnected line from the radio. There is also a physical button that connects the power switch to ground when pressed, effectively having the same effect as pressing the power button on the head of the radio.

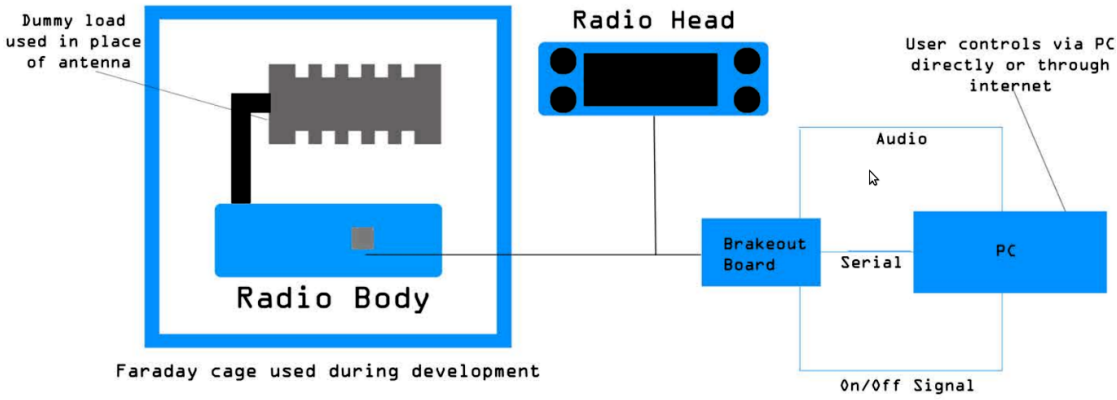


Figure 3.1: A hardware overview of the project.

Figure 3.2: A picture of the metal locker used to contain radio waves made from the radio body. The black box below the wooden shelf is the radio body. The box above is the dummy load.

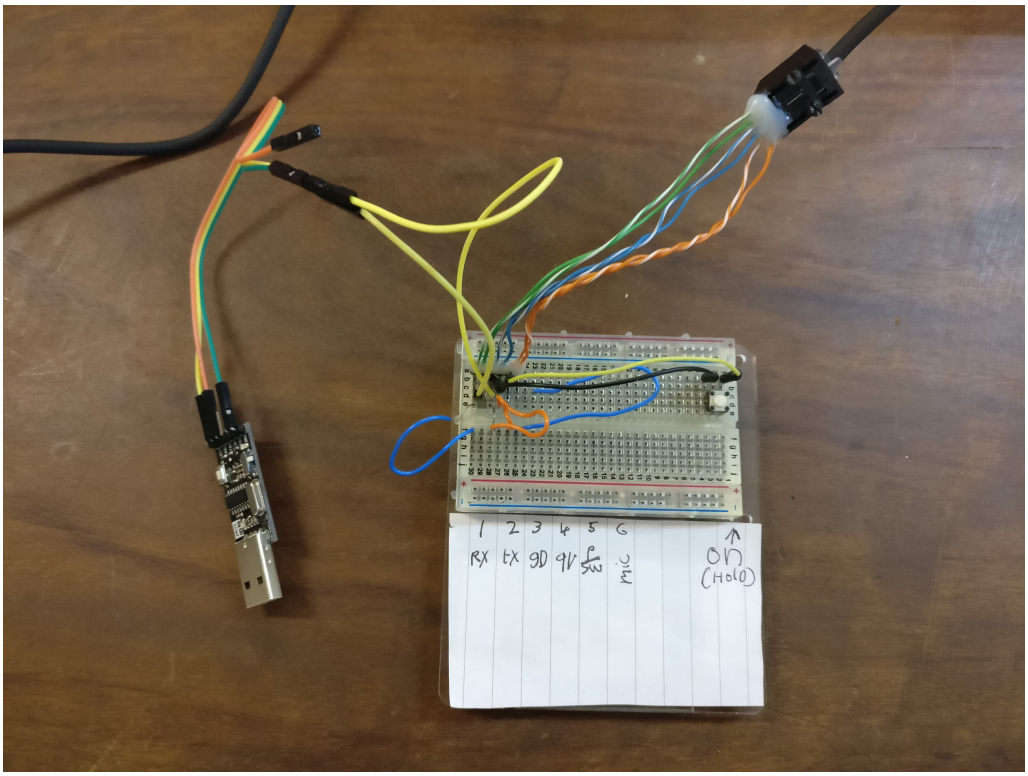


Figure 3.3: Prototype breakout board used for development.

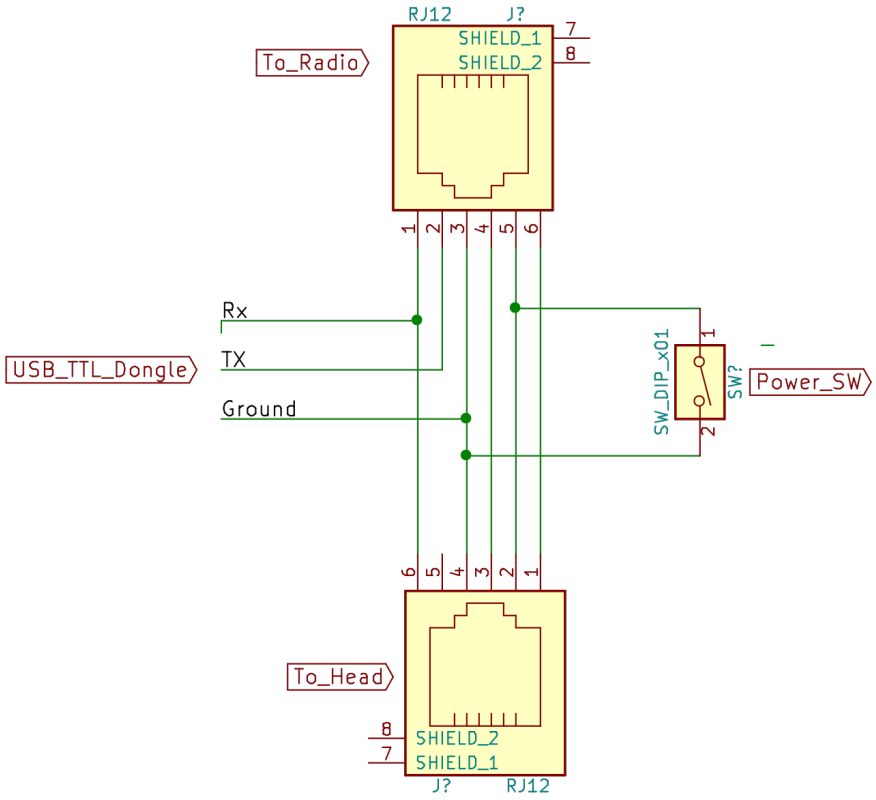


Figure 3.4: Circuit diagram of the brake out board

3.2 Overall Architecture

The more generic functions of the application will be abstracted out to a library so that they can be reused within other applications. The application will provide a console or shell for the user to control the radio from. A console is an appropriate interface as the XP methodology demands that designs be made as simple as possible, while still satisfying acceptance tests. Another reason is that the program will ultimately be, an intermediary for signals and not the primary way for the user to control the radio. This is due to the large number of existing and successful free radio control applications designed for such a purpose; for example GRig [15].

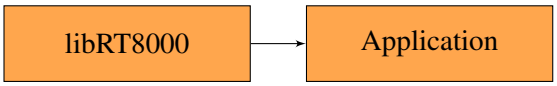


Figure 3.5: The basic overall library/application architecture

The the radio must receive packets every 80ms, else it will automatically shutdown. With such a time critical task such as this, the routine was given a separate thread to operate in, so as to lower the impact of other routines on its timing. This is described in figure 3.6.

Items in the queue are only removed (popped) when there is more than a single packet inside. This is done in order to prevent the automatic shutdown of the radio. The effect is that the last packet is kept in the queue and resent continuously until a new packet is added or the program shutdown.

While designing the packet sender thread the idea of the "default packet" was imagined. This packet will contain much of the state of the radio. To change the volume the default packet is directly modified. More complex actions, for example actions that require a button to be pressed only momentarily, will add new packets to the queue. As there will possibly be duplicate packets in the queue, the queue implementation must hold only a reference to the default packet.

It was anticipated that the bulk of memory usage will be to store representations of packets. Therefore the data structure should be as memory efficient as possible. Although this must balance with the ease of use i.e sections being able to be mapped accordingly. Packets are to be stored as a bitfield. This is so that the RX packet will take up only 42 bytes (or as close to this as possible). Bit-wise operations would then be used to read and set bits inside the packet. This has the added benefit that sending via serial will not require any additional processing, as the representation is identical.

Packets sent to the screen (RX) will hereby be referred to as "display packets". These are used to generate the output of the display, where a set bit in the packet corresponds to an illuminated segment on the display. The packets that are TX to the radio body will be referred to "control packets" as they communicate the current state of the controls back to the radio.

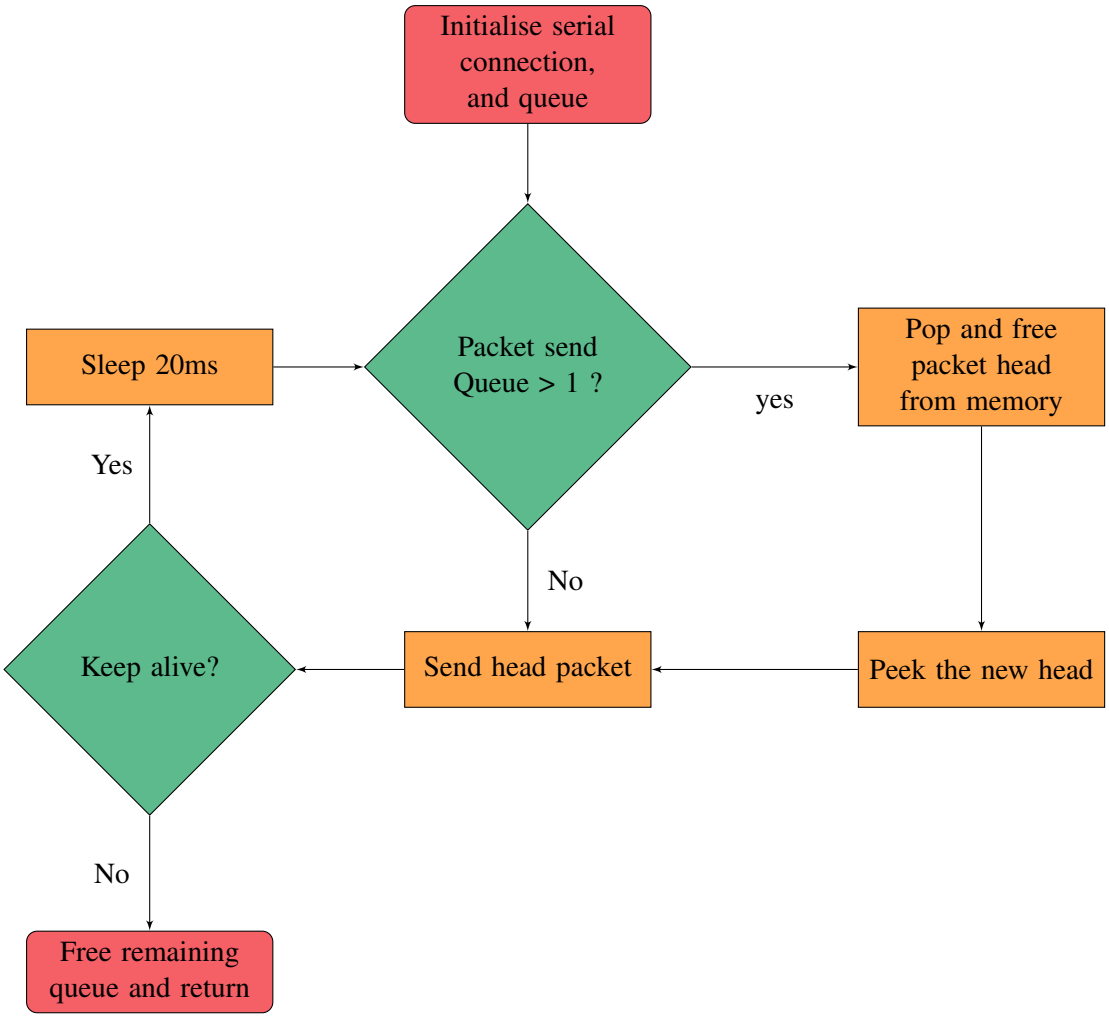


Figure 3.6: Flow diagram of the Sender thread

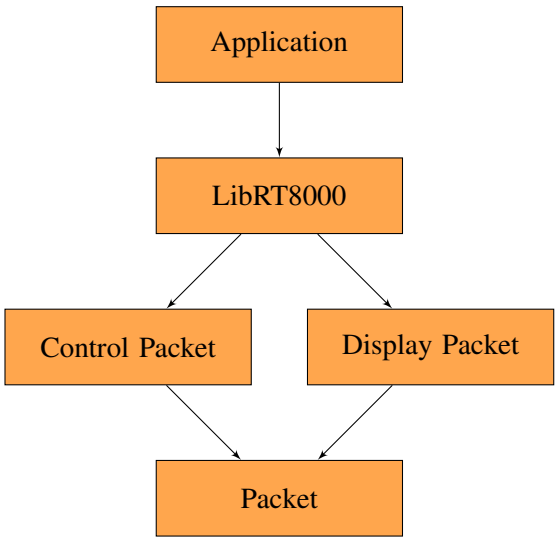


Figure 3.7: A more detailed vision of relations of the application

3.3 Language choice

A number of candidate languages were selected as potentially appropriate. Five main criteria were used to decide what was most appropriate:

- Prior experience
- Existing serial support
- Ease of bitfield storage
- Portability
- Speed

Python

First a scratch program was created to show that it was actually possible to store Bitfields, with mappings to their relevant sections i.e the left volume section (See section 3.1 for this code). This proved promising, but there still remained the question on whether Python could store this in a sufficiently compact manner. In order to test this a 336 bit number (42 bytes) was stored in the interpreter and then queried on how much space had been allocated for it.

```
>>>import sys

>>>a = (1 << 336)
>>>sys.getsizeof(a)
72
```

Python had given a 71% overhead to the bitfield. 30 bytes more than required is sub-optimal for our usage.

The other concern was how fast the serial library was in Python. While Python does not have serial in its standard library, it does have a well supported third party library named "PySerial" [16]. A scratch program was written to test this.

```
import serial
ser = serial.Serial('/dev/ttyUSB0')  # open serial port

for _ in range(100):
    ser.write((1 << 336))
    ser.flush() # wait until message sent

ser.close()
```

This program sends 100 x 336 bits as fast as possible to the serial device located named "ttyUSB0". The serial line was then connected to a digital oscilloscope in order to measure the

performance. The width between packets was a high 40ms, where the actual delay of the radio was 20ms between packets.

At the time this was thought to be unacceptable, however it was later discovered that the body of the radio would take the average of 50ms of received packets before acting and could tolerate up to 80ms of delay before shutting down [4, pg.21]. However, it was thought that the right choice was made as there was no way to guarantee that these performance issues would not escalate into a larger timing problem further into development. In addition a lower level language would give the application an opportunity to use less system resources.

Golang

Golang is a new compiled language created in 2007. It brings many newer ideas that interpreted languages have successfully used such as type inheritance and method overloading. Golang was designed to supplement the C programming language. One of the original authors of Golang is Ken Thompson (who is also one of the original authors of C). Its syntax is a combination of C and python, yet it is more strict than either. For example, including an unused import will induce a compiler error and endorses a definitive style guide instead of multiple competing standards.

Golang could be the ideal language to use in the future, but currently its standard library does not support serial communication. The most supported third party library is still in the alpha stage. This ruled out Golang over competing languages that already had a stable application programming interface (API) for such actions.

C

Other projects of similar nature have used C (including other ham radio drivers). C allows you to define a data structure with a very exact number of bytes.

```
//adapted from
 ↪ http://stackoverflow.com/questions/8584577/access-bits-in-a-char-in-c

//This tells the compiler to not put padding bits between values
 ↪ in the struct.
#pragma pack(1) //as we don't want space between our bits here
typedef struct {
        unsigned int data: 7;       // 7 more bits of the byte
        unsigned int check_num: 1; // Used to signify the first
         ↪ byte in the packet
} FT8900BYTE;

typedef union {
        FT8900BYTE section;
        unsigned char raw;
} PACKET_BYTE;
#pragma pack()
```

```
int main()
{
        PACKET_BYTE packet[42];
        return 0;
}
```

C also has serial support in its standard library with the include“<termios.h>”. While this is not as simple to use as other languages, it is the most comprehensive of them by far. Although “Pyserial” may have had a had better timeout implementation as it had a hard limit of seconds after the last bit instead of time between packets like the C implementation.

Comparison

Table 3.1 summarises the comparison of candidate languages. Despite lack of experience, C came out as superior by 11 points. When working in C, a quick check of how much involvement of strings there will be is necessary, as strings are a weakness of the language. Only the logging and shell would involve heavy use of strings, meaning that the bulk of the code would not be affected.

Language	experience	serial support	bitfield storage	portability	speed	total
Python	9	5	5	6	4	29
Golang	4	2	8	7	10	31
C	2	10	10	10	10	42

Table 3.1: Summary of languages comparison

Chapter 4

Implementation

An iteration time-span of two weeks was used, from the start of the project on 30th January, 2017 to the end of the second term 3rd April, 2017. This gave five two week long iterations. This decision is later reflected on in the evaluation.

4.1 Iteration 0

30th January, 2017 – 6th February, 2017

The zeroth iteration was the exploration phase of the project. The majority of the scratch programs and story (CRC) cards were written during this time. Assumptions from Ben Cooper's report [4] were verified by using an oscilloscope in a serial decoding mode. Also studied was the radio itself, including its service and user manuals [17].

As access to the radio was delayed for the first half of this week, focus was given to the possibility of integrating a solution into existing applications instead of a bespoke piece of software. The commonly used application for users that desire computer radio control was the application Hamlib [6]. The Hamlib documentation was studied and the developer mailing list utilised to discuss the suitability of a patch for the FT-8900. Hamlib provides a generic API for over 200 ham radios. It also has a client/server mode for TCP networking making remote control easier. In addition there are language bindings for C, C++, Perl and Python.

After some discussion with the developers of Hamlib it became apparent that the radio would require more than purely a USB cable to interface with the computer, and therefore it was not really something that should be in their codebase. They helpfully referenced many other projects that had been created for similar purposes on other radios. These were written in Python and Golang (hence their inclusion when deciding language choice in section 3.3). Before this decision the project was going to be a patch to Hamlib to add a new "backend".

4.2 Iteration 1

6th February, 2017 – 20th February, 2017

The first iteration could be considered the first "real" iteration as it involved actual software implementation. The initial Git commit was created and the overall file structure of the application was designed. For Makefiles CMake [18] was used, an open-source tool to generate the Makefiles. Cmake was first used because the test suite was sometimes an uncommon dependency on machines and CMake had the ability to download dependencies if needed. CMake also generates the necessary Makefiles based on the current state of libraries on the system. Later this system became invaluable to link and generate the multiple executables required.

This iteration was planned around developing the first feature, setting the radio frequency. However there was some way to go before this was possible. First, the bulk of "packet.h" had to be made to represent each 8 bit segment in a packet. An assumption was made that there are 8 bits in a byte. This will cost some portability to very old systems that do not allocate 8 bits to a byte. However, if this becomes an issue an alternate definition can be swapped in at compile time using a pre-processor rule that checks this the assumption is incorrect.

Additionally, the necessary constants that were expected in control packets were made, for example such as the default values and their allowed ranges. The end result of this iteration was that it could supply valid packets to the radio. This meant that it could be run without the original radio head attached. Before this, the radio would have automatically shutdown due to inactivity and while there was no way to do this in real-time, the packet could be configured to be "presssing" a button on the radio. While this iteration had not met its goal this was to be somewhat expected for the first feature. Firstly because setting the frequency was one of the most difficult features and secondly the beginnings of the application had to be created first. This feature was continued in the second iteration.

4.3 Iteration 2

20th February, 2017 – 6th March, 2017

For this iteration was there was temptation to start work on the most undeveloped part of the program, the display packet. So that it too could be represented like the control packet program. However the project stuck to its original stated goal of developing to set the frequency.

A packet sending thread was required to send the required stream of packets continuously. However work was first done on adding linting to the Jenkins CI to avoid potential bugs. As debugging threads is often a major pain point in programming, therefore the project should attempt to avoid this by testing often. This had an immediate pay off by finding a number of potential bugs within the application.

It was as this point the worst bug of the project was encountered. The effect was a pointer getting mangled thereby causing a segfault in a completely unrelated library that was used in the main application. This stemmed from setting, but then not resetting a compiler option (pragma_pack) in the program. This changes the amount of padding in memory between blocks of used memory. It was likely that the library relied on the completely opposite assumption of how much padding

there was between memory. While unit tests did not help find the bug they helped to rule out a large portion of the codebase as a culprit.

For the packet sending thread, a linked list was used as a way to queue packets to be sent to the radio. In C it is often common to make intrusive data structures. A intrusive data structure is when extra fields inside the struct are used to help with the the queue implementation. In this case that would mean placing pointers to the next and previous nodes in the queue within the control packet struct itself. The non-intrusive alternative is to have a second struct that holds a pointer to the control packet and along with the information to the next item in the list. The benefits of an intrusive data structure is that memory allocation is only done once (with Malloc as there is only one struct instead of two). At a large scale, unnecessary use of Malloc could have a significant impact on speed [19], with the trade off being that the structure is now less easy to use in another linked list implementation.

As this queue was to be used in a library it is possible that other programmers would not want to use the sender queue. Instead they may wish to want to represent control packets as compactly as possible. In addition, the current actions through the user shell did not invoke more than a few Malloc calls at a time. Therefore, the performance benefits were negligible, so the non-intrusive method was chosen. If in the future, if performance becomes an issue then this would be something to consider. However, currently this would be unnecessary pre-optimisation.

After some manual testing of the application, the radio was found to not respond to keypresses that were for less than 50ms. After some investigation this was likely due to software accounting bouncing of the voltages from the buttons. There is some noise created temporally by the circuits when the button moved, so the program instead takes the average reading inside 50ms. The radio did this no matter how many packets were send inside this time. Therefore it was concluded that it was useless to send packets as fast as the head does and instead added a longer waiting period between packets. This lowered the CPU usage of the program by 90%. At the end of this iteration the application was able to create a stream of packets that upon user request would successfully dial the requested numbers into the radio.

4.4 Iteration 3

6th March, 2017 – 20th March, 2017

Iteration 3 focused on users setting the volume, push-to-talk (PTT) and Squelch controls. These features were bundled together due to there similar implementation as they only required updating a single packet. Both left and right receivers can be set independently. Input is then validated to be within the expected range of the radio (and is tested accordingly). If a user gives a value out of range the closest valid number is used instead.

Logging output levels were added so that output could be filtered by importance, otherwise the output of the program was too verbose, making reading output impossible for regular users of the system. A wrapper function for “vprintf()” was made that took an extra Enum of logging level constants. The desired logging level can then be set with the -v flag. The use of Syslog was considered, but it was concluded that this would not be very useful for most users of the system. This is because they would have to configure the logging levels themselves in a separate system configuration file.

```
#define BIT_LOCATED_AT(byte, bit) (((byte) * 8) + (bit))

enum display_packet_bitmasks {
        //Misc
        AUTO_POWER_OFF = BIT_LOCATED_AT(0, 1),
        KEYPAD_LOCK    = BIT_LOCATED_AT(30, 4),
        SET            = BIT_LOCATED_AT(4, 0)
}
```

Figure 4.1: An example of 3 of the 190 mappings used in the display packet.

At this stage a rudimentary prompt was created a for users to interact with the program in real-time. Initially this was done very poorly, as a large case statement of string comparisons. This was done as this was the fastest way to implement the current needed features, and a project demonstration was at the end of the iteration. This reduced the readability and robustness of the code so was revisited latter in iteration 4. This code debt was arguably a good investment as users were able to see the intention of the final product and provide meaningful feedback early.

4.5 Iteration 4

20th March, 2017 – 3rd April, 2017

This Iteration focused on features that required information displayed on the screen. However a major refactor of the library was undertaken so that it could be properly included through a single header file. Before this, the library was included in the main application through its C files, meaning that the main program compiled its own version of the library. The end result was a much simpler dependency tree.

Reading the frequency display packets first involved mapping out all the bits in the packet used for each digit. The actual mapping process had been done by an unknown author and posted to the Internet [5]. This was complex as the required sequences of bits that were scattered throughout the packet. A large number of constants were defined for these mappings so that the functions that used them remained readable. These constants were the position of the bit in the overall packet. To make reading of this code essayer, a pre-processor macro (Shown in figure 4.1) was devised so that the developer could list the position using the bit and byte number.

After the bits had been retrieved they had to then be decoded back into the digit that they represented on the screen. A 13x10 truth table was devised to decode the 13 segment displays that represent the 10 digits. The radio will sometimes display letters here under some configurations in order to show station names (named by the user). Decoding this was deemed unnecessary, as the system only needed to read the frequencies. If letters are shown on the screen -1 is returned to signify an error.

Other features such as discerning which receiver was selected as the “main”, reading the set power level and if each receiver was “busy” were implemented. These functions simply checked a single bit in the display packets so were trivial to implement.

When testing this feature there were a number of behaviours of the radio to account for. Radio

#	A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	1	1	1	1	1	0	0	1	0	1	1	1
1	1	0	1	0	0	0	0	0	0	0	0	0	0
2	0	1	1	0	1	1	0	0	1	0	0	1	0
3	1	1	1	0	1	1	0	0	1	0	0	0	0
4	1	1	1	0	0	1	0	0	0	0	0	0	1
5	1	1	0	0	1	1	0	0	1	0	0	0	1
6	1	1	0	0	1	1	0	0	1	0	0	1	1
7	1	0	1	0	1	0	0	0	0	0	0	0	0
8	1	1	1	0	1	1	0	0	1	0	0	1	1
9	1	1	1	0	1	1	0	0	1	0	0	0	1

Table 4.1: A Truth table that decodes the 13 bit segment display to their corresponding numbers. This had to be made as it differs from generic segmented displays.

transmission is only permitted on a subset of the frequencies that it is capable of receiving on. To account for this frequency validation on user input was implemented. The application had to be aware if the radio was currently permitted to transmit on a given frequency.

Frequency validation had not only to be added to the frequency functions but also the PTT and Power commands. This was an undefined behaviour that was not in the user manual [17] of the radio. The power level cannot be changed when on frequency that does not allow transmission. Furthermore the radio does not notify the user in the usual way, an error sound followed with the text “Error” written on the screen. Instead the radio does nothing when the button is pressed. The only workaround for this was to check if the current frequency allowed transmission before setting power levels. Users are given a warning message with an explanation if this was not the case.

The next feature was to be able to boot the radio from the software. This can only be done by connecting the 5th line on the serial connector cable (the power switch line) to the 3rd line (ground). A signal that could be sent via the Data Terminal Ready (DTR) pin was implemented so that when given the “–dtr-on” flag, the application attempts to turn on the radio. If after 3 failed attempts the radio has not turned on, then the application shuts down. This was implemented in software, however was partly abandoned due to the the increase in complexity of hardware required. The user would require extra circuitry to read this signal and ground the pins. It may be possible to achieve this using a single MOSFET to complete the circuit when the DTR is set to high.

The rest of the time was used to refactor the shell. Each command was extracted into its own function instead of one massive case statement. An array of structs was created that held a pointer to that function, the function keyword, the number of arguments and help text. The shell then iterated through the array to match with the keyword and number of required arguments. This implementation made the shell very modular and also allowed for meta functions such as the help command to be easily added and implemented.

Chapter 5

Testing

5.1 Approach

Testing is an extremely important element in developing any application. The approach used for this project used automated unit tests, static code analysis and linting tests. For manual testing, acceptance tests on the completion of a feature were performed before merging into the main branch. Additionally, user tests of the whole system and memory inspection tests have been performed using the profiler Valgrind [20].

5.2 Automated Testing

A series of tests have been automated for the project. These tests are run on new commits pushed to the central repository. This is done using a webhook from Github to notify the automation platform Jenkins to run a new build. Jenkins [12] will then pull new changes from the repository for testing.

The project is first tested by making sure it compiles without errors or any major warnings. The build time is recorded so that the developer can later see what changes had the largest impact and take appropriate action if required. Next the application unit tests are run with the results being output to XML files that are then read by Jenkins. These results are then formatted to show the number of passing and failing tests over time. Finally Cppcheck [21] is run. This looks for common problems and mistakes in code such as unfreed memory or use of non-portable syntax and functions, such as syntax exclusive to GCC. Finally the results are fed back to be displayed on Github and as well as on a live dashboard (see figure 5.1).

5.2.1 Unit Tests

The project utilised unit tests throughout development (See appendix 3.2 for test output). This was an invaluable tool not only when re-factoring but allowed development of the application outside of the lab where the radio was held while still being able to test the program. Unit tests were primarily used to test the expected use-case as well as a number of important edge cases. For example if the pointer given was a null pointer the function should not de-reference it (otherwise

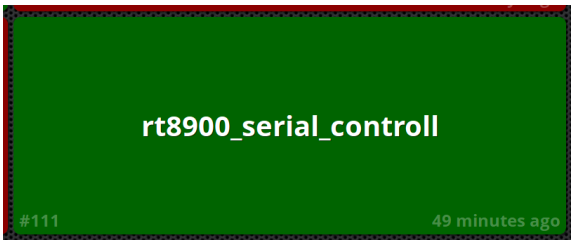


Figure 5.1: A screenshot of the CI dashboard. This can be left open on a screen to track the build status. On failed builds this would turn red with a reason specified.

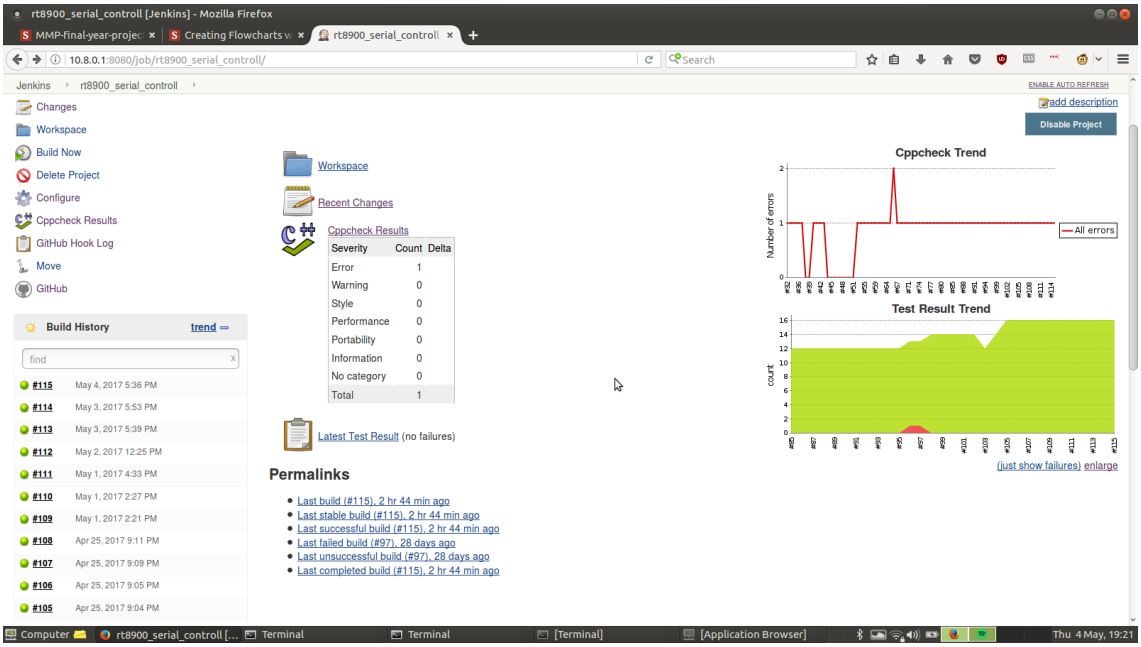


Figure 5.2: Jenkins showing test and error trends over time.

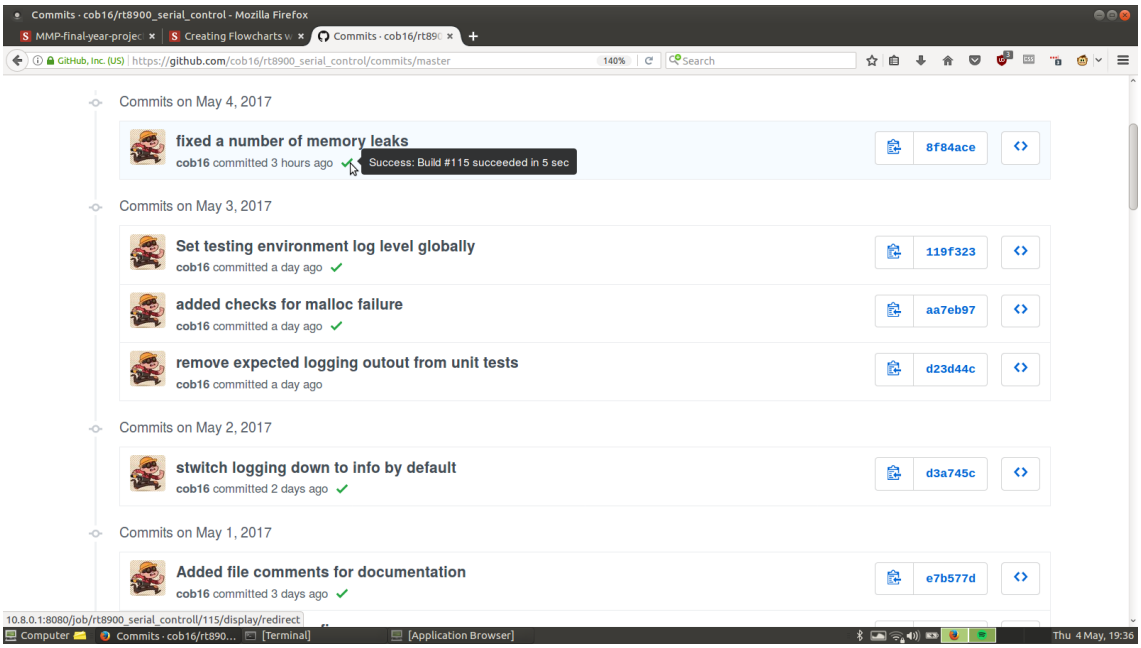


Figure 5.3: Github commit list showing a tick on successful builds of each commit. This is useful historical information and is invaluable in preventing merges of unstable branches.

the program would crash).

5.3 Manual Testing

Manual testing of the user shell was done by testing behaviour when expected and unexpected input was given. This was used to improve the user experience until a reasonably robust system had been made. User input is checked for the correct number of keywords and arguments. Arguments are safely cast to integers and range checked before being passed on to their corresponding functions. The input buffer will dynamically relocate to fit the input, up to a hard limit. The interrupt signal is handled from with in the application so that the program can be shutdown gracefully at any time.

The application was also tested on a number of releases by watching a user with no prior experience with the system. Their feedback was accounted for by re-prioritising features as well as making tweaks to the interface to better meet users expectations. These tests were combined by compiling and running the program on a separate system (see figure 5.4)to that used during development to ensure that all necessary files were included in Git.

Finally acceptance tests were conducted. Commands were checked to make sure their function worked as expected. This was done by cross-referencing the expected output with the output of the screen on the radio. The results of this are displayed in table 5.1.

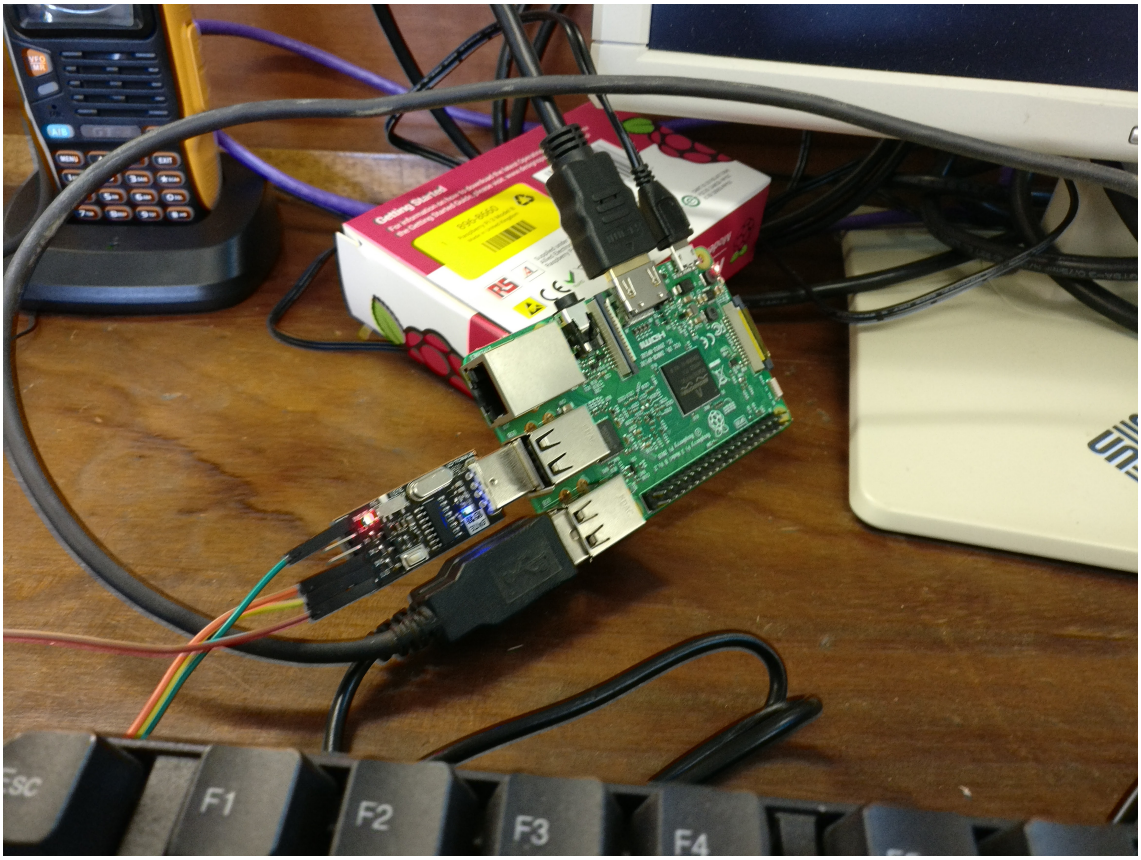


Figure 5.4: A Raspberry Pi running the application

Feature	Acceptance test	Status
Frequency	As a user I can set any valid frequency I desire for both VFO's of the radio. I can also get the current set frequencies from the radio.	Complete
Push to talk	As a user I am able to transmit on frequencies that the radio permits. I am also able to check if the radio is transmitting.	Complete
Volume	As a user I want to set the volumes of the radios independently.	Complete
Squelch	As a user I want to customise the squelch levels of the radio to reduce the amount of background noise on my radio.	Complete
Power	As a user I want to set the transmission power of each the radio.	Complete
Power on the radio	As a user I want the radio to automatically turn on when the application starts.	Incomplete

Table 5.1: Table of Acceptance tests and outcome

5.3.1 Memory profiling

Valgrind [20] was used to check for memory leaks in the developed program using the following command.

```
valgrind --trace-children=yes --leak-check=full ./main/rt8900c
 ↪  -v5 /dev/ttyUSB0
```

This was an invaluable tool that found 4 leaks in the application. Its output also lists which function the memory was allocated to, making fixes easy. Massif was also used to measure the amount of memory that the application used (See figure 5.5). The peak memory usage of the application is 3.6 KiB. This peak occurs when a frequency is being dialled due to the large use of the sending queue to press each button in sequence. This is more than adequate for the current use-case, allowing for even the smallest microcontrollers to run the application.

```
valgrind --tool=massif
 ↪  ./rt8900_serial_control/cmake-build-debug/main/rt8900c -v5
 ↪  /dev/ttyUSB0
```

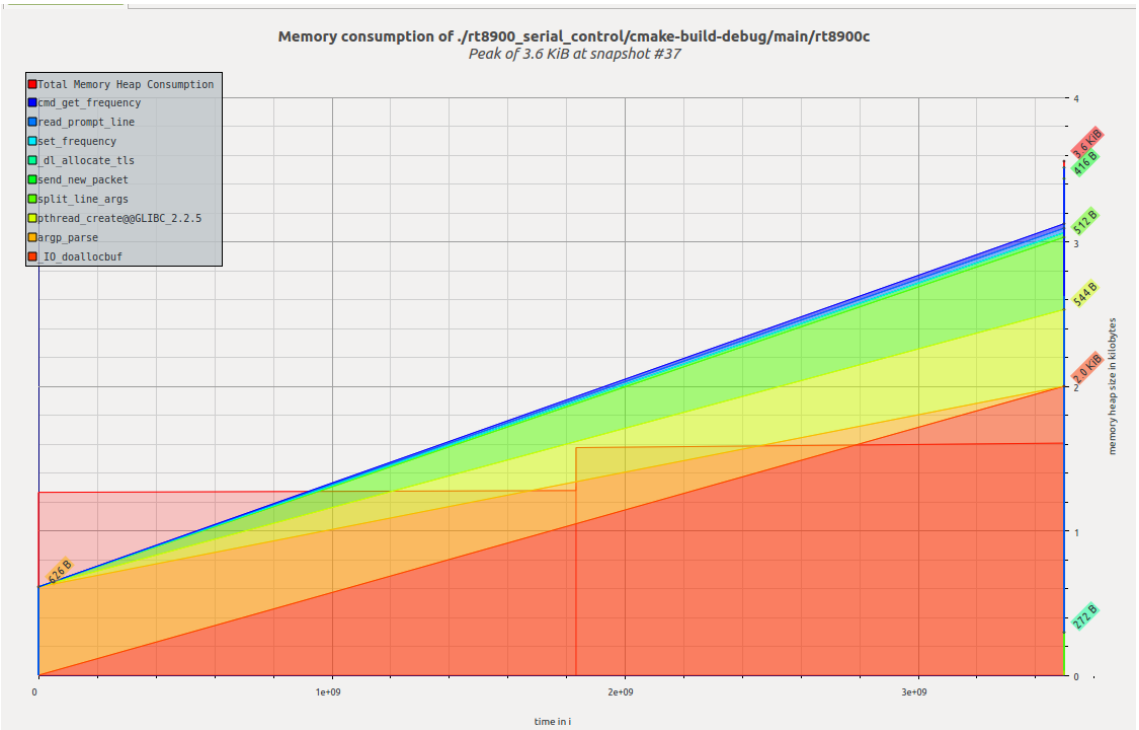


Figure 5.5: Memory usage of the application over time. This recording was done while the user tested every available function in the application.

5.4 Known issues

There were two known issues with the project at the time of submission. These issues were not deemed critical due to their current workarounds but are important improvements in development of the application.

Frequency validation

Frequency validation follows the given list of frequencies from the user manual [17]. However after testing, it was found that the right receiver supports only a subset of the frequencies that are listed. Further experimentation is required to discover this subset so that the validation can be correct for both receivers. This bug is limited in impact as upon receiving an invalid frequency the radio simply does nothing. In addition the user can see that the frequency was not changed from the output of the program.

Display packet reading

Many functions that read and write packets currently have built in delays in order to give the radio time to send out a new display packet and to be processed by the reader thread. The true amount of time was not tested, therefore some functions sleep for as much as a second before continuing. This could be improved by implementing a getter function that blocks until a new packet is received. The problem with this is that this provides a risk if a new packet is never received (as the function would then block forever). Ideally the program should try to recover by timing out and falling back to a last known safe packet. More research and analysis of solutions into reading the display packet would have helped to mitigate this.

Chapter 6

Evaluation

6.1 Design Reflection

The initial design of the application at the time was mostly adequate for the project. The structure of the program closely resembles what was planned in figure 3.7. It differs from the end result (See figure 6.2) in the use of some utility files such as “log.h” and the standard lib includes.

Greater research and design for the process of receiving and processing packets into the system could have been made. While the current system is adequate, its functions have to account for reading slightly delayed packets from the application buffer. This component of the system should be more efficient, as actions of the library will be expected to be as fast as possible, especially if the action is linked to a simple button in a GUI application.

Language	files	blank	comment	code
C	7	233	174	1341
C/C++ Header	10	141	100	521
C++	4	68	43	246
CMake	4	22	15	60
SUM:	25	464	332	2168

Figure 6.1: The project source code at the time of submission contains 130 commits on 2168 lines of code primarily written in C.

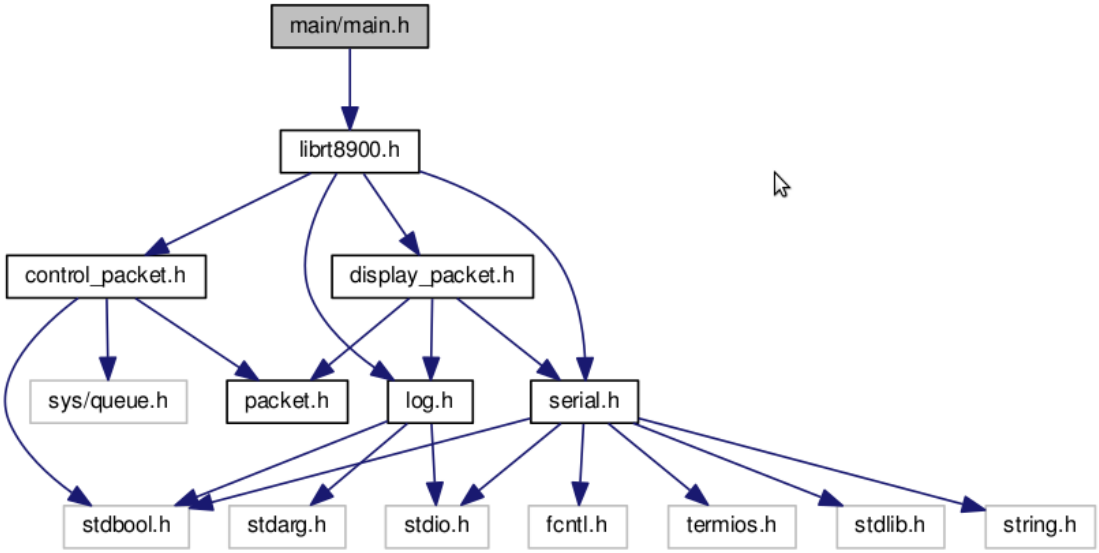


Figure 6.2: Graph of the entire application dependency tree. This includes all standard library usages. The graph differs from the designed architecture in the use of a number of utility files such as serial.h and log.h

6.2 Suitability of tools

C

C was the best choice for this project in order to work as closely with data structures and serial interfaces. The relatively small size and low use of strings in the project contributed to the suitability of using the C language. Initially, unit tests with C++ prohibited certain C11 functions, however this was remedied with proper library linking. Compilation was exceedingly quick. Tools where always included directly in Linux distribution package managers unlike many other languages such as Python that use their own.

Jenkins

Using Jenkins over some of the free online platforms was a superior choice as it provided a better amount of customisation for the builds. For example, the build was able to use executables such as Cppcheck and GCC from the host system. Using a dedicated system over a service gave faster builds as well as more reliable time metrics so the build time could be monitored effectively.

Git

Even for a single developer, the use of git helped maintain changes between computers. Fixes could occur in the lab while new features could be simultaneously developed on a separate branch for easy merges later. This helped fuel the ability to make quick changes without fear of versioning problems. The commit log helped to document changes, helping to review iterations for the blog posts as well as for this report.

6.3 Improvements

After the second iteration, the length of iteration cycle could have been shortened from the two week period down to one. This was evident in the later iterations, as these contained many more features, therefore they could be shortened so that the iterations remained more focused. A reduction down to a single week iteration was mistakenly forgone for the sake of consistency of iteration lengths.

The command line interface (CLI) was an expected feature, however it was an oversight to not provide an overall design for it. Stories and acceptance tests could have been made for these at the beginning of the project. This resulted in significant re-factoring in the last iteration in order to improve the overall code quality and maintainability of the shell. Better research and planning of a CLI as well as the the display packet reading thread earlier would have avoided difficulty in determining the testing latency later.

It was found out later that the targeted radio was the FT-8900r has a number of variants for each region. This is a problem as these models differ in their allowed frequencies for transmission. Currently only the Europe version is properly supported, with the US and Asia regions missing their configuration. This was a problem as it was difficult to source the frequency lists. In the future the likely solution would to have a number of lists for transmission that can be chosen via an optional flag to choose the model at runtime.

Unit tests were written primarily for library functions that manipulated packets. Testing of the actual user interface by capturing input and output of the program may have been a useful automation in order to decrease the time spent testing. During implementation this was not a priority due to how quick manual testing to the same end was. It can be foreseen however, that this would become more difficult in the future, as more and more features are added to the project.

The most useful development in the future would be to add an inter process communication method. The user shell would be moved out to a separate client that would talk to the server via a socket. This would allow other applications to utilise the socket FT-8900 control such as Hamlib with a small patch. This would also permit making the CLI in another language, which may have facilitated faster development.

6.4 Objectives

Objective	Status
Final report	Complete
Frequency	Complete
Push to talk	Complete
Volume	Complete
Squelch	Complete
Power	Complete
Powering on the radio	Partial (Completed in software)
Code documentation	Complete
Schematic	Complete

Table 6.1: List of objectives/deliverables

It is the opinion of the developer that the project aims have been met. Users of the application have been provided with a useful, portable and fast application to control their FT-8900 radios without the high cost of existing solutions. The hardware requirement of turning on the radio was only partially met. However a remote shack could still easily leave their radio on, ready for transmission (the radio will not consume much power when idle). For audio the data port at the back of the radio already provides this function. A good experience for a remote shack operator can be made with the use of a VOIP application such as Mumble [22] for audio and SSH to control the project application.

Glossary

API application programming interface. 16

CAT Computer Aided Transceiver. 1, 2

CLI command line interface. 31

CRC Class, Responsibilities, and Collaboration. 8

Doxygen A tool that generates documentation from source code. 5

DTR Data Terminal Ready. 22

FDD feature driven development. 6

FT-8900 Yaesu FT-8900R. iii, 1, 2, 10, 18, 31, 32

linting A linter is any tool that flags suspicious usage in software.. 19, 23

minimum viable product Enough features to satisfy early adopters and start getting feedback from initial users.. iii

MOSFET MOSFET stands for metalâĂŞoxideâĂŞsemiconductor field-effect transistor. This can be used as a digital switch that is opened and closed when a signal voltage is given to its gate pin.. 22

PTT push-to-talk. 20, 22

RGR Red, Green, Refactor. 8

RX serial transmission from the radio. 3, 5, 8, 10, 13

Squelch Mutes the radio until a strong enough signal is detected. This has the effect of hiding background “white noise” when there is no incoming signal.. 5, 20

TTL Transistor to transistor logic. 3, 4

TX serial transmission to the radio. 3–5, 8, 10, 13

XP extreme programming. 6, 8, 13

Appendices

Appendix A

Third-Party Code and Libraries

Google Test

The unit tests for this project where written in the Google test framework [11] without modification. This project is under the 2-Clause BSD Licenece and can be found at: `https://github.com/google/googletest/blob/master/googletest/LICENSE`.

Queue.h

The <sys/queue.h> library is technically not part of the standard library (only de-facto). It was use to create a First in last out queue for the sender thread. This was used as it was available on targeted platforms therefore there was no requirement to re-implement. It is licenced under the 3-Clause BSD and can be found at `http://man7.org/linux/man-pages/man3/queue.3.license.html`.

Doxygen

A documentation generator that parses source code to produce output in HTML or latex (among others). Licenced under the GPL v2. This can be found at `https://www.gnu.org/licenses/old-licenses/gpl-2.0.html`.

Appendix B

Ethics Submission

AU Status
Undergraduate or PG Taught

Your aber.ac.uk email address
cob16@aber.ac.uk

Full Name
cormac brady

Please enter the name of the person responsible for reviewing your assessment.
Reyer Zwiggelaar

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment
rrz@aber.ac.uk

Supervisor or Institute Director of Research Department
cs

Module code (Only enter if you have been asked to do so)
CS39440

Proposed Study Title
Computer control of the Yaesu FT-8900R

Proposed Start Date
30/01/2017

Proposed Completion Date
08/05/2017

Are you conducting a quantitative or qualitative research project?
Mixed Methods

Does your research require external ethical approval under the Health Research Authority?
No

Does your research involve animals?
No

Are you completing this form for your own research?
Yes

Does your research involve human participants?
No

Institute
IMPACS

Please provide a brief summary of your project (150 word max)
Many modern transceivers are designed to be controllable by by a computer. The computer being able to operate all the controls and also provide information/data for transmission/reception by the radio. We own two good quality Yaesu FT8900 radios. "Officially" these are not fully computer controllable, you can exchange data but cannot do actual control. The control panel of the radio is attached by a cable and designed so the physical controls can be mounted elsewhere. Some open source projects have at least partially 'reverse engineered' the panel to radio cable and control protocol. This project, which might build upon the existing open source work, will design and implement a way for computers to control the radios. This could be

developed and tested by any student (without an amateur license) by attaching a 'dummy load' rather than an antenna. A student with an amateur license can do more and amateur license training starts on Wednesday 28th October for thos interested.

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?
Yes

Will appropriate measures be put in place for the secure and confidential storage of data?
Yes

Does the research pose more than minimal and predictable risk to the researcher?
No

Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to?
No

Please include any further relevant information for this section here:

If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one.
Yes

Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and that you will inform your department should the proposal significantly change.
Yes

Please include any further relevant information for this section here:

Appendix C

Code Examples

3.1 Python Bitfield

```
from enum import Enum
import bits_mod

"""
would use 'pyserial' and bits_mod (or more useful rewrite)
This allows us to store the packet inside a long.
Perhaps the most dense data structure for this problem.
"""


class EndBit(Enum):
    """This ENUM defines what the most significant bit should
     ↪ be.
    The first byte should end with 1 else 0"""
    FIRST_BYTE = 1
    OTHER_BYTE = 0


class PacketByte:
    """This class allows easy manipulation, conversion and
     ↪ handling of our
    packet bytes"""

    def __init__(self, data_bytes, endbit):
        if endbit in EndBit:
            self._endbit = endbit

         # this will through ValueError back if it fails
        self.data_bytes = int(data_bytes, 2)
```

```
packet = bits_mod.Bits(104)


def packet_range(start, stop):
    """Range wrapper that auto works in reverse"""
    if start <= stop:
        return list(range(start, stop))
    else:
        return list(range(start, stop, -1))


#how map coule be represented
packet_map = {
    'left_encoder': packet_range(0, 7),
    'right_encoder': packet_range(8, 15),
    'well_made_function': [2, 44, 45] + packet_range(90, 104),
}

for i in packet_map['left_encoder']:
    if i % 2 != 0:
        packet.mark(i)


def get_bits(array_of_indexes):
    bits = b''
    for i in array_of_indexes:
        if packet.is_true(i):
            bits += b'1'
        else:
            bits += b'0'
    return bits


print(get_bits(packet_map['left_encoder'])) #get related bytes


def test_mapping_ranges(packet_map):
    expected_range = set(range(0, 104))
    actual_range = set()
    for i in packet_map.values():

        # see if there are overlapping mappings
        if actual_range.intersection() is None:
            actual_range.add(i)
        else:
            pass #fail out as there are overlapping maps
    if set(range(0, 104)).difference(actual_range) is not None:
pass # fail as there are missing mappings
```

3.2 Unit test output

```
[==========] Running 18 tests from 6 test cases.
[----------] Global test environment set-up.
[----------] 4 tests from TestDisplayPacket
[ RUN      ] TestDisplayPacket.test_find_packet_start
[       OK ] TestDisplayPacket.test_find_packet_start (0 ms)
[ RUN      ] TestDisplayPacket.test_shift_array
[       OK ] TestDisplayPacket.test_shift_array (0 ms)
[ RUN      ] TestDisplayPacket.test_get_range
[       OK ] TestDisplayPacket.test_get_range (0 ms)
[ RUN      ] TestDisplayPacket.test_operational_range
[       OK ] TestDisplayPacket.test_operational_range (0 ms)
[----------] 4 tests from TestDisplayPacket (0 ms total)

[----------] 3 tests from TestDisplayPacketReaders
[ RUN      ] TestDisplayPacketReaders.test_read_busy
[       OK ] TestDisplayPacketReaders.test_read_busy (0 ms)
[ RUN      ] TestDisplayPacketReaders.test_packet_read
[       OK ] TestDisplayPacketReaders.test_packet_read (0 ms)
[ RUN      ] TestDisplayPacketReaders.test_read_14_seg
[       OK ] TestDisplayPacketReaders.test_read_14_seg (0 ms)
[----------] 3 tests from TestDisplayPacketReaders (0 ms total)

[----------] 2 tests from Librt8900Test
[ RUN      ] Librt8900Test.test_in_freq_range
[       OK ] Librt8900Test.test_in_freq_range (0 ms)
[ RUN      ] Librt8900Test.test_current_freq_valid
[       OK ] Librt8900Test.test_current_freq_valid (0 ms)
[----------] 2 tests from Librt8900Test (0 ms total)

[----------] 2 tests from ControlPacketTest
[ RUN      ] ControlPacketTest.PACKET_BYTE
[       OK ] ControlPacketTest.PACKET_BYTE (0 ms)
[ RUN      ] ControlPacketTest.CONTROL_PACKET
[       OK ] ControlPacketTest.CONTROL_PACKET (0 ms)
[----------] 2 tests from ControlPacketTest (0 ms total)

[----------] 2 tests from TestKeypadButtons
[ RUN      ] TestKeypadButtons.test_set_button
[       OK ] TestKeypadButtons.test_set_button (0 ms)
[ RUN      ] TestKeypadButtons.test_button_from_int
[       OK ] TestKeypadButtons.test_button_from_int (0 ms)
[----------] 2 tests from TestKeypadButtons (0 ms total)

[----------] 5 tests from TestAPISetters
[ RUN      ] TestAPISetters.test_safe_int_char
```

```
[       OK ] TestAPISetters.test_safe_int_char (0 ms)
[ RUN      ] TestAPISetters.test_set_L_R_volume
[       OK ] TestAPISetters.test_set_L_R_volume (0 ms)
[ RUN      ] TestAPISetters.test_get_L_R_volume
[       OK ] TestAPISetters.test_get_L_R_volume (0 ms)
[ RUN      ] TestAPISetters.test_set_L_R_squelch
[       OK ] TestAPISetters.test_set_L_R_squelch (0 ms)
[ RUN      ] TestAPISetters.test_get_L_R_squelch
[       OK ] TestAPISetters.test_get_L_R_squelch (0 ms)
[----------] 5 tests from TestAPISetters (0 ms total)

[----------] Global test environment tear-down
[==========] 18 tests from 6 test cases ran. (1 ms total)
[  PASSED  ] 18 tests.
```

Appendix D

Code Documentation

This contains a copy of the auto generated code documentation. Some sections are very useful to consult, however this print version is less so than the HTML version. The HTML version can attained ether by flowing the README.md instructions or by navigating to:

`https://cormacbrady.info/rt8900_docs/`

Rt-8900 Serial Control

0.3.0

Generated by Doxygen 1.8.11

Contents

1 Main Page 1

2 Data Structure Index 3

2.1 Data Structures . . . 3

3 File Index 5

3.1 File List . . . 5

4 Data Structure Documentation 7

4.1 button_transmit_value Struct Reference . . . 7

4.2 cmd Struct Reference . . . 7

4.3 control_packet Struct Reference . . . 8

4.4 CONTROL_PACKET_INDEXED Union Reference . . . 9

4.4.1 Detailed Description . . . 9

4.5 control_packet_q_node Struct Reference . . . 10

4.5.1 Detailed Description . . . 11

4.6 control_packet_sender_config Struct Reference . . . 11

4.6.1 Detailed Description . . . 11

4.7 display_packet_receiving_config Struct Reference . . . 11

4.7.1 Detailed Description . . . 12

4.8 FT8900BYTE Struct Reference . . . 12

4.9 PACKET_BYTE Union Reference . . . 12

4.9.1 Detailed Description . . . 13

4.10 radio_side Struct Reference . . . 13

4.11 radio_state Struct Reference . . . 13

4.12 range_KHz Struct Reference . . . 14

4.12.1 Detailed Description . . . 14

4.13 SERIAL_CFG Struct Reference . . . 14

5 File Documentation **15**

5.1 librt8900/control_packet.h File Reference . . . 15

5.1.1 Detailed Description . . . 18

5.1.2 Function Documentation . . . 18

5.1.2.1 safe_int_char(int number) . . . 18

5.1.2.2 send_control_packets(void *c) . . . 18

5.1.2.3 set_squelch(struct control_packet *packet, int left, int right) . . . 18

5.1.2.4 set_squelch_left(struct control_packet *packet, int number) . . . 18

5.1.2.5 set_squelch_right(struct control_packet *packet, int number) . . . 19

5.2 librt8900/display_packet.h File Reference . . . 19

5.2.1 Detailed Description . . . 22

5.2.2 Function Documentation . . . 22

5.2.2.1 insert_shifted_packet(DISPLAY_PACKET packet, unsigned char buffer[], size_t buffer_length, int start_of_packet_index) . . . 22

5.2.2.2 is_main(struct radio_state *radio, struct radio_side *side) . . . 22

5.2.2.3 read_frequency(DISPLAY_PACKET packet, struct radio_state *state) . . . 22

5.2.2.4 read_main(DISPLAY_PACKET packet, struct radio_state *state) . . . 23

5.2.2.5 segment_to_int(int segment_bitmask) . . . 23

5.3 librt8900/librt8900.h File Reference . . . 23

5.3.1 Detailed Description . . . 25

5.3.2 Function Documentation . . . 25

5.3.2.1 check_radio_rx(SERIAL_CFG *config) . . . 25

5.3.2.2 current_freq_valid(struct radio_side *radio) . . . 25

5.3.2.3 get_display_packet(SERIAL_CFG *config, DISPLAY_PACKET packet) . . . 25

5.3.2.4 get_range(int frequency_khz) . . . 25

5.3.2.5 in_freq_range(int frequency_khz) . . . 25

5.3.2.6 receive_display_packets(void *c) . . . 25

5.3.2.7 send_control_packets(void *c) . . . 26

5.3.2.8 send_new_packet(SERIAL_CFG *config, struct control_packet *new_packet, enum pop_queue_behaviour free_choice) . . . 26

5.3.2.9 set_frequency(SERIAL_CFG *cfg, struct control_packet *base_packet, int number) 26

5.3.2.10 set_left_power_level(SERIAL_CFG *cfg, struct control_packet *base_packet, enum rt8900_power_level power_level) . . . 26

5.3.2.11 set_main_radio(SERIAL_CFG *cfg, struct control_packet *base_packet, enum radios side) . . . 26

5.3.2.12 set_right_power_level(SERIAL_CFG *cfg, struct control_packet *base_packet, enum rt8900_power_level power_level) . . . 26

5.3.2.13 shutdown_threads(SERIAL_CFG *cfg) . . . 27

5.4 librt8900/log.h File Reference . . . 27

5.4.1 Detailed Description . . . 28

5.4.2 Function Documentation . . . 28

5.4.2.1 log_msg(enum rt8900_logging_level level, char const *fmt,...) . . . 28

5.5 librt8900/packet.h File Reference . . . 28

5.5.1 Detailed Description . . . 29

5.5.2 Enumeration Type Documentation . . . 30

5.5.2.1 check_num_values . . . 30

5.5.3 Function Documentation . . . 30

5.5.3.1 find_packet_start(unsigned char buffer[], size_t length) . . . 30

5.6 librt8900/serial.h File Reference . . . 30

5.6.1 Detailed Description . . . 31

5.7 main/main.h File Reference . . . 32

5.7.1 Detailed Description . . . 32

Index **33**

Chapter 1

Main Page

#FT8900 Controller Provides serial control for the YAESU FT-8900R Transceiver.

##Usage

```
Usage: rt8900c [OPTION...] <serial port path>
Provides serial control for the YAESU FT-8900R Transceiver.

  -d, --dtr-on               Use the DTR pin of the serial connection as a
                             power button for the rig. (REQUIRES compatible
                             hardware)
      --hard-emulation       Exactly emulates the radio head instead of being
                             lazy_sending (worse performance, no observed
                             benefit, only useful for debugging)
  -v, --verbose[=LEVEL]       Produce verbose output add a number to select
                              level (1 = ERROR, 2= WARNING, 3=INFO, 4=ERROR,
                              5=DEBUG) output default is 'warning'.
  -?, --help                  Give this help list
      --usage                 Give a short usage message
  -V, --version               Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Report bugs to <cormac.brady@hotmail.co.uk>.
```

Build and Install

For ubuntu but can be adapted for other distribution's and OS

```
sudo apt install cmake git build-essential
git clone <this repo url> rt8900c
cd rt8900c
cmake .
make
```

##Run tests

```
cmake .
make test

#for verbose output
./test/test_librt8900/test_librt8900
```

Credits

- CmakeLists build files taken from `this example`

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

button_transmit_value . . . 7
cmd . . . 7
control_packet . . . 8
CONTROL_PACKET_INDEXED
    Used to get the packet as an array . . . 9
control_packet_q_node
    A internal, non intgrated queue struct for the packet queue . . . 10
control_packet_sender_config
    Configuration for sending packets . . . 11
display_packet_receiving_config
    Configuration for receiving packets . . . 11
FT8900BYTE . . . 12
PACKET_BYTE
    Used to store 1 byte of data from the packet . . . 12
radio_side . . . 13
radio_state . . . 13
range_KHz
    Represents one of the capable ranges of the radio . . . 14
SERIAL_CFG . . . 14

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

librt8900/control_packet.h 15
librt8900/display_packet.h 19
librt8900/librt8900.h
The main headder file for the librt8900 libary. Contains functions that use both the contol_packet and DISPLAY_PACKET 23
librt8900/log.h
Logging wrapper with levels for the librt8900 library 27
librt8900/packet.h
Data representation of a single byte in a packet 28
librt8900/serial.h
Serial handling 30
main/main.h 32

Chapter 4

Data Structure Documentation

4.1 button_transmit_value Struct Reference

Data Fields

- signed char **row**
- signed char **column**

The documentation for this struct was generated from the following file:

- librt8900/control_packet.h

4.2 cmd Struct Reference

Data Fields

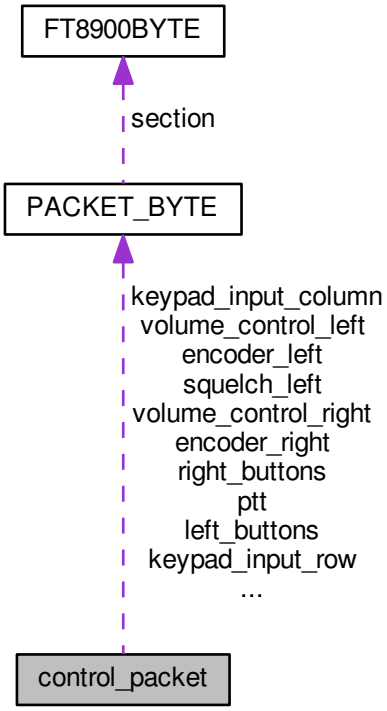
- int(∗ **cmd_pointer**)(char ∗∗args, SERIAL_CFG ∗config, struct control_packet ∗base_packet)
- char ∗ **keyword**
- char ∗ **discription**
- char ∗ **usage**
- int **num_args**

The documentation for this struct was generated from the following file:

- main/main.c

4.3 control_packet Struct Reference

Collaboration diagram for control_packet:



Data Fields

- PACKET_BYTE **encoder_right**
- PACKET_BYTE **encoder_left**
- PACKET_BYTE **ptt**
- PACKET_BYTE **squelch_right**
- PACKET_BYTE **volume_control_right**
- PACKET_BYTE **keypad_input_row**
- PACKET_BYTE **volume_control_left**
- PACKET_BYTE **squelch_left**
- PACKET_BYTE **keypad_input_column**
- PACKET_BYTE **left_buttons**
- PACKET_BYTE **right_buttons**
- PACKET_BYTE **main_buttons**
- PACKET_BYTE **hyper_mem_buttons**

The documentation for this struct was generated from the following file:

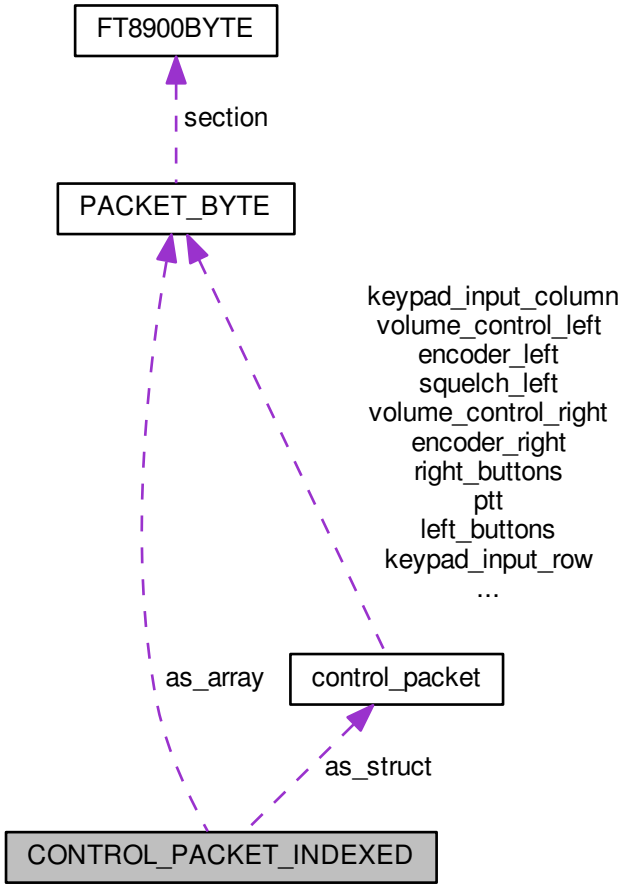
- librt8900/control_packet.h

4.4 CONTROL_PACKET_INDEXED Union Reference

used to get the packet as an array

`#include <control_packet.h>`

Collaboration diagram for CONTROL_PACKET_INDEXED:



Data Fields

- struct control_packet **as_struct**
- PACKET_BYTE **as_array** [13]

4.4.1 Detailed Description

used to get the packet as an array

The documentation for this union was generated from the following file:

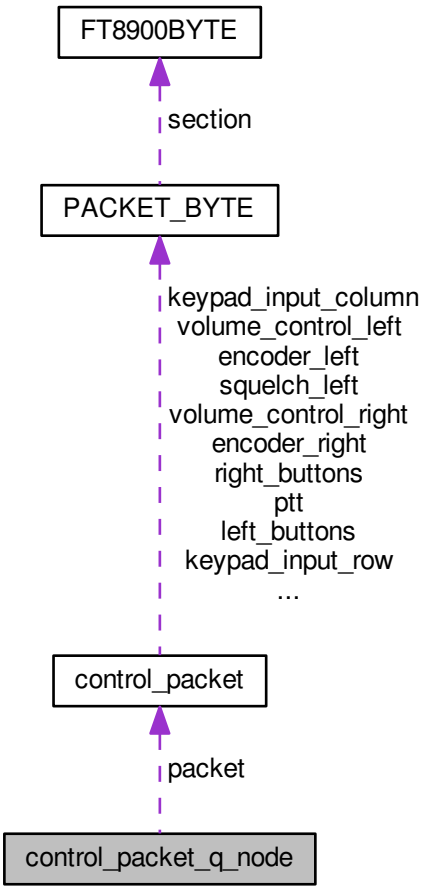
- librt8900/control_packet.h

4.5 control_packet_q_node Struct Reference

A internal, non intgrated queue struct for the packet queue.

`#include <control_packet.h>`

Collaboration diagram for control_packet_q_node:



Public Member Functions

- **TAILQ_ENTRY** (control_packet_q_node) nodes

Data Fields

- struct control_packet * **packet**
- enum pop_queue_behaviour **free_packet**

4.5.1 Detailed Description

A internal, non intgrated queue struct for the packet queue.

The documentation for this struct was generated from the following file:

- librt8900/control_packet.h

4.6 control_packet_sender_config Struct Reference

Configuration for sending packets.

```
#include <librt8900.h>
```

Data Fields

- bool **lazy_sending**
- bool **dtr_pin_for_on**
- pthread_barrier_t * **initialised**
- struct CONTROL_PACKET_Q_HEAD * **queue**
- bool **keep_alive**

4.6.1 Detailed Description

Configuration for sending packets.

The documentation for this struct was generated from the following file:

- librt8900/librt8900.h

4.7 display_packet_receiving_config Struct Reference

Configuration for receiving packets.

```
#include <librt8900.h>
```

Data Fields

- bool **keep_alive**
- bool **radio_seen**
- pthread_mutex_t **raw_packet_lock**
- unsigned char **latest_raw_packet** [DISPLAY_PACKET_SIZE]

4.7.1 Detailed Description

Configuration for receiving packets.

The documentation for this struct was generated from the following file:

- librt8900/librt8900.h

4.8 FT8900BYTE Struct Reference

Data Fields

- unsigned int **data**: 7
- unsigned int **check_num**: 1

The documentation for this struct was generated from the following file:

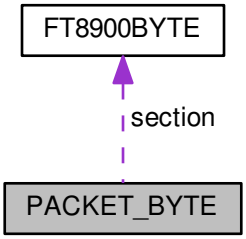
- librt8900/packet.h

4.9 PACKET_BYTE Union Reference

Used to store 1 byte of data from the packet.

`#include <packet.h>`

Collaboration diagram for PACKET_BYTE:



Data Fields

- FT8900BYTE **section**
- unsigned char **raw**

4.9.1 Detailed Description

Used to store 1 byte of data from the packet.

The documentation for this union was generated from the following file:

- librt8900/packet.h

4.10 radio_side Struct Reference

Data Fields

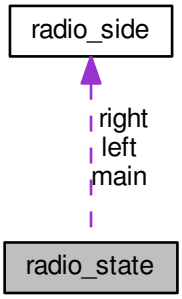
- int **busy**
- int **frequency**
- enum rt8900_power_level **power_level**

The documentation for this struct was generated from the following file:

- librt8900/display_packet.h

4.11 radio_state Struct Reference

Collaboration diagram for radio_state:



Data Fields

- struct radio_side * **main**
- struct radio_side **left**
- struct radio_side **right**

The documentation for this struct was generated from the following file:

- librt8900/display_packet.h

4.12 range_KHz Struct Reference

Represents one of the capable ranges of the radio.

`#include <librt8900.h>`

Data Fields

- const bool **tx_allowed**
- const char * **name**
- const int **low**
- const int **high**

4.12.1 Detailed Description

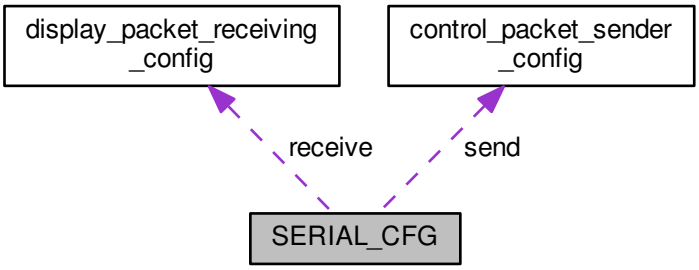
Represents one of the capable ranges of the radio.

The documentation for this struct was generated from the following file:

- librt8900/librt8900.h

4.13 SERIAL_CFG Struct Reference

Collaboration diagram for SERIAL_CFG:



Data Fields

- char * **serial_path**
- int **serial_fd**
- bool **shutdown_on_timeout**
- struct control_packet_sender_config **send**
- struct display_packet_receiving_config **receive**

The documentation for this struct was generated from the following file:

- librt8900/librt8900.h

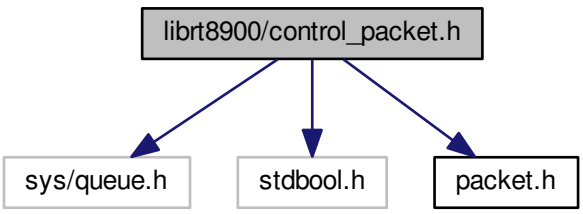
Chapter 5

File Documentation

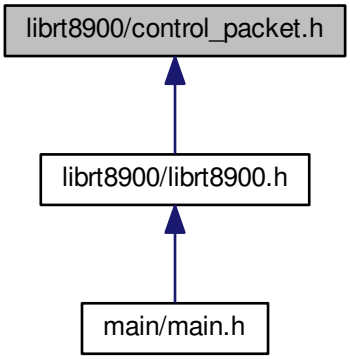
5.1 librt8900/control_packet.h File Reference

```
#include <sys/queue.h>
#include <stdbool.h>
#include "packet.h"
```

Include dependency graph for control_packet.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct control_packet
- struct button_transmit_value
- struct control_packet_q_node

  A internal, non intgrated queue struct for the packet queue.
- union CONTROL_PACKET_INDEXED

  used to get the packet as an array

Macros

- #define **MILLISECONDS_BETWEEN_PACKETS_STANDARD** 3
- #define **MILLISECONDS_DEBOUNCE_WAIT** 51
- #define **DEFAULT_VOLUME** 0x1f
- #define **maloc_control_packet**(pointer_name) struct control_packet ∗(pointer_name) = (struct control_packet∗) malloc(sizeof(∗(pointer_name)))
- #define **VD_INDEX_0** 0X00
- #define **VD_INDEX_1** 0X1A
- #define **VD_INDEX_2** 0X32
- #define **VD_INDEX_3** 0X4C
- #define **VD_INDEX_4** 0X64
- #define **VD_NONE** 0X7F
- #define **BUTTON_NONE_VALUE** {VD_NONE, VD_NONE}
- #define **BUTTON_1_VALUE** {VD_INDEX_0, VD_INDEX_1}
- #define **BUTTON_2_VALUE** {VD_INDEX_0, VD_INDEX_2}
- #define **BUTTON_3_VALUE** {VD_INDEX_0, VD_INDEX_3}
- #define **BUTTON_A_VALUE** {VD_INDEX_0, VD_INDEX_4}
- #define **BUTTON_UP_VALUE** {VD_INDEX_1, VD_INDEX_0}
- #define **BUTTON_4_VALUE** {VD_INDEX_1, VD_INDEX_1}
- #define **BUTTON_5_VALUE** {VD_INDEX_1, VD_INDEX_2}
- #define **BUTTON_6_VALUE** {VD_INDEX_1, VD_INDEX_3}
- #define **BUTTON_B_VALUE** {VD_INDEX_1, VD_INDEX_4}

- #define **BUTTON_DOWN_VALUE** {VD_INDEX_2, VD_INDEX_0}
- #define **BUTTON_7_VALUE** {VD_INDEX_2, VD_INDEX_1}
- #define **BUTTON_8_VALUE** {VD_INDEX_2, VD_INDEX_2}
- #define **BUTTON_9_VALUE** {VD_INDEX_2, VD_INDEX_3}
- #define **BUTTON_C_VALUE** {VD_INDEX_2, VD_INDEX_4}
- #define **BUTTON_X_VALUE** {VD_INDEX_3, VD_INDEX_1}
- #define **BUTTON_0_VALUE** {VD_INDEX_3, VD_INDEX_2}
- #define **BUTTON_HASH_VALUE** {VD_INDEX_3, VD_INDEX_3}
- #define **BUTTON_D_VALUE** {VD_INDEX_3, VD_INDEX_4}
- #define **BUTTON_P1_VALUE** {VD_INDEX_4, VD_INDEX_1}
- #define **BUTTON_P2_VALUE** {VD_INDEX_4, VD_INDEX_2}
- #define **BUTTON_P3_VALUE** {VD_INDEX_4, VD_INDEX_3}
- #define **BUTTON_P4_VALUE** {VD_INDEX_4, VD_INDEX_4}

Enumerations

- enum **main_menu_buttons** {
  NOT_PRESSED = 0, **R_ENCODER_BUTTON** = 1, **L_ENCODER_BUTTON** = (1 << 1), **SET_BUTTON** = (1 << 2),
  WIRES_BUTTON = (1 << 3) }
- enum **radios** { **RADIO_LEFT**, **RADIO_RIGHT** }
- enum **right_menu_buttons** {
  RIGHT_NONE = DATA_MAX_NUM, **RIGHT_LOW** = 0x00, **RIGHT_VM** = 0x20, **RIGHT_HM** = 0x40, **RIGHT_SCN** = 0x60 }
- enum **left_menu_buttons** {
  LEFT_NONE = DATA_MAX_NUM, **LEFT_LOW** = RIGHT_SCN, **LEFT_VM** = RIGHT_HM, **LEFT_HM** = RIGHT_VM,
  LEFT_SCN = RIGHT_LOW }
- enum pop_queue_behaviour { **PACKET_FREE_AFTER_SEND** = 0, **PACKET_ONLY_SEND** = 1 }

  Options on if the packet added to the queue will be freed once sent.

Functions

- typedef TAILQ_HEAD (CONTROL_PACKET_Q_HEAD, control_packet_q_node) CONTROL_PACKET_Q_HEAD

  Create our packet queue struct.
- void **set_keypad_button** (struct control_packet *packet, const struct button_transmit_value *button)
- void **set_main_button** (struct control_packet *packet, const enum main_menu_buttons button)
- void **set_left_button** (struct control_packet *packet, const enum left_menu_buttons button)
- void **set_right_button** (struct control_packet *packet, const enum right_menu_buttons button)
- const struct button_transmit_value * **button_from_int** (int i)
- signed char safe_int_char (int number)

  Takes and returns a int if it can fit into a PACKET_BYTE.
- int set_volume_left (struct control_packet *packet, int number)

  set the volume between 0-127. 0 is mute.
- int set_volume_right (struct control_packet *packet, int number)

  set the volume between 0-127. 0 is mute.
- int set_volume (struct control_packet *packet, int left, int right)

  Set the left and right volume. between 0-127. 0 is mute.
- int set_squelch_left (struct control_packet *packet, int number)
- int set_squelch_right (struct control_packet *packet, int number)
- int set_squelch (struct control_packet *packet, int left, int right)
- void ptt (struct control_packet *base_packet, int ptt)

  toggle transmission 2 to start 1 to stop
- void * send_control_packets (void *c)
- void **packet_debug** (const struct control_packet *packet, CONTROL_PACKET_INDEXED *input_packet_arr)

Variables

- const struct button_transmit_value **KEYPAD_BUTTON_NONE**
- const struct button_transmit_value **KEYPAD_NUMBER_BUTTONS** [10]
- const struct button_transmit_value **KEYPAD_BUTTON_A**
- const struct button_transmit_value **KEYPAD_BUTTON_B**
- const struct button_transmit_value **KEYPAD_BUTTON_C**
- const struct button_transmit_value **KEYPAD_BUTTON_D**
- const struct button_transmit_value **KEYPAD_BUTTON_HASH**
- const struct button_transmit_value **KEYPAD_BUTTON_X**
- const struct button_transmit_value **KEYPAD_BUTTON_P1**
- const struct button_transmit_value **KEYPAD_BUTTON_P2**
- const struct button_transmit_value **KEYPAD_BUTTON_P3**
- const struct button_transmit_value **KEYPAD_BUTTON_P4**
- const struct control_packet control_packet_defaults

  The recommended defaults for the control packet.

5.1.1 Detailed Description

Functions to manipulate control_packet structs

5.1.2 Function Documentation

5.1.2.1 signed char safe_int_char (int *number*)

Takes and returns a int if it can fit into a PACKET_BYTE.

Returns

NULL if the number will not fit into the packet (7 bits) else will return the provided int

5.1.2.2 void* send_control_packets (void * *c*)

Starts sending control packets as defined by SERIAL_CFG

This function is designed to be started as a thread

5.1.2.3 int set_squelch (struct **control_packet** * *packet*, int *left*, int *right*)

Set the left and right squelch. between 0-127. 127 filters no noise.

5.1.2.4 int set_squelch_left (struct **control_packet** * *packet*, int *number*)

Set the left squelch between 0-127. 127 filters no noise.e.

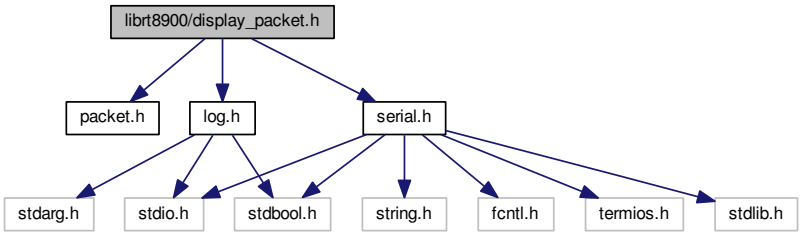
5.1.2.5 int set_squelch_right (struct control_packet * *packet,* int *number*)

Set the right squelch between 0-127. 127 filters no noise.

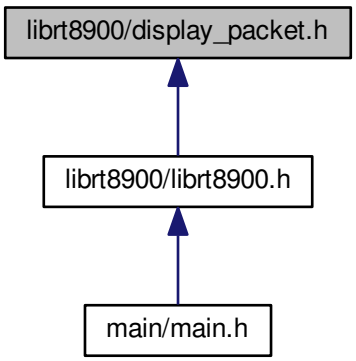
5.2 librt8900/display_packet.h File Reference

```
#include "packet.h"
#include "serial.h"
#include "log.h"
```

Include dependency graph for display_packet.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct radio_side
- struct radio_state

Macros

- #define **DISPLAY_PACKET_SIZE** 42
- #define **MS_PACKET_WAIT_TIME** 25
- #define **BIT_LOCATED_AT**(byte, bit) (((byte) ∗ 8) + (bit))

Typedefs

- typedef PACKET_BYTE **DISPLAY_PACKET**[DISPLAY_PACKET_SIZE]

Enumerations

- enum **display_packet_bitmasks** {
  AUTO_POWER_OFF = BIT_LOCATED_AT(0, 1), **KEYPAD_LOCK** = BIT_LOCATED_AT(30, 4), **SET** = B↩
  IT_LOCATED_AT(4, 0), **BD8600** = BIT_LOCATED_AT(4, 0),
  LEFT_BUSY = BIT_LOCATED_AT(12,2), **LEFT_MAIN** = BIT_LOCATED_AT(36, 2), **LEFT_TX** = BIT_LO↩
  CATED_AT(36, 3), **LEFT_MINUS** = BIT_LOCATED_AT(36, 5),
  LEFT_PLUS = BIT_LOCATED_AT(36, 4), **LEFT_DCS** = BIT_LOCATED_AT(9, 3), **LEFT_DEC** = BIT_LO↩
  CATED_AT(37, 0), **LEFT_ENC** = BIT_LOCATED_AT(36, 6),
  LEFT_SKIP = BIT_LOCATED_AT(37, 2), **LEFT_PMS** = BIT_LOCATED_AT(37, 1), **LEFT_POWER_LOW** =
  BIT_LOCATED_AT(1, 5), **LEFT_POWER_MEDIUM** = BIT_LOCATED_AT(0, 2),
  LEFT_FREQ_1_A = BIT_LOCATED_AT(13, 0), **LEFT_FREQ_1_B** = BIT_LOCATED_AT(13, 1), **LEFT_F**↩
  REQ_1_C = BIT_LOCATED_AT(13, 2), **LEFT_FREQ_1_D** = BIT_LOCATED_AT(13, 3),
  LEFT_FREQ_1_E = BIT_LOCATED_AT(13, 4), **LEFT_FREQ_1_F** = BIT_LOCATED_AT(13, 5), **LEFT_FR**↩
  EQ_1_G = BIT_LOCATED_AT(13, 6), **LEFT_FREQ_1_H** = BIT_LOCATED_AT(14, 1),
  LEFT_FREQ_1_I = BIT_LOCATED_AT(14, 2), **LEFT_FREQ_1_J** = BIT_LOCATED_AT(14, 3), **LEFT_FR**↩
  EQ_1_K = BIT_LOCATED_AT(14, 5), **LEFT_FREQ_1_L** = BIT_LOCATED_AT(14, 6),
  LEFT_FREQ_1_M = BIT_LOCATED_AT(15, 0), **LEFT_FREQ_2_A** = BIT_LOCATED_AT(10, 5), **LEFT_F**↩
  REQ_2_B = BIT_LOCATED_AT(10, 6), **LEFT_FREQ_2_C** = BIT_LOCATED_AT(11, 0),
  LEFT_FREQ_2_D = BIT_LOCATED_AT(11, 1), **LEFT_FREQ_2_E** = BIT_LOCATED_AT(11, 2), **LEFT_F**↩
  REQ_2_F = BIT_LOCATED_AT(11, 3), **LEFT_FREQ_2_G** = BIT_LOCATED_AT(11, 4),
  LEFT_FREQ_2_H = BIT_LOCATED_AT(11, 6), **LEFT_FREQ_2_I** = BIT_LOCATED_AT(12, 0), **LEFT_FR**↩
  EQ_2_J = BIT_LOCATED_AT(12, 1), **LEFT_FREQ_2_K** = BIT_LOCATED_AT(12, 3),
  LEFT_FREQ_2_L = BIT_LOCATED_AT(12, 4), **LEFT_FREQ_2_M** = BIT_LOCATED_AT(12, 5), **LEFT_F**↩
  REQ_3_A = BIT_LOCATED_AT(8, 3), **LEFT_FREQ_3_B** = BIT_LOCATED_AT(8, 4),
  LEFT_FREQ_3_C = BIT_LOCATED_AT(8, 5), **LEFT_FREQ_3_D** = BIT_LOCATED_AT(8, 6), **LEFT_FRE**↩
  Q_3_E = BIT_LOCATED_AT(9, 0), **LEFT_FREQ_3_F** = BIT_LOCATED_AT(9, 1),
  LEFT_FREQ_3_G = BIT_LOCATED_AT(9, 2), **LEFT_FREQ_3_H** = BIT_LOCATED_AT(9, 4), **LEFT_FRE**↩
  Q_3_I = BIT_LOCATED_AT(9, 5), **LEFT_FREQ_3_J** = BIT_LOCATED_AT(9, 6),
  LEFT_FREQ_3_K = BIT_LOCATED_AT(10, 1), **LEFT_FREQ_3_L** = BIT_LOCATED_AT(10, 2), **LEFT_F**↩
  REQ_3_M = BIT_LOCATED_AT(10, 3), **LEFT_FREQ_4_A** = BIT_LOCATED_AT(5, 2),
  LEFT_FREQ_4_B = BIT_LOCATED_AT(5, 3), **LEFT_FREQ_4_C** = BIT_LOCATED_AT(5, 4), **LEFT_FRE**↩
  Q_4_D = BIT_LOCATED_AT(5, 5), **LEFT_FREQ_4_E** = BIT_LOCATED_AT(5, 6),
  LEFT_FREQ_4_F = BIT_LOCATED_AT(6, 0), **LEFT_FREQ_4_G** = BIT_LOCATED_AT(6, 1), **LEFT_FRE**↩
  Q_4_H = BIT_LOCATED_AT(6, 3), **LEFT_FREQ_4_I** = BIT_LOCATED_AT(6, 4),
  LEFT_FREQ_4_J = BIT_LOCATED_AT(6, 5), **LEFT_FREQ_4_K** = BIT_LOCATED_AT(7, 0), **LEFT_FRE**↩
  Q_4_L = BIT_LOCATED_AT(7, 1), **LEFT_FREQ_4_M** = BIT_LOCATED_AT(7, 2),
  LEFT_FREQ_5_A = BIT_LOCATED_AT(3, 0), **LEFT_FREQ_5_B** = BIT_LOCATED_AT(3, 1), **LEFT_FRE**↩
  Q_5_C = BIT_LOCATED_AT(3, 2), **LEFT_FREQ_5_D** = BIT_LOCATED_AT(3, 3),
  LEFT_FREQ_5_E = BIT_LOCATED_AT(3, 4), **LEFT_FREQ_5_F** = BIT_LOCATED_AT(3, 5), **LEFT_FRE**↩
  Q_5_G = BIT_LOCATED_AT(3, 6), **LEFT_FREQ_5_H** = BIT_LOCATED_AT(4, 1),
  LEFT_FREQ_5_I = BIT_LOCATED_AT(4, 2), **LEFT_FREQ_5_J** = BIT_LOCATED_AT(4, 3), **LEFT_FREQ**↩
  _5_K = BIT_LOCATED_AT(4, 5), **LEFT_FREQ_5_L** = BIT_LOCATED_AT(4, 6),
  LEFT_FREQ_5_M = BIT_LOCATED_AT(5, 0), **LEFT_FREQ_6_A** = BIT_LOCATED_AT(0, 5), **LEFT_FRE**↩
  Q_6_B = BIT_LOCATED_AT(0, 6), **LEFT_FREQ_6_C** = BIT_LOCATED_AT(1, 0),
  LEFT_FREQ_6_D = BIT_LOCATED_AT(1, 1), **LEFT_FREQ_6_E** = BIT_LOCATED_AT(1, 2), **LEFT_FRE**↩

Q_6_F = BIT_LOCATED_AT(1, 3), **LEFT_FREQ_6_G** = BIT_LOCATED_AT(1, 4),
LEFT_FREQ_6_H = BIT_LOCATED_AT(1, 6), **LEFT_FREQ_6_I** = BIT_LOCATED_AT(2, 0), **LEFT_FREQ↩
_6_J** = BIT_LOCATED_AT(2, 1), **LEFT_FREQ_6_K** = BIT_LOCATED_AT(2, 3),
LEFT_FREQ_6_L = BIT_LOCATED_AT(2, 4), **LEFT_FREQ_6_M** = BIT_LOCATED_AT(2, 5), **LEFT_FRE↩
Q_7** = 0, **LEFT_FREQ_PERIOD** = BIT_LOCATED_AT(0, 3),
RIGHT_BUSY = BIT_LOCATED_AT(28,2), **RIGHT_MAIN** = BIT_LOCATED_AT(32, 0), **RIGHT_TX** = BIT_↩
LOCATED_AT(32, 1), **RIGHT_MINUS** = BIT_LOCATED_AT(32, 3),
RIGHT_PLUS = BIT_LOCATED_AT(32, 2), **RIGHT_DCS** = BIT_LOCATED_AT(22, 2), **RIGHT_DEC** = BIT↩
_LOCATED_AT(32, 5), **RIGHT_ENC** = BIT_LOCATED_AT(32, 4),
RIGHT_SKIP = BIT_LOCATED_AT(33, 0), **RIGHT_PMS** = BIT_LOCATED_AT(32, 6), **RIGHT_POWER_L↩
OW** = BIT_LOCATED_AT(17, 5), **RIGHT_POWER_MEDIUM** = BIT_LOCATED_AT(16, 2),
RIGHT_FREQ_1_A = BIT_LOCATED_AT(29, 0), **RIGHT_FREQ_1_B** = BIT_LOCATED_AT(29, 1), **RIGH↩
T_FREQ_1_C** = BIT_LOCATED_AT(29, 2), **RIGHT_FREQ_1_D** = BIT_LOCATED_AT(29, 3),
RIGHT_FREQ_1_E = BIT_LOCATED_AT(29, 4), **RIGHT_FREQ_1_F** = BIT_LOCATED_AT(29, 5), **RIGHT↩
_FREQ_1_G** = BIT_LOCATED_AT(29, 6), **RIGHT_FREQ_1_H** = BIT_LOCATED_AT(30, 1),
RIGHT_FREQ_1_I = BIT_LOCATED_AT(30, 1), **RIGHT_FREQ_1_J** = BIT_LOCATED_AT(30, 3), **RIGHT↩
_FREQ_1_K** = BIT_LOCATED_AT(30, 5), **RIGHT_FREQ_1_L** = BIT_LOCATED_AT(30, 6),
RIGHT_FREQ_1_M = BIT_LOCATED_AT(31, 0), **RIGHT_FREQ_2_A** = BIT_LOCATED_AT(26, 5), **RIGH↩
T_FREQ_2_B** = BIT_LOCATED_AT(26, 6), **RIGHT_FREQ_2_C** = BIT_LOCATED_AT(27, 0),
RIGHT_FREQ_2_D = BIT_LOCATED_AT(27, 1), **RIGHT_FREQ_2_E** = BIT_LOCATED_AT(27, 2), **RIGH↩
T_FREQ_2_F** = BIT_LOCATED_AT(27, 3), **RIGHT_FREQ_2_G** = BIT_LOCATED_AT(27, 4),
RIGHT_FREQ_2_H = BIT_LOCATED_AT(27, 6), **RIGHT_FREQ_2_I** = BIT_LOCATED_AT(28, 0), **RIGHT↩
_FREQ_2_J** = BIT_LOCATED_AT(28, 1), **RIGHT_FREQ_2_K** = BIT_LOCATED_AT(28, 3),
RIGHT_FREQ_2_L = BIT_LOCATED_AT(28, 4), **RIGHT_FREQ_2_M** = BIT_LOCATED_AT(28, 5), **RIGH↩
T_FREQ_3_A** = BIT_LOCATED_AT(24, 3), **RIGHT_FREQ_3_B** = BIT_LOCATED_AT(24, 4),
RIGHT_FREQ_3_C = BIT_LOCATED_AT(24, 5), **RIGHT_FREQ_3_D** = BIT_LOCATED_AT(24, 6), **RIGH↩
T_FREQ_3_E** = BIT_LOCATED_AT(25, 0), **RIGHT_FREQ_3_F** = BIT_LOCATED_AT(25, 1),
RIGHT_FREQ_3_G = BIT_LOCATED_AT(25, 2), **RIGHT_FREQ_3_H** = BIT_LOCATED_AT(25, 4), **RIGH↩
T_FREQ_3_I** = BIT_LOCATED_AT(25, 5), **RIGHT_FREQ_3_J** = BIT_LOCATED_AT(25, 6),
RIGHT_FREQ_3_K = BIT_LOCATED_AT(26, 1), **RIGHT_FREQ_3_L** = BIT_LOCATED_AT(26, 2), **RIGHT↩
_FREQ_3_M** = BIT_LOCATED_AT(26, 3), **RIGHT_FREQ_4_A** = BIT_LOCATED_AT(21, 2),
RIGHT_FREQ_4_B = BIT_LOCATED_AT(21, 3), **RIGHT_FREQ_4_C** = BIT_LOCATED_AT(21, 4), **RIGH↩
T_FREQ_4_D** = BIT_LOCATED_AT(21, 5), **RIGHT_FREQ_4_E** = BIT_LOCATED_AT(21, 6),
RIGHT_FREQ_4_F = BIT_LOCATED_AT(22, 0), **RIGHT_FREQ_4_G** = BIT_LOCATED_AT(22, 1), **RIGH↩
T_FREQ_4_H** = BIT_LOCATED_AT(22, 3), **RIGHT_FREQ_4_I** = BIT_LOCATED_AT(22, 4),
RIGHT_FREQ_4_J = BIT_LOCATED_AT(22, 5), **RIGHT_FREQ_4_K** = BIT_LOCATED_AT(23, 0), **RIGHT↩
_FREQ_4_L** = BIT_LOCATED_AT(23, 1), **RIGHT_FREQ_4_M** = BIT_LOCATED_AT(23, 2),
RIGHT_FREQ_5_A = BIT_LOCATED_AT(19, 0), **RIGHT_FREQ_5_B** = BIT_LOCATED_AT(19, 1), **RIGH↩
T_FREQ_5_C** = BIT_LOCATED_AT(19, 2), **RIGHT_FREQ_5_D** = BIT_LOCATED_AT(19, 3),
RIGHT_FREQ_5_E = BIT_LOCATED_AT(19, 4), **RIGHT_FREQ_5_F** = BIT_LOCATED_AT(19, 5), **RIGHT↩
_FREQ_5_G** = BIT_LOCATED_AT(19, 6), **RIGHT_FREQ_5_H** = BIT_LOCATED_AT(20, 1),
RIGHT_FREQ_5_I = BIT_LOCATED_AT(20, 2), **RIGHT_FREQ_5_J** = BIT_LOCATED_AT(20, 3), **RIGHT↩
_FREQ_5_K** = BIT_LOCATED_AT(20, 5), **RIGHT_FREQ_5_L** = BIT_LOCATED_AT(20, 6),
RIGHT_FREQ_5_M = BIT_LOCATED_AT(21, 0), **RIGHT_FREQ_6_A** = BIT_LOCATED_AT(16, 5), **RIGH↩
T_FREQ_6_B** = BIT_LOCATED_AT(16, 6), **RIGHT_FREQ_6_C** = BIT_LOCATED_AT(17, 0),
RIGHT_FREQ_6_D = BIT_LOCATED_AT(17, 1), **RIGHT_FREQ_6_E** = BIT_LOCATED_AT(17, 2), **RIGH↩
T_FREQ_6_F** = BIT_LOCATED_AT(17, 3), **RIGHT_FREQ_6_G** = BIT_LOCATED_AT(17, 4),
RIGHT_FREQ_6_H = BIT_LOCATED_AT(17, 6), **RIGHT_FREQ_6_I** = BIT_LOCATED_AT(18, 0), **RIGHT↩
_FREQ_6_J** = BIT_LOCATED_AT(18, 1), **RIGHT_FREQ_6_K** = BIT_LOCATED_AT(18, 3),
RIGHT_FREQ_6_L = BIT_LOCATED_AT(18, 4), **RIGHT_FREQ_6_M** = BIT_LOCATED_AT(18, 5), **RIGH↩
T_FREQ_7** = BIT_LOCATED_AT(16, 0), **RIGHT_FREQ_PERIOD** = BIT_LOCATED_AT(16, 3) }

- enum **rt8900_power_level** {
  POWER_UNKNOWEN = 0, **POWER_LOW**, **POWER_MEDIUM_FUZZY**, **POWER_MEDIUM_1**,
  POWER_MEDIUM_2, **POWER_HIGH** }

Functions

- int **display_packet_read** (DISPLAY_PACKET packet, const enum display_packet_bitmasks bit_number)

- void insert_shifted_packet (DISPLAY_PACKET packet, unsigned char buffer[], size_t buffer_length, int start_of_packet_index)
- int segment_to_int (int segment_bitmask)
- void read_busy (DISPLAY_PACKET packet, struct radio_state *state)

  Gets busy state from display_packet.
- void read_main (DISPLAY_PACKET packet, struct radio_state *state)
- void read_power_fuzzy (DISPLAY_PACKET packet, struct radio_state *state)

  Gets the power levels of the radios only using reads.
- int read_frequency (DISPLAY_PACKET packet, struct radio_state *state)

  Writes the frequency to the state packet.
- int is_main (struct radio_state *radio, struct radio_side *side)

  Check if the input radio is the main.
- void read_packet_state (DISPLAY_PACKET packet, struct radio_state *state)

  runs all packet read functions on the given packet

5.2.1 Detailed Description

Functions to get and sed data from the display_packet

5.2.2 Function Documentation

5.2.2.1 void insert_shifted_packet (DISPLAY_PACKET *packet*, unsigned char *buffer[]*, size_t *buffer_length*, int *start_of_packet_index*)

Write to the packet in the correct order.

Warning

This assumes buffer array length is DISPLAY_PACKET_SIZE (42)

5.2.2.2 int is_main (struct radio_state * *radio*, struct radio_side * *side*)

Check if the input radio is the main.

Returns

0 if radio is not currently main

5.2.2.3 int read_frequency (DISPLAY_PACKET *packet*, struct radio_state * *state*)

Writes the frequency to the state packet.

Returns

0 on success and 1 on error.

5.2.2.4 void read_main (DISPLAY_PACKET *packet*, struct radio_state * *state*)

Sets the main correct pointer to the correct radio,

Returns

NULL if nether selected

5.2.2.5 int segment_to_int (int *segment_bitfield*)

Takes a bitfield and matches to known numbers an char of bits (ordered as described above)

Returns

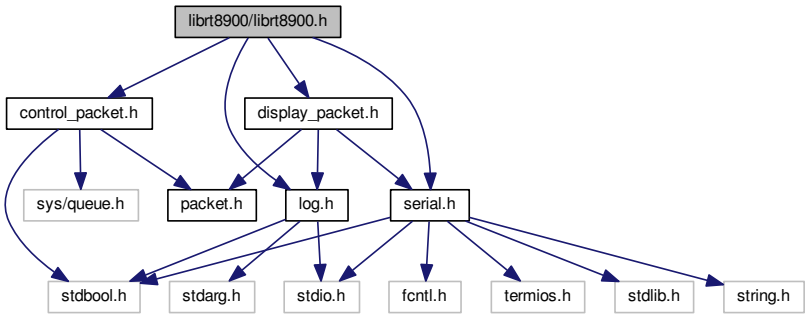
0-9 and -1 on error

5.3 librt8900/librt8900.h File Reference

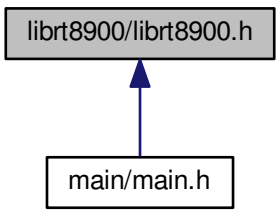
The main headder file for the librt8900 libary. Contains functions that use both the contol_packet and DISPLAY_↩ PACKET.

```
#include "log.h"
#include "serial.h"
#include "control_packet.h"
#include "display_packet.h"
```

Include dependency graph for librt8900.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct range_KHz

  Represents one of the capable ranges of the radio.
- struct control_packet_sender_config

  Configuration for sending packets.
- struct display_packet_receiving_config

  Configuration for receiving packets.
- struct SERIAL_CFG

Macros

- #define **VALID_POWER_LEVEL**(num) ((num) == POWER_LOW || (num) == POWER_MEDIUM_FUZZY || (num) == POWER_HIGH)

Enumerations

- enum frequency_permission { **INVALID_FREQUENCY** = 0, **VALID_FREQUENCY_RX_ONLY** = 1, **VALID_FREQUENCY** = 2 }

  Represents the abilities of the radio for a particular frequency.

Functions

- void * send_control_packets (void *c)
- void * receive_display_packets (void *c)
- const struct range_KHz * get_range (int frequency_khz)
- int in_freq_range (int frequency_khz)
- int get_frequency (struct radio_side *radio)

  Gets the current frequency of the radio.
- bool current_freq_valid (struct radio_side *radio)
- void send_new_packet (SERIAL_CFG *config, struct control_packet *new_packet, enum pop_queue_behaviour free_choice)
- bool check_radio_rx (SERIAL_CFG *config)

  Check that we are received from the radio at lest once.
- void wait_to_send (const SERIAL_CFG *cfg)

  Blocks until there are no new packets to send.
- void shutdown_threads (SERIAL_CFG *cfg)

  gracefully stops send_control_packets() and receive_display_packets()
- int set_frequency (SERIAL_CFG *cfg, struct control_packet *base_packet, int number)

  Adds the required packets to dial a number.
- int set_main_radio (SERIAL_CFG *cfg, struct control_packet *base_packet, enum radios side)

  Switches context to the desired radio.
- int set_left_power_level (SERIAL_CFG *cfg, struct control_packet *base_packet, enum rt8900_power_level power_level)

  sets left power level on radio
- int set_right_power_level (SERIAL_CFG *cfg, struct control_packet *base_packet, enum rt8900_power_level power_level)

  sets right power level on radio
- int set_power_button (SERIAL_CFG *cfg)

  Experimental! Sets the dtr pin low for one second to trigger radio on.
- int get_display_packet (SERIAL_CFG *config, DISPLAY_PACKET packet)

  writes the most recent packet to

5.3.1 Detailed Description

The main headder file for the librt8900 libary. Contains functions that use both the contol_packet and DISPLAY_↩ PACKET.

5.3.2 Function Documentation

5.3.2.1 bool check_radio_rx (SERIAL_CFG * *config*)

Check that we are received from the radio at lest once.

Warning

Will still return True if the radio is disconnected after some time.

Returns

true is the radio has been seen else false

5.3.2.2 bool current_freq_valid (struct radio_side * *radio*)

Returns

True if the radio is currently set to a frequency it can TX on

5.3.2.3 int get_display_packet (SERIAL_CFG * *config,* DISPLAY_PACKET *packet*)

writes the most recent packet to

Parameters

packet.	Gets the most recent packet. Will block if there is currently a half updated most recent packet

5.3.2.4 const struct range_KHz* get_range (int *frequency_khz*)

Returns (struct range_KHz) if input is within that range. Else returns NULL Looks for tx&rx ranges first

5.3.2.5 int in_freq_range (int *frequency_khz*)

returns 0 if invalid range, * or 1 if only rx allowed, * and 2 for all allowed

5.3.2.6 void* receive_display_packets (void * *c*)

Writes the latest packet to a segment of memory This function is designed to be started as a thread

5.3.2.7 void* send_control_packets (void * c)

Starts sending control packets as defined by SERIAL_CFG

This function is designed to be started as a thread

5.3.2.8 void send_new_packet (SERIAL_CFG * config, struct control_packet * new_packet, enum pop_queue_behaviour free_choice)

Adds a control_packet (pointer) to the send queue, should only be called once the queue has been initialized

5.3.2.9 int set_frequency (SERIAL_CFG * cfg, struct control_packet * base_packet, int number)

Adds the required packets to dial a number.

They are then be added to the queue

Returns

0 on success

5.3.2.10 int set_left_power_level (SERIAL_CFG * cfg, struct control_packet * base_packet, enum rt8900_power_level power_level)

sets left power level on radio

Presses the Low button on the radio until the selected power is set

5.3.2.11 int set_main_radio (SERIAL_CFG * cfg, struct control_packet * base_packet, enum radios side)

Switches context to the desired radio.

First check you are not already on this mode using is_main() else you will enter frequency edit mode!

5.3.2.12 int set_right_power_level (SERIAL_CFG * cfg, struct control_packet * base_packet, enum rt8900_power_level power_level)

sets right power level on radio

Presses the Low button on the radio until the selected power is set

Returns

0 on sucess, 1 on internal error and 2 on user error

5.3.2.13 void shutdown_threads (SERIAL_CFG * *cfg*)

gracefully stops send_control_packets() and receive_display_packets()

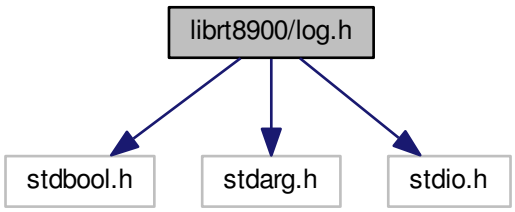
This can be used anytime to gracefully stop sending and receiving on serial Threads will be able to join after running ths function

5.4 librt8900/log.h File Reference

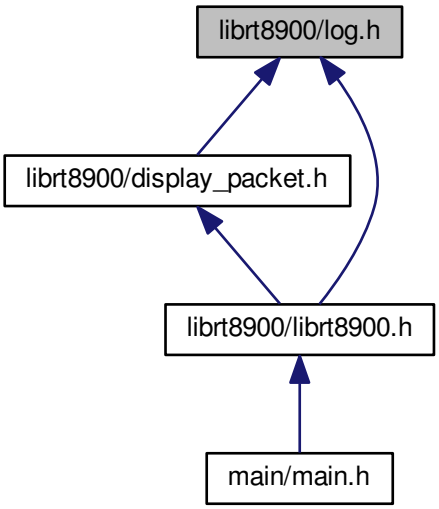
logging wrapper with levels for the librt8900 library.

```
#include <stdbool.h>
#include <stdarg.h>
#include <stdio.h>
```

Include dependency graph for log.h:



This graph shows which files directly or indirectly include this file:



Enumerations

- enum **rt8900_logging_level** {
  RT8900_NOLOG = 0, **RT8900_FATAL**, **RT8900_ERROR**, **RT8900_WARNING**,
  RT8900_INFO, **RT8900_DEBUG**, **RT8900_TRACE** }

Functions

- void set_log_level (enum rt8900_logging_level i)

  sets the global logging level of lib rt8900
- void log_msg (enum rt8900_logging_level level, char const ∗fmt,...)

  Used in place of println for logging.

5.4.1 Detailed Description

logging wrapper with levels for the librt8900 library.

Contains functions that use both the control_packet and DISPLAY_PACKET.

5.4.2 Function Documentation

5.4.2.1 void log_msg (enum rt8900_logging_level *level,* char const ∗ *fmt,* ...)

Used in place of println for logging.

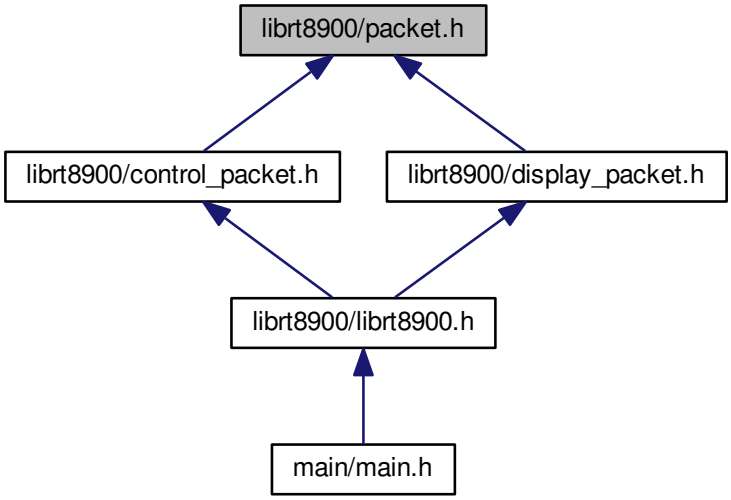
Parameters

level	is the log level this message should appear at

5.5 librt8900/packet.h File Reference

Data representation of a single byte in a packet.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct FT8900BYTE
- union PACKET_BYTE

  Used to store 1 byte of data from the packet.

Macros

- #define **NUM_DATA_BITS** 8
- #define **NUM_STOP_BITS** 1
- #define **NUM_PARITY_BITS** 0

Enumerations

- enum check_num_values { **SBZ** = 0X00, **SBO** = 0X01 }
- enum common_7bit_data_values { **DATA_MAX_NUM** = 0x7F, **DATA_MIN_NUM** = 0X00 }

  The highest and lowest possible values that can be held in the data section of the packet.

Functions

- void **print_char** (char byte)
- int find_packet_start (unsigned char buffer[ ], size_t length)

5.5.1 Detailed Description

Data representation of a single byte in a packet.

5.5.2 Enumeration Type Documentation

5.5.2.1 enum check_num_values

The last bit in a packet byte. 1 if it is the first in the packet.

5.5.3 Function Documentation

5.5.3.1 int find_packet_start (unsigned char *buffer[ ],* size_t *length*)

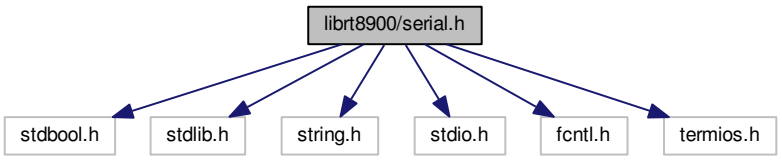
the start of the known packet could be anywhere in the buffer this function finds the starting index based of it's bit marker

5.6 librt8900/serial.h File Reference

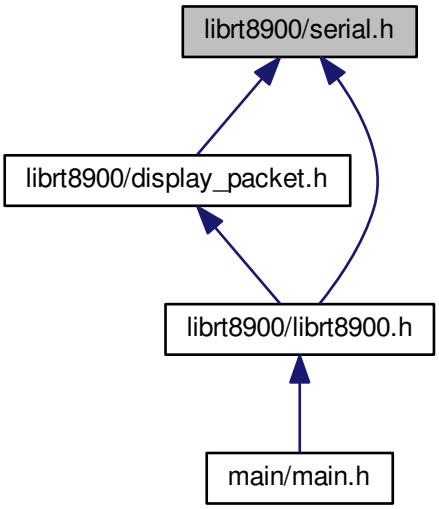
Serial handling.

```
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>
```

Include dependency graph for serial.h:



This graph shows which files directly or indirectly include this file:



Functions

- void open_serial (int *fd, char **serial_path, bool *dtr_pin_for_on)

  Open and configure a serial port for sending and receiving from radio.

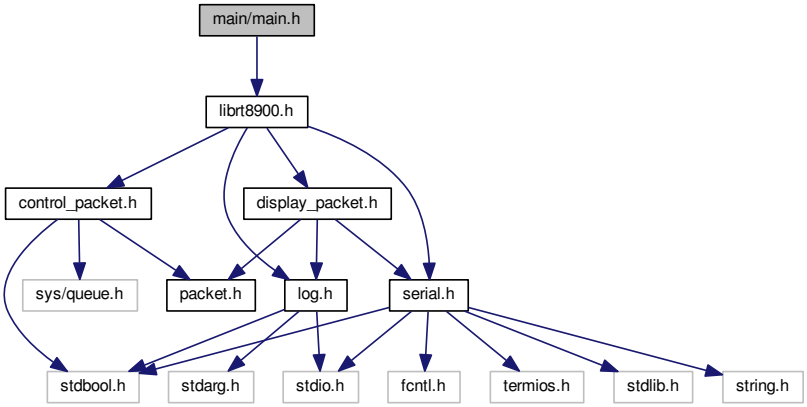
5.6.1 Detailed Description

Serial handling.

5.7 main/main.h File Reference

```
#include "librt8900.h"
```

Include dependency graph for main.h:



Macros

- #define **PROMPT_BUFFER_SIZE** 32
- #define **TURN_ON_RADIO_TRYS** 3
- #define **ANSI_COLOR_YELLOW** "\x1b[33m"
- #define **ANSI_COLOR_GREEN** "\x1b[32m"
- #define **ANSI_COLOR_RESET** "\x1b[0m"

Functions

- int **cmd_help** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_exit** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_get_frequency** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_set_frequency** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_get_busy** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_get_main** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_set_main** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_get_power** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_set_power** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_get_ptt** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_set_ptt** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_set_volume** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)
- int **cmd_set_squelch** (char **args, SERIAL_CFG *config, struct control_packet *base_packet)

5.7.1 Detailed Description

A command line shell for controlling the YAESU FT-8900R Transceiver.

Index

button_transmit_value, 7

CONTROL_PACKET_INDEXED, 9
check_num_values
    packet.h, 30
check_radio_rx
    librt8900.h, 25
cmd, 7
control_packet, 8
control_packet.h
    safe_int_char, 18
    send_control_packets, 18
    set_squelch, 18
    set_squelch_left, 18
    set_squelch_right, 18
control_packet_q_node, 10
control_packet_sender_config, 11
current_freq_valid
    librt8900.h, 25

display_packet.h
    insert_shifted_packet, 22
    is_main, 22
    read_frequency, 22
    read_main, 22
    segment_to_int, 23
display_packet_receiving_config, 11

FT8900BYTE, 12
find_packet_start
    packet.h, 30

get_display_packet
    librt8900.h, 25
get_range
    librt8900.h, 25

in_freq_range
    librt8900.h, 25
insert_shifted_packet
    display_packet.h, 22
is_main
    display_packet.h, 22

librt8900.h
    check_radio_rx, 25
    current_freq_valid, 25
    get_display_packet, 25
    get_range, 25
    in_freq_range, 25
    receive_display_packets, 25
    send_control_packets, 25
    send_new_packet, 26
    set_frequency, 26
    set_left_power_level, 26
    set_main_radio, 26
    set_right_power_level, 26
    shutdown_threads, 26
librt8900/control_packet.h, 15
librt8900/display_packet.h, 19
librt8900/librt8900.h, 23
librt8900/log.h, 27
librt8900/packet.h, 28
librt8900/serial.h, 30
log.h
    log_msg, 28
log_msg
    log.h, 28

main/main.h, 32

PACKET_BYTE, 12
packet.h
    check_num_values, 30
    find_packet_start, 30

radio_side, 13
radio_state, 13
range_KHz, 14
read_frequency
    display_packet.h, 22
read_main
    display_packet.h, 22
receive_display_packets
    librt8900.h, 25

SERIAL_CFG, 14
safe_int_char
    control_packet.h, 18
segment_to_int
    display_packet.h, 23
send_control_packets
    control_packet.h, 18
    librt8900.h, 25
send_new_packet
    librt8900.h, 26
set_frequency
    librt8900.h, 26
set_left_power_level
    librt8900.h, 26
set_main_radio

librt8900.h, 26
set_right_power_level
librt8900.h, 26
set_squelch
control_packet.h, 18
set_squelch_left
control_packet.h, 18
set_squelch_right
control_packet.h, 18
shutdown_threads
librt8900.h, 26

Annotated Bibliography

[1] S. Duckworth, “What is amateur radio?” rsgb.org/main/get-started-in-amateur-radio/what-is-amateur-radio/, 2017, accessed 21 Apr 2017.

A description of what is amateur radio.

[2] Microbit, “RRC-1258MkIIs Yaesu Twin,” http://shop.microbit.se/webshop/catalog/product_info.php?cPath=21&products_id=48, 2015, accessed 21 Apr 2017.

The only competing remote radio control solution that I found. Priced at $708 for the 2 required boxes

[3] James "Scott" Duckworth (NA4IT), “Digital Mode Operation vs CAT Control,” http://www.eham.net/articles/23794, June 2010, accessed February 2015.

A good explanation of the terminology around controlling radios.

[4] *Computer Control of a Yaesu FT-8900 Amateur Radio*. Available upon request from the computer science department Aberystwyth, 2016, bgc@aber.ac.uk.

A report from a past undergraduate on the same topic. This paper provides excellent starting reference identifying approximately 60-80% of the head-body protocol for the FT8900 radio.

[5] Interlock Rochester, “Ft-8800r tech documentation,” http://www.interlockroc.org/wiki/index.php/F8800R_tech_documentation, accessed on 13/02/2016.

This is some reverse engineering of the 8800r a similar model of radio. I found that it’s packet structure is the same as the 8900r, after reading a cashed copy of this (the original is no longer available.) Its information is contained within Ben Coopers report [4].

[6] Nate Bargmann (Maintainer) and contributors, “Ham Radio Control Libraries,” Hamlib.org, 2017, accessed 1 May 2017.

An open source application that provides a generic API for a number of radios. AKA an abstraction layer for application programmers to work on.

[7] D. Wells, “Extreme programming: A gentle introduction,” http://www.extremeprogramming.org/, 2013, accessed 6 May 2017.

The somewhat official XP page. Explains the core concept of XP and agile in general, then goes into further detail. Uses hidden links in the text so that you can quiz people if they actual read the document.

[8] J. Rasmusson, "Extreme Programming," http://www.agilenutshell.com/xp, 2015, accessed 26 Apr 2017.

This is a good bluet summary of the basics of XP. This was useful memory prompt when writing about XP.

[9] K. Beck, D. Elza, J. Brewer, Y. Sharon, E. Smith, R. Jeffries, D. M. Nicol, P. Merel, *et al.*, "Extreme Programming For One," http://wiki.c2.com/?ExtremeProgrammingForOne, 2014, accessed 26 Apr 2017.

A discussion of using XP as a single developer from some the creators of the XP methodology such as Kent Beck and Eric Smith. This document was pivotal in creating my own methodology for the project

[10] James "Scott" Duckworth (NA4IT), "Lone developer project management," http://wiki.c2.com/?LoneDeveloperProjectManagement, June 2010, accessed February 2015.

Shows what changes can be made to allow for Extreme programming with one developer. Also points outs relevant anti-patterns to avoid and how to avoid them.

[11] B. Donahue, G. Miller, S. Roth, A. Klautsan, M. Mentovai, J. Ma, D. Greiman, M. Molteni, *et al.*, "Google Test," https://github.com/google/googletest/, accessed 1 May 2017.

The library I used for unit testing. Very fast and well made with a number of useful assertion methods, setup functions etc.

[12] "Jenkins," https://jenkins.io/, accessed on 6/04/2017.

Jenkins is one of the top open source automation tools. Jenkins can be used to automate more than just VCS tests but this is the intended focus for this project.

[13] L. torvolds, "Linux kernel coding style," https://github.com/torvalds/linux/blob/master/Documentation/process/coding-style.rst, Nov 2016, accessed April 2017.

Reference to the coding style guide used. The guide was written by Linus Torvalds and contains much of his infamous humour making the read a lot less dry than most style guides. It also contains a good deal of rational. This code style is no doubt controversial, but seems to work for the C language.

[14] T. Preston-Werner, "Semantic Versioning 2.0.0," http://semver.org/spec/v2.0.0.html, accessed 28 April 2017.

The semantic versioning scheme. I used this to decriobe vershons of the my program I used the V2.0.0 definition of this scheme for my project.

[15] A. Csete *et al.*, "Free, Ham Radio Control Application," http://groundstation.sourceforge.net/grig/index.php, July 2015, accessed May 1 2017.

Gnome RIG is simple a GUI application written using the Gtk+ and Gnome widgets.

[16] C. Liechti, "Pyserial," https://pypi.python.org/pypi/pyserial, accessed 29 April 2017.

The library that is tested for use if the application was to be made in python

[17] http://www.yaesu.co.uk/files/FT-8900R_Opr%20Manual.pdf, 2003, accessed 2nd April, 2017.

The user manual for the FT-8900R radio. This was a somewhat use-full reference but was not enough to implement any features from. There are many behaviours that are not documented or mentioned. For example how the power can not be changed if the radio is set to a channel that it is not permitted to transmit on.

[18] "cmake," https://cmake.org/, accessed on 6/04/2017.

Cmake was used to buid and test the project application. It uses its own scripting language files to generate make files will all the correct targets and options that are need for specific platforms

[19] D. Stenberg, "Fewer mallocs in curl," https://daniel.haxx.se/blog/2017/04/22/fewer-mallocs-in-curl/, accessed 3 May 2017.

This is a blog post from the current maintainer of the open souce program "Curl" on how grouping struts in the codebase so that there are fewer mallocs resulted in a %30 improvement in performance. The author uses performance benchmarks to provide evidence to this point. It should be noted that this application made hundreds and sometimes thousands of malloc calls in its regular operation, meaning that this improvement was made at a large scale. This was the counter argument in the decision making process of choosing a non-intrusive data structure.

[20] "Valgrind," http://valgrind.org/, accessed 5 May 2017.

Valgrind is a set of analysis tools to find and improve memory management. It can work almost any binary and is actually a virtual machine.

[21] "Mumble," http://cppcheck.sourceforge.net/, accessed 5 May 2017.

This is a tool for C and C++ code analysis. I used this tool in my project to good effect. However it required a lot of setup of flags to get the desired output that could be interpreted by Jenkins.

[22] "Mumble," http://wiki.mumble.info/wiki/Main_Page, accessed 5 May 2017.

This is a low latency, low bandwidth, low resource using voice over IP application. While its primary usage is for gaming it is often used in places where a high quality version of more popular applications such as Skype is used. Mumble is capable of authentication and encryption and is often trusted to run exposed to the Internet. This application can be used in conjunction with the project to achieve a fully working remote shack. It is envisioned that the mumble server would be run on the remote side and users would connect there clients to it.

[23] Icom Inc, "HAM RADIO TERMS," http://www.icomcanada.com/techbulletin/tb1/ham_radio_terms.pdf, 2002, accessed 23 April 2017.

Explanation of many radio terms that I (and the reader may) find a use-full reference guide.