The University of Oxford

Engineering Science

Fourth Year Project

PiCom: A Digital Communication Test Bed Based on Raspberry Pi

Candidate

Cameron Eadie

Exeter College

Supervisor

Justin Coon

Trinity Term, 2018

FINAL HONOUR SCHOOL OF ENG

UNIVERSITY OF OXFORD

DECLARATION OF AUTHORSHIP

You should complete this certificate. It should be bound into your fourth year project report, immediately after your title page. Three copies of the report should be submitted to the Chairman of examiners for your Honour School, c/o Clerk of the Schools, examination Schools, High Street, Oxford.

Name (in capitals): ……………………………………………………………………………………………

College (in capitals): …………………………………… **Supervisor:** ………………………………………………

Title of project (in capitals): ………………………………..…………………………………………………

Page count (excluding risk and COSHH assessments): …………………………………………………

Please tick to confirm the following:

I have read and understood the University's disciplinary regulations concerning conduct in examinations and, in particular, the regulations on plagiarism (*The University Student Handbook. The Proctors' and Assessors' Memorandum, Section 8.8*; available at https://www.ox.ac.uk/students/academic/student-handbook)	☐
I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at https://www.ox.ac.uk/students/academic/guidance/skills.	☐
The project report I am submitting is entirely my own work except where otherwise indicated.	☐
It has not been submitted, either partially or in full, for another Honour School or qualification of this University (except where the Special Regulations for the subject permit this), or for a qualification at any other institution.	☐
I have clearly indicated the presence of all material I have quoted from other sources, including any diagrams, charts, tables or graphs.	☐
I have clearly indicated the presence of all paraphrased material with appropriate references.	☐
I have acknowledged appropriately any assistance I have received in addition to that provided by my supervisor.	☐
I have not copied from the work of any other candidate.	☐
I have not used the services of any agency providing specimen, model or ghostwritten work in the preparation of this project report. (See also section 2.4 of Statute XI on University Discipline under which members of the University are prohibited from providing material of this nature for candidates in examinations at this University or elsewhere: http://www.admin.ox.ac.uk/statutes/352-051a.shtml.)	☐
The project report does not exceed 50 pages (including all diagrams, photographs, references and appendices).	☐
I agree to retain an electronic copy of this work until the publication of my final examination result, except where submission in hand-written format is permitted.	☐
I agree to make any such electronic copy available to the examiners should it be necessary to confirm my word count or to check for plagiarism.	☐

Candidate's signature: …………………………………………….. **Date:** ………………………..

Abstract

RED - Important information to check/change

BLUE - Sections and parts that still need writing/editing

GREEN - Formatting of sections and layout of images/figures

Here we shall have the abstract.

Write Abstract - Write this last.

USE "CLEARPAGE" (INCLUDE FIGURES) AND "NEWPAGE" TO MAKE SURE SECTIONS ARE AS THEY SHOULD BE

Read back through the documentation, help pages, my own notes on how 4YP should be written and presented

Contents

1 Introduction **5**

1.1 Motivation 6

1.2 Background - Literature Review 6

1.3 Contributions 8

2 The Raspberry Pi and the Test Bed **9**

2.1 Raspberry Pi Fundamentals 9

2.1.1 Setting up the Raspberry Pi 10

2.2 Test Bed Architecture 14

2.2.1 Digital Analogue Converter 14

2.2.2 Analogue Digital Converter 15

2.2.3 Multiplier 17

2.2.4 Quadrature Pulse Generator 18

2.2.5 Parts Used 18

2.3 Programming 21

2.3.1 On-Off Keying 23

2.3.2 Advanced Modulation Schemes 25

3 Electronic Testing **38**

3.1 Electrical Characteristics of the Raspberry Pi 38

3.2 Computational Characteristics of the Raspberry Pi 40

3.2.1 Comparing Python and C 41

3.2.2 Maximum Frequency 42

3.3 Characterising Components of the Test Bed 43

3.3.1 Overclocking Components 44

4 Communications Testing 45

4.1 SNR for Different Modulation Schemes . . . 46

4.2 Error Rate . . . 46

4.3 Channel Coding . . . 46

5 Conclusion 47

Bibliography 48

Appendices 50

A Risk Assessments 51

A.1 General Risk Assessment . . . 51

A.2 Computer Risk Assessment . . . 53

Todo list

RED - Important information to check/change . . . 1

BLUE - Sections and parts that still need writing/editing . . . 1

GREEN - Formatting of sections and layout of images/figures . . . 1

Write Abstract - Write this last. . . . 1

USE "CLEARPAGE" (INCLUDE FIGURES) AND "NEWPAGE" TO MAKE SURE SECTIONS ARE AS THEY SHOULD BE . . . 1

Read back through the documentation, help pages, my own notes on how 4YP should be written and presented . . . 1

Still need to write the Contributions Chapter (last) . . . 8

Include references to the data sheets of all of the components . . . 14

Write up Multiplier and sine generator . . . 17

Depending whether it's doable, include this section without multipliers on transmitter, pseudo ground on output . . . 17

Replace this with picture/diagram compound . . . 18

Add references to above suppliers . . . 19

Continue to add to this list as you write the Architecture section - remove redundant information already discussed in the Section itself for each part . . . 20

Make sure it's consistent with use of Quadrature Sinusoid Generator vs Oscillator in report 20

Rephrase this and make sure the section referenced is as consistent with this comment as possible - reference that paper or lit review . . . 21

Make sure the corresponding section (USE THE RIGHT SUB-SECTION) does reference the fact that the DAC doesn't work exactly as it's supposed to and how I dealt with it . . . 27

Write the C Sections (and if time, the test masks code) . . . 33

Write Computing and Components Sections - I have all the data, screen shots and component characterising values . . . 40

Comparison of transmitter bank vs individual write and receiver read (callback) and read vars on clock pulse vs bank read on clock pulse – Also mention 42

Neet to write up the Comms Testing – Mention the fact that we had to deal with deletion channel vs error channel, clocked vs unclocked, Justin's project - http://diwine-project.eu/public/publications.html

I'm not entirely clear on how you would calculate the SNR for these signals, considering we end up with a bit stream on either end and so bit error rate is easy to understand. Some insight as I'm writing up the test results would be handy – An SNR measurement assumes there is noise in the system. If you measure noise naturally, or otherwise artificially generate it, then the SNR would be E[wanted signal power]/E[noise signal power]. 46

Write Conclusion 47

Make sure each figure is referenced explicitly in text and has a title, maks sure section references have the title where ambiguous otherwise, check for inline code consistency . 47

Format for web pages should be: 50

Chapter 1

Introduction

The idea of a software radio is attributed to Dr. Joseph Mitola III in the early 1990's, referring to radios which could be reconfigured by changing the software, allowing for changes to communications protocols without changing the underlying hardware [1]. The ideal software radio would consist of a computer connected to a Digital-Analogue Converter (DAC) to an antenna as a transmitter, and an antenna connected to an Analogue-Digital Converter (ADC) to a computer as a receiver. An ideal software radio is not realisable with the current converter technology, but with a few dedicated hardware components implementing the radio frequency (RF) front end, it is possible to develop a radio communication system which can implement a variety of alternative modulation schemes with little to no modification to the hardware simply by changing how the software operates. This is the idea of a 'software-defined radio'.

'Software-defined radio' (SDR) test beds have become more popular as the technology becomes more accessible and cost-efficient. These are test beds used to test the usefulness and effectiveness of various coding and modulation schemes for wireless communication. Digital communication systems in the modern era are constantly evolving as researchers formulate new more efficient methods of transmitting data, approaching channel capacity. These schemes are developed theoretically, but they eventually need to be tested in order to prove their utility, being able to physically test this with software instead of custom hardware is a significant advantage. There are a number of available 'software-defined radio' test beds, however most of these can be very expensive. Although these test beds supply advanced tool kits, the base functionality is really all that is required to successfully test new communication schemes.

1.1 Motivation

The Raspberry Pi is a low-cost computer based on an ARM processor running a Linux distribution. It has a number of software-programmable General Purpose Input/Output (GPIO) pins which can be used to interface with external devices. This combination of computational power and versatility has caught the attention of a wide range of engineers and hobbyists [2]. The Raspberry Pi provides an easily accessible way of developing an alternative SDR test bed which doesn't require thousands of pounds to purchase (see Section 1.2). This would make such test beds significantly more available, such that one might be built by anyone interested in investigating the physical realisation of their ideas.

Modern test beds often consist of radio transmitters and receivers wired together with adjustable attenuation between them. This allows for the testing of these radio systems in close proximity, while simulating different distances between them [3]. The aim of this project is to develop a basic wired digital communications test bed between two Raspberry Pis, with one as the transmitter and the other as the receiver. This will serve as a proof of concept for the development of low-budget software-defined radio test beds, and it is developed with costing in mind. The next stage is then to attempt to characterise the performance of the test bed. This consists of characterising the Raspberry Pi and other components used, as well as the test bed as pertains to its ability to communicate. This includes testing its reliability for different modulation and coding schemes, and identifying key characteristics such as noise and bandwidth limitations.

1.2 Background - Literature Review

There are a number of SDR test beds which have been developed in recent years. A large number of them make use of one or more Ettus Universal Software Radio Peripherals (USRP), including the National Instruments Communications Kit which contains two such devices and is often used as an educational tool. These USRPs are powerful tools, however the software defined radio peripherals range in price from *$880* to *$3,510*. The reconfigurable SDR peripherals for rapid prototyping are even more expensive with prices starting around *$5,000* **??**. An available free open source software used by a number of projects is GNU Radio. [**web˙GNURadio:http://garethhayes.net/gnu-radio-rtl˙sdr-raspberry-pi/**]. While this is a great option for a number of test beds, it does require a lot of processing power and space, which would impact the performance of a small processor. There have also been issues compiling GNU Radio on Raspberry Pis but this can be figured out [**web˙GNUonRPi:http://garethhayes.net/gnu-rad**

However, because of these issues and the significant code base which would take time to master, the decision to develop independent software was made. It could be included in an alternative iteration of this project's test bed at a later point if desired.

In 2008 a group at the University of Notre Dame developed a portable software radio using only commercial off-the-shelf components, an Ettus USRP and GNU Radio [**web˙PortableSR**]. It weighed 7 pounds and was the first portable software radio of its kind to their knowledge. A single device made up of these components cost *$3,700* (about *$1,900* at the time) and constituted a large development for the flexibility of radio systems. Another group in 2014 using a USRP and GNU Radio developed the first SDR test bed for the testing of a RF subsampling receiver [**web˙SDRTB˙SubSamplingReceiver**]. This receiver sampled a 4 GHz signal at 100 MS/s with almost zero bit error and shows that radio systems are approaching the ideal software defined radio.

There are also projects which use a large number of these USRPs to simulate multiple interacting nodes in a radio communications network. One of these is the DIWINE project, investigating wireless communication through a dense relay/node cluster. In their final White Paper they described the use of six Ettus USRPs in a testing setup, two source, two relay and two destination nodes in a Smart Meter Network [4]. An alternative to the Ettus USRP is the WARP (Wireless Open-Access Research Platform) Board developed by Rice University. It includes two programmable RF interfaces and a number of peripherals. A demonstration using two WARP Boards for a software-defined visible light communications system was presented by a few researchers [5]. This system was used to demonstrate optical Orthogonal Frequency Division Multiplexing (OFDM). The WARP v3 kit, which contains the WARP v3 Board, power supply and SD card costs *$4,900* for academic customers or *$6,900* for other customers (about *$3,600* and *$5,000* respectively).

The problem with these solutions is that they are all prohibitively expensive, meaning that not as many researchers have access to SDR test beds as they would like.

WARP visible light

WiPi: [6]

You will also need to lay out the basic communication-related tasks that you will focus on. I.e., explain the need to develop this test bed for investigating modulation and coding schemes. The degree of detail you give here is a matter of preference and flow.

1.3 Contributions

The body of this report is structured into three main chapters. Chapter 2, ”The Raspberry Pi and the Test Bed” describes how the Raspberry Pi is set up for its operation in the test bed and what comprises the physical architecture, as well as investigating the code which was developed to test various modulation schemes. Chapter 3, ”Electronic Testing” is aimed at characterising the Raspberry Pi electrically and computationally to understand its capabilities and limitations. It also characterises the physical external components used. Chapter 4, ”Communications Testing” runs through tests for different modulation and coding schemes at different frequencies in order to characterise the test bed in a communications context, as well as discover its limits.

My contributions to the existing literature, what difference I have made (in the third person...) - Write once I have finished writing up tests and results.

Still need to write the Contributions Chapter (last)

Chapter 2

The Raspberry Pi and the Test Bed

The communications test bed consists of a transmitter and a receiver Raspberry Pi, and a number of custom chips arranged on prototyping breadboards. This section details the physical set-up of the test bed, the Raspberry Pi interface and the installation of the custom libraries used. It also covers the technical specifications of the components and the details of the software underlying its operation.

2.1 Raspberry Pi Fundamentals

The Raspberry Pi is a small, simple and affordable ARM-based computing module (Figure 2.1). It has General Purpose Input/Output (GPIO) pins which it can use to interface with external peripherals. The GPIOs can only take binary values of '0' ($0\,\mathrm{V}$) or '1' ($V_{max} = 3.3V$) and so converters are needed to output or input analogue signals. The Pi is run on a Linux-based operating system called Raspbian, which is available for download from the Raspberry Pi official website [7].

The Raspberry Pi 3 Model B+ which is used for this project has a $1.4\,\mathrm{GHz}$ 64-bit quad-core processor The main chip on the board is the Broadcom BCM2835. The Pi's pins can either be numbered sequentially as pins 1 through 40 down the board - this is the BOARD numbering scheme, or they can be referred to by their Broadcom pin numbers - the BCM numbering scheme. The second scheme numbers the available GPIO pins in the range 2-27 (Pins 0 and 1 are reserved for system use).

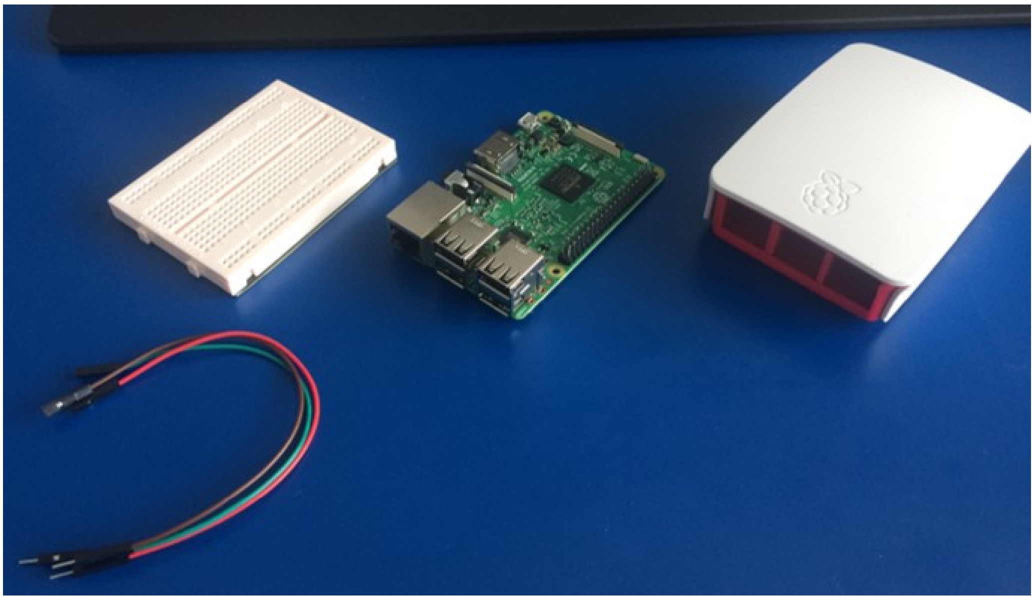


Figure 2.1: The Raspberry Pi with its case, a prototyping breadboard and some jumper cables

2.1.1 Setting up the Raspberry Pi

The Raspberry Pi can be interfaced with in a number of ways, but first it needs to be set up with its operating system on the SD card [8]. Raspbian can be controlled via the command line interface (CLI) - typing in commands - or through an X Window graphical user interface (GUI) similar to Windows Operating System. Either of these options can be used by connecting a screen, keyboard and mouse to the Pi, but this is not always available, so it is useful to have Secure Shell (SSH) set up for the command line, and Virtual Network Computing (VNC) for the X Window. Secure Shell is necessary regardless as it is used in the test bed execution. When actually using both Raspberry Pis in the test bed, both of them will be running in command line mode as this saves resources, although during development, VNC or a screen using the X Window makes it much easier to test and edit code. In the test bed, the Transmitter Pi will be connected to a screen and peripherals in command line mode, and then it will start the Receiver via SSH, which will run headless (without a screen or control peripherals) and receive the data, process and output it, and store information about the run in a log file which can be accessed later or again through SSH.

The setup procedure is shown in Figure 2.2. Once the operating system is written to the SD card, SSH can be enabled by creating an empty file *ssh* in the main portion of the card before putting it in the Raspberry Pi (for setup using a screen and peripherals, this can be turned on in the configuration settings). Now an Ethernet cable can be connected from a computer to the Pi and it can be logged into. A program such as PuTTY [9] for Windows can be used, with *pi@raspberrypi.local* as the host name so the IP address is not needed. The standard login is *username: pi* and *password: raspberry*.

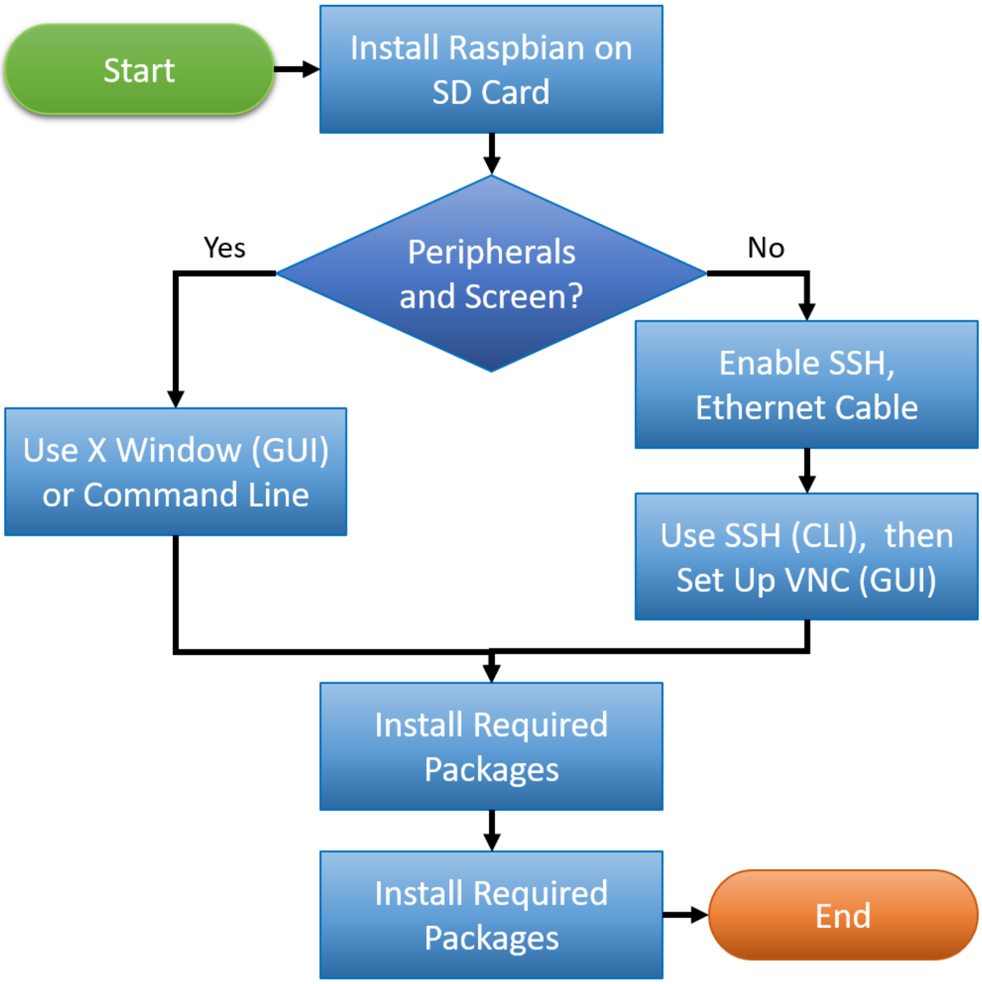


Figure 2.2: The setup procedure of the Raspberry Pi

The command `sudo raspi−config` accesses the configuration settings where the file system can be expanded to the whole SD card, and other utilities such as Wi-Fi and VNC can be turned on. Again, all utilities except for SSH should be turned off when using the final test bed to improve performance of the devices. Once the Pi is connected to Wi-Fi and the IP address is known, it can be accessed remotely via the free RealVNC VNC Viewer [10] and the X Window can be started with the command `startx` if it is not already set to open the X Window on startup in the configuration settings. This access can be made permanent by setting the Pi to have a static IP address or with a free RealVNC account which lets it be accessed independent of the IP address.

With full control of the Pi, all that is needed is to download all of the libraries and software used by the test bed and clone the GitHub repository with its code. The Python version used is 3.5.3. and as the Raspberry Pi has Python 2 and 3, pip - the Python package manager - is called as pip3 for python3. Any dependencies not in this list give clear instructions as to how they are downloaded when this list is run line by line.

```
sudo apt-get update

sudo apt-get install vpnc
sudo apt-get install libffi6 libffi-dev python-dev
sudo pip3 install cryptography paramiko

sudo apt-get install libatlas-base-dev
sudo pip3 install numpy imageio
sudo apt-get install pigpio

sudo apt-get install git
cd "<Path for the code e.g. /pi/home/Documents/>"
git clone https://github.com/CamEadie/4YP_PiCom

sudo apt-get update && sudo apt-get upgrade && sudo apt-get dist-upgrade

# Used in Transmit_Binary_Data(): import RPi.GPIO as GPIO
# ... This library is already included in the Raspberry Pi
# ... Only for OOK transmission which uses Python not C
# Used in Check_Input_Masks(): from RPiSim.GPIO import GPIO
# ... pip GPIOSimulator (or pip3 if necessary)
# ... Only works in Windows, simulates Raspberry Pi pins (use like RPi.GPIO commands)
```

Listing 2.1: Libraries and Packages Required for the Test Bed

Included are the following, as well as their dependencies:

- VPNC - VPN software used to access Oxford VPN in order to use the OWL Wi-Fi network [11]
- Paramiko - Python library used for SSH to start the receiver [12]
- NumPy - Python library used for easier array manipulation as well as interface with images [13]
- imageio - Python library used to read and save image files as NumPy arrays [14]

- pigpio - C library used to interface with the GPIO pins [15]
- Git - Version control software which also allows for the cloning of the project code to any device [16]

2.2 Test Bed Architecture

Include references to the data sheets of all of the components

The test bed comprises the two Raspberry Pis and a number of chips to provide the functions required for more advanced modulation schemes. This is built up as three arrangements with increasing complexity. The first is the Pis connected together by two wires, a serial data line and a clock line (Figure 2.3). This arrangement is similar to that used for an I^2C bus, however the code written for this form of communication does not rely on any available modes of serial interfacing because it needs to be extensible to the parallel communication in the next arrangements. The second arrangement is used for Pulse Amplitude Modulation schemes. It uses a single parallel Digital Analogue Converter (DAC) connected to the transmitter which transmits to a parallel Analogue Digital Converter connected to the receiver, allowing for multi-level signals to be transmitted between the two devices. The final arrangement extends the set-up to two DACs and two ADCs. These signals are then multiplied by a sine wave and a cosine wave respectively, and can be separated due to the orthogonality of the two signals at the receiver. This arrangement is used for Quadrature Amplitude Modulation as well as Orthogonal Frequency Division Multiplexing.

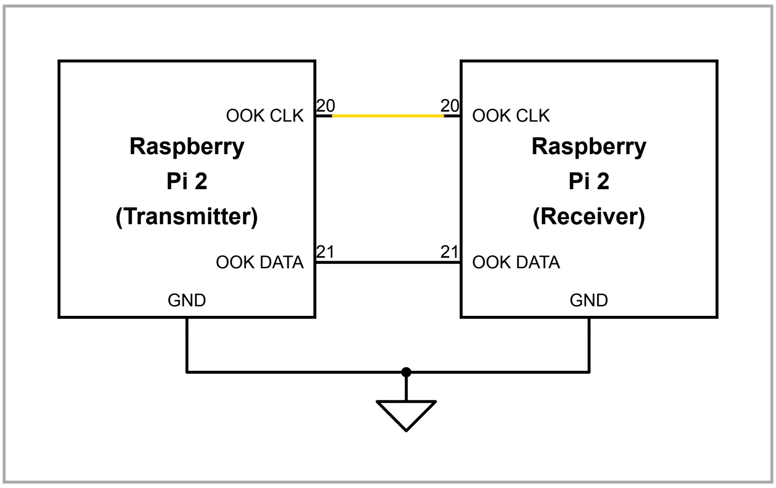


Figure 2.3: Test Bed Set-up For On-Off Keying

2.2.1 Digital Analogue Converter

The Digital Analogue Converter used is the AD5424, an 8-bit CMOS current output DAC with an easy interface to microcontrollers. It has a 17 ns write cycle and a maximum update rate of 20.4 MSPS. There are possible configurations in the data sheet using an inverting operational amplifier to produce the output, however due to the single supply and difficulties finding low-power operational amplifiers which provide rail-to-rail output for a 5V single supply, this was avoided where the output through a load resistor can produce required voltages. The Read/Write ($R/\overline{W}$) pin is pulled low so the chip is permanently in write mode; read back of the parallel digital outputs is not required. The Chip Select pin ($\overline{CS}$) needs a falling edge and a rising edge to complete a write cycle. This is because the DAC has a latched interface; the rising edge loads the new parallel data to be held in the latches and immediately provide the analogue output. The latches are not transparent however (the input is not visible while $\overline{CS}$ is low and the chip is selected), therefore the output holds the analogue conversion of the latched digital byte until the falling edge of its input. This pin is therefore connected to the clock pin of the Raspberry Pi, and the shape of the clock signal used to best utilise the DAC is discussed in Section 2.3.2.1. The same clock pulses are also fed to the $\overline{WR}$ (start conversion) pin of the analogue digital converter. Connections between the transmitter Raspberry Pi and the DAC are in Figure 2.4.

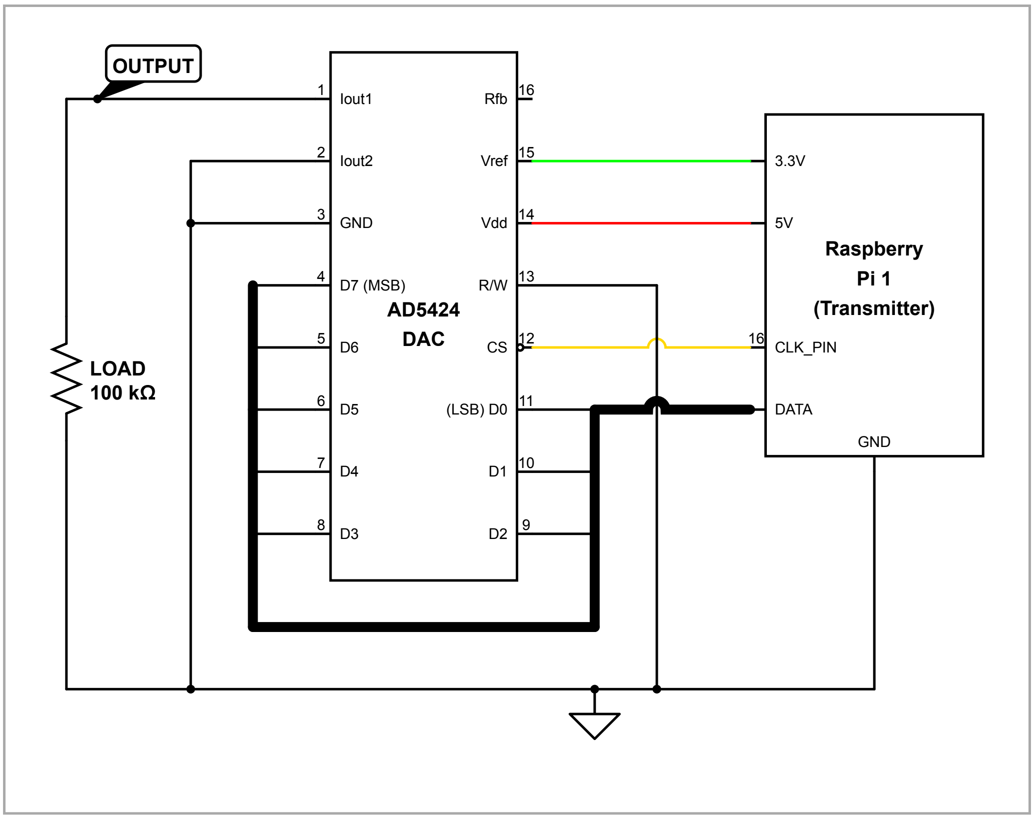


Figure 2.4: Layout of the Digital Analogue Converter

2.2.2 Analogue Digital Converter

The Analogue Digital Converter used is the ZN448, an 8-bit successive approximation ADC designed to be easily interfaced to microprocessors. It operates with an on-chip clock with capability to be over-clocked, and has a 9 µs conversion time guarantee. Figure 2.5 is a diagram of the ADC connections to both Raspberry Pis. The converter is cleared when the $\overline{WR}$ (start conversion) pin is pulled low. On the rising edge of this pin, the comparison of $\frac{V_{REF}}{2}$ to the Most Significant Bit (MSB) set to '1' and all other bits set to '0' is made, and the conversion from analogue to digital values continues by successively setting the next bit, making a comparison and so on. Therefore this pin uses the clock line from the transmitter Raspberry Pi. The $\overline{BUSY}$ (end of conversion) pin goes low when the a conversion starts ($\overline{WR}$ is pulled low), and goes high when the conversion is complete. The positive edge of this signal is taken as the clock for the Receiver, as there is some delay in the ADC's calculation but on the rising edge of $\overline{BUSY}$ the resulting digital value at the output is correct as long as the transmitter is not transmitting so fast as to transmit the next value before the ADC can finish converting the last one. The $\overline{RD}$ (output enable) pin is pulled low to allow non-destructive readout.

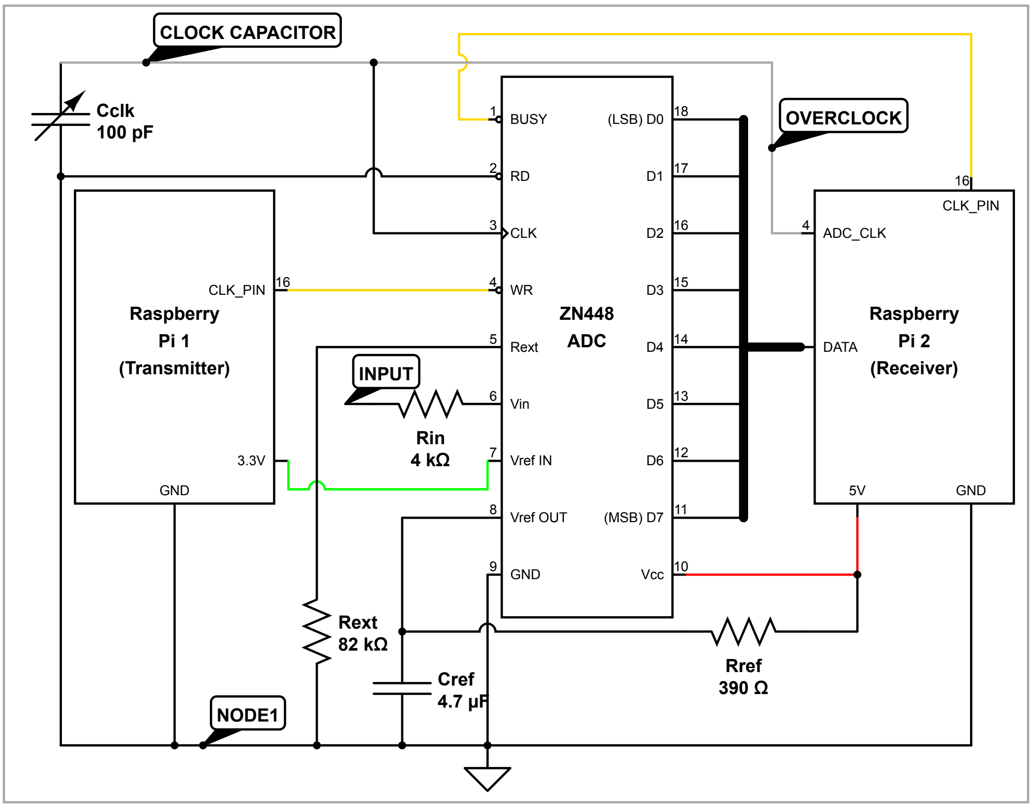


Figure 2.5: Layout of the Analogue Digital Converter

The DAC's internal clock frequency is selected with the CLK pin, either with a single clock capacitor (with an equivalent frequency table provided in the data sheet) or by providing an external clocking signal (see Section 3.3.1). Both options are shown as grey lines in the diagram and only one should be used. The conversion by successive approximation works by approximating one bit per clock pulse over eight clock pulses, and the data sheet suggests due to accepting a completely asynchronous convert pulse with respect to the clock, valid data is produced between 7.5 and 8.5 clock pulses later. The maximum internal clock frequency is 1 MHz and given the guaranteed 9 µs conversion time (0.11 MSPS), a clock frequency giving 9 pulses between conversions is suggested to ensure complete conversion. The capacitor value of 100 pF in Figure 2.5 corresponds to a 900 kHz internal clock frequency, allowing for a sampling rate of at least 0.1 MSPS (or equivalently a baud rate of at least 100 kHz) which is close to the extent of the converter.

Figure 2.6 is a picture of of the two Raspberry Pis connected together for baseband 4PAM communication.

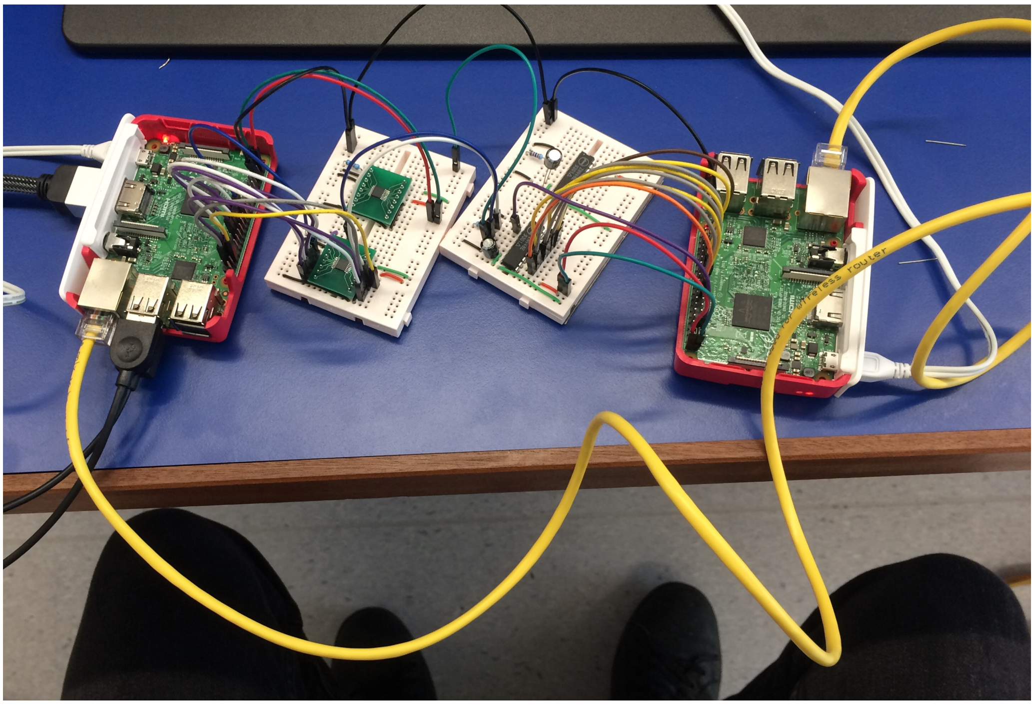


Figure 2.6: Layout of Test Bed for Pulse Amplitude Modulation

2.2.3 Multiplier

This section still needs the technical specifications and layout in the test bed of the multiplier.

Write up Multiplier and sine generator

Depending whether it's doable, include this section without multipliers on transmitter, pseudo ground on output

The Pi is not able to output negative voltages. As a result, Pulse Amplitude Modulation uses the range 0 to V_{max} rather than $-V_{max}$ to V_{max}. Similarly, Quadrature Amplitude Modulation uses a grid of I and Q values all in the positive quadrant (0,1,2,3 not -3,-1,1,3). The same "negative" effect is still achieved by using the "GROUND" of the multipliers (which are fully differential) as $V_{max/2}$ using a voltage divider. This means that the sine and cos signals would be inverted when multiplied by the values 0 and 1 in the same way they would be for -3 and -1, and the transmitted signal will essentially be a normal PAM or QAM signal plus a DC component of $V_{max/2}$.

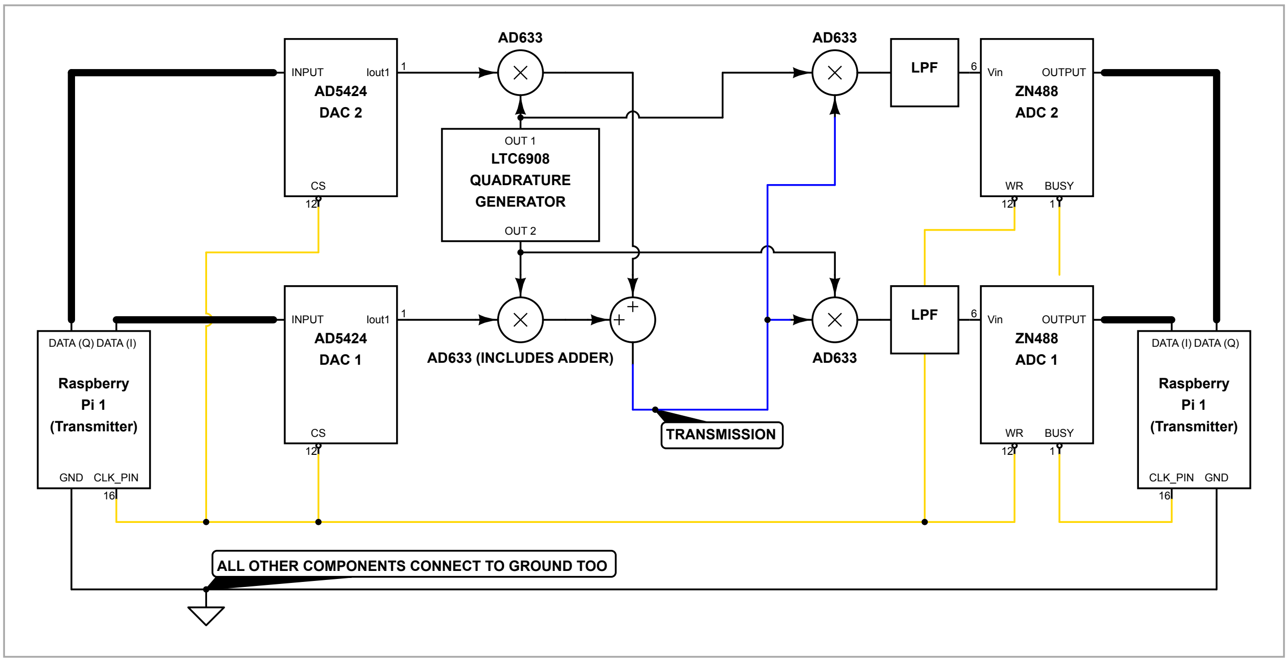


Figure 2.7: Layout of the Test Bed for QAM and OFDM

2.2.4 Quadrature Pulse Generator

x(t) = a(t) cos(t) + b(t)sin(t)

i(t) = 2x(t)cos(t) =

q(t) = 2x(t)sin(t) =

2.2.5 Parts Used

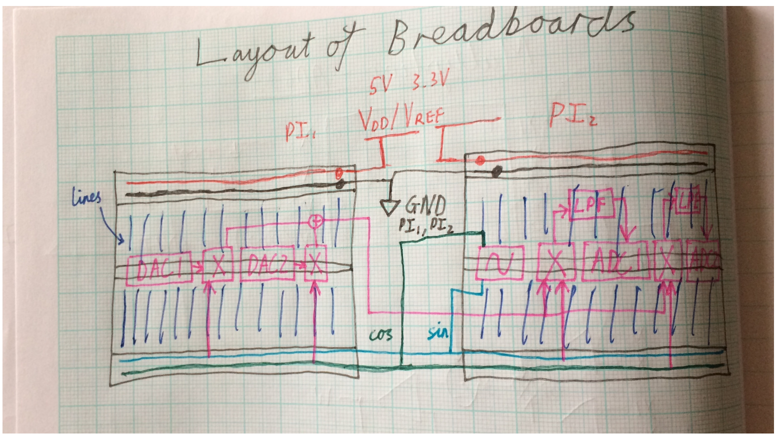


Figure 2.8: The Final Configuration of the Test Bed

Replace this with picture/diagram compound

Table 2.1 is a table of all of the key parts used in the construction of the test bed, excluding small items such as resistors and capacitors. The total price - without Value Added Tax (about 20% on average) and handling charges of individual orders - comes to $\pounds 166.56$. Even with all of the additional components and charges this allows the test bed to be constructed for under $\pounds 250$. This is significantly cheaper than any of the examples considered in Section 1.2. This demonstrates that it is possible to construct a simple software-defined radio communications test bed using two Raspberry Pis at a much reduced cost than by purchasing a standard test bed. Although these test beds may provide certain advanced capabilities, the key functionalities required should be achievable with this design. The amount to which this architecture can successfully achieve these goals is a question investigated by the rest of the report.

Component	Quantity	Price ($\pounds$)
Raspberry Pi Model B+ + Case	2	28.39 + 4.95
Breadboards + Jumper cables	–	21.96 + 11.05
DAC (AD5424) + Surface mount	2	4.19 + 3.36
ADC (ZN488)	2	8.57
Multiplier (AD633)	4	7.29
Quadrature Pulse Generator (LTC6908-2) + Surface mount	1	3.35 + 2.12
Test bed total price	–	165.56

Table 2.1: Prices of components used in the test bed (prices as purchased from RS, Digi-Key, Farnell and Mouser Electronics excluding VAT and handling charges)

Add references to above suppliers

It is worth noting that there is a large variety of available options for each component of the test bed. Each possibility has certain advantages and disadvantages, and a lot of the options are not suitable due to the power requirements or ease of interfacing with the Raspberry Pi. As a result of this, the parts used in this project were the most suitable parts which could be found and successfully sourced. However, there may be more suitable chips available given more time or experience to find them, and being aware of this would be useful if this project were to be extended and/or replicated. All parts could be replaced with minimal adjustment to the physical test bed layout and code.

Changes which would be made with hindsight, if components with the required qualities could be found, are as follows:

Continue to add to this list as you write the Architecture section - remove redundant information already discussed in the Section itself for each part

- A number of chips used are surface mounted, requiring difficult soldering to solder pads. This would be useful if they were to be used on a printed circuit board for a final design, but on a prototyping breadboard, dual in-line packages would have been easier to use where available.
- The Digital Analogue Converter was chosen for its easy interfacing with microcontrollers, but it has a current-output and a voltage-output device with the same characteristics would be preferable. The converter also had issues with correct value output (see Section 3.3) which would need to be fixed before OFDM could be implemented.
- Analogue Digital Converter???
- The Quadrature Sinusoid Generator uses an oscillator chip which outputs 90^o out-of-phase square waves, and the chip used was the only simple one which did this or anything close. Ideally the outputs would already be sinusoidal (one quadrature chip or a sine generator and phase shifter) so that low pass filters with fixed frequency response could be omitted from the design. A component using an input clock could potentially use the Raspberry Pi's hardware clock and make the carrier frequency purely software-dependent. No obvious chip could be found for this and ones that did anything in this area were expensive and complicated.

  Make sure it's consistent with use of Quadrature Sinusoid Generator vs Oscillator in report
- The multiplier is designed to operate around $10\,\mathrm{V}$, and so has a built in $10\,\mathrm{V}$ normalisation in the multiple which attenuates the signal (which is at lower voltages) and requires re-amplification before transmission - an oversight when choosing it. A similar chip designed for lower voltages would be ideal.
- Low Pass Filters to remove high frequency signals in the demodulation part of QAM were again made using fixed-value components, if frequency response of a filter could be altered in software, all frequencies used (carrier and symbol) could be hardware independent.

2.3 Programming

The Raspberry Pi is used for its low cost, ease of use, and the fact that it has programmable Input/Output (I/O) pins. The I/O pins can be programmed using different libraries in either Python or C. The standard GPIO library which comes installed with Raspbian is RPi.GPIO for Python [17]. This is used for the On-Off Keying part of the communications test bed. Python is relatively slow however, and so a C library is used for the pin-level manipulation for all modulation schemes requiring multi-pin outputs to Digital Analogue Converters to generate multi-level signals. This is done both for the improved speed performance of the C library, and the capability of this library to output to multiple pins at once. Section 3.2.1 goes into a detailed investigation of the differences between the libraries used and the reasons for choosing C over Python for the advanced modulation schemes. All of the code and the report for the project are maintained on GitHub, and may be found at `https://github.com/CamEadie/4YP_PiCom`.

The transmitter and receiver code for all modulation schemes considered works from a single final version of the test bed code. This comprises of the Python transmitter `PiComTx_5_DAC.py`, and receiver `PiComRx_5_DAC.py`, as well as the executables compiled from C code for the transmitter `PiTransmit_3`, and receiver `PiReceive`. The main function in the Python files for each the transmitter and receiver is split into On-Off Keying and Advanced Modulation Schemes. On-Off Keying (Section 2.3.1) is the simplest form of clocked communication, and acts as a proof of concept for the Raspberry Pis as a test bed. It is implemented using Python lists to store '1's and '0's to represent the binary stream. These are output using the native RPi.GPIO library.

The OOK part was added into the final code for the Advanced Modulation Schemes (Section 2.3.2) from previous versions retrospectively, as all of the Advanced Schemes are implemented in the same code. This was done to make it easier to conduct all communications tests through a single program interface. It also allows all of the modulation schemes to be used on essentially the same layout with minor changes to the hardware, and this adheres better to the idea of Software Defined Radio being as software-defined and hardware-change-independent as possible (Section 1.2).

> Rephrase this and make sure the section referenced is as consistent with this comment as possible - reference that paper or lit review

The advanced modulation schemes considered are 4-level Pulse Amplitude Modulation, 256-level Pulse Amplitude Modulation (used more for setting up the DAC and ADC, as differentiating between levels this precisely is not viable), 16-Quadrature Amplitude Modulation and Orthogonal Frequency Division Multiplexing. Code for this section implements the data stream as bytes in a *NumPy* array rather than as bits in a list. This has certain computational advantages but also allows for the integration of image handling with *imageio*. This means that images can be transmitted instead of random data, allowing for visualisation of the bit error rate etc. of the transmission. These schemes also use a separate compiled C module for the actual transmitting and receiving of data.

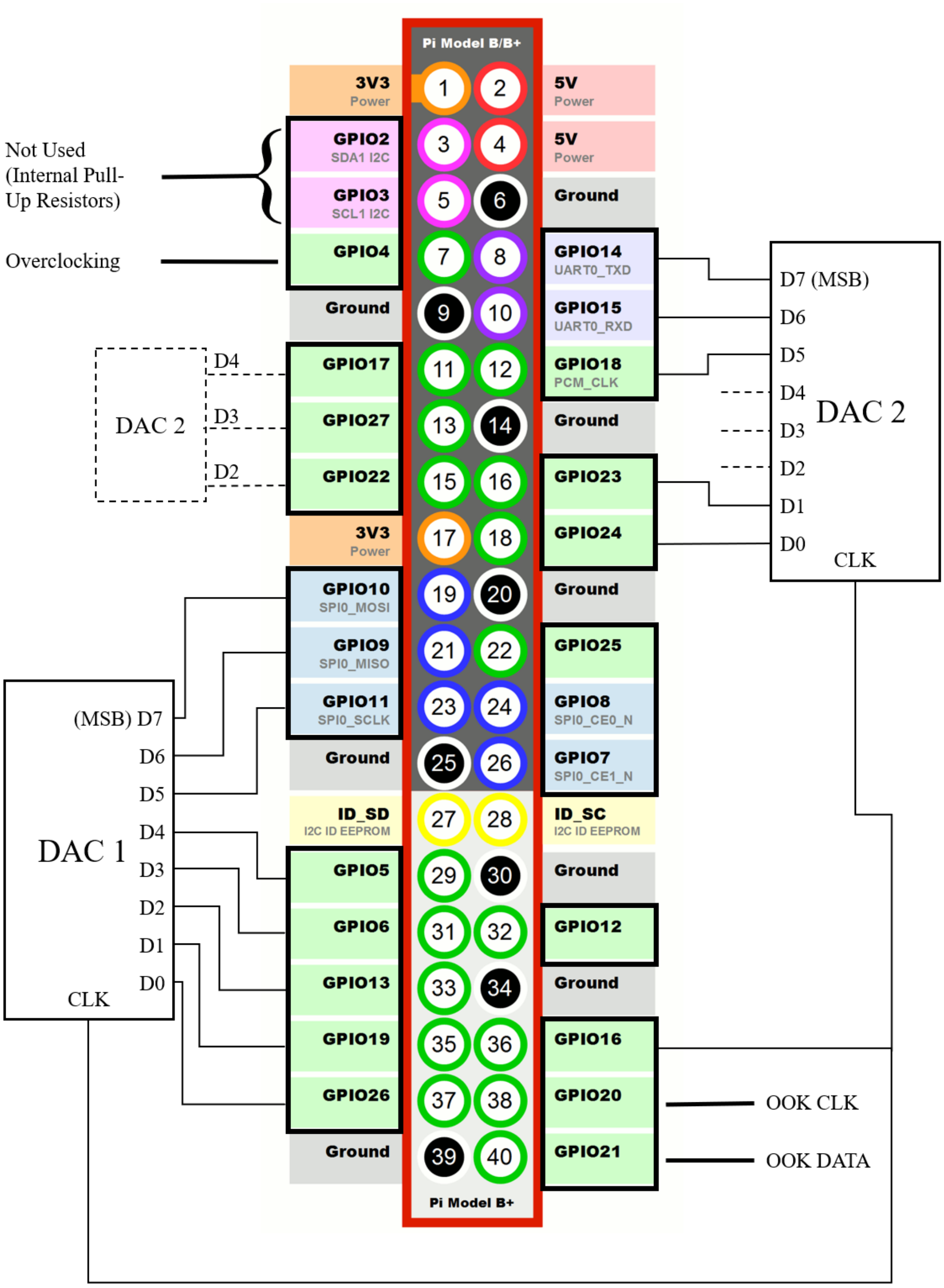


Figure 2.9: Pin Diagram of Connections to Raspberry Pi (reference to image source) - These are alterable fairly easily in the code so changes can be made with little overhead

Pins used currently for the OOK transmission as well as for the DACs are shown in Figure 2.9. The pins with black boxes around them are the accessible GPIO pins, and they are referred to by their BCM number (GPIO# in the boxes) rather than their BOARD number (consecutive numbers in the circles). Of particular interest due to their differences are pins 2, 3 and 4. Pins 2 and 3 are used for the clock and data lines of I^2C bus communication and so have internal pull-up resistors. Due to the fact that the test bed uses pins set with pull-down resistors, these pins aren't used. Pin 4 is the only pin on the Raspberry Pi B+ with access to a hardware clock which can be programmed for external use. It is thus used for overclocking external components which can be fed a clock signal, which is discussed in Section 3.3.1. The code defines the pins used for transmission as global variables at the start so that the rest of the code can be pin-independent. Excepting the the DAC clock pin (pin 16) which is defined in the C code, Listing 2.2 shows these pin definitions.

```
# OOK Pins
CLK_PIN = 20
DATA_PIN = 21
# DAC Pins
DAC_PINS_1, DAC_PINS_2 = [10, 9 , 11, 5 , 6 , 13, 19, 26], \
[14, 15, 18, 17, 27, 22, 23, 24]
```

Listing 2.2: Pins used for OOK and the DACs

2.3.1 On-Off Keying

On-Off Keying is a modulation scheme based on using V_{max} as '1' and 0 V as '0'. The transmit section of the test bed loops through the data list, outputting each value to the data pin followed by a clock pulse using the sleep function. The speed and accuracy of the *GPIO.output()* and *sleep()* functions is covered in Section 3. The receiver uses the function *GPIO.wait_for_edge()* on rising edges of the clock pin to trigger reading of the data pin. Using a function which is polling the clock pin as opposed to interrupt-driven solution is not ideal, but it was more easily written and so used for this early-stage modulation scheme.

Earlier versions of this modulation scheme included a function *Prep_Binary_Data()* which added initial and final padding to the data of '1's to prevent missing timing of the beginning and end of transmission. The function also added a padding bit to the data when either value had been repeated a certain number of times (for example a '1' if '0' had been repeated 5 times in a row). This function was removed when *Encode_Error_Correction()* was included, as forward error correction was seen as a better way of avoiding these and other errors without including padding bits which may be difficult to remove if errors did occur in the code transmission. The channel coding used is syndrome decoding and is discussed more in Section 4.3.

2.3.1.1 Starting the Receiver

The receiver is started with the library Paramiko [12] which is used to make an SSH connection, and then to execute a command on the device which it connects to. The article "Using paramiko to send SSH commands" by Sebastian Dahlgren was particularly useful in understanding how the library is properly used [18]. The host names of the transmitter and the receiver were changed to *raspberrypi1* and *raspberrypi2* to differentiate them, and their passwords changed to *rasPass1* and *rasPass2*. The devices are connected by an Ethernet cable so no connection to the internet is required, and so the transmitter can use the local host name of the receiver *raspberrypi2.local*.

The command to start the program is:

```
command = "sudo python3 " + \
"/home/pi/Documents/4YP_PiCom/4YP_PiCom_Receiver/PiComRx_5_DAC.py" + \
" " +str(mask_length) + " " +str(TRANSMISSION_TYPE)
```

Listing 2.3: Command Line to Start the Receiver

The super user call *sudo* is necessary for control of the GPIO pins in the receiver code. When programs are run from a command in command line mode, it can have additional information to it through the use of command line arguments. These are additional space-separated strings after the name of the program which can be accessed by the program. The command line argument *mask_length* is required by the receiver, and ensures that the amount of data received is the amount expected, and allows for the checking of deletions in the transmission. The other argument *TRANSMISSION_TYPE* is not required, but if passed it overwrites the transmission type in the receiver code to ensure that it is expecting the same modulation scheme that the transmitter is using.

The SSH connection is closed as soon as the command is executed so that both Raspberry Pis are not expending resources during transmission, and any readout to *stdout*, the standard output of the program over the connection is ignored. All logging of the receiver is instead appended to a Python list *LOGS*, and this variable is written to a file *LOGS.txt* at the end of execution. This includes all of the trivial to calculate results of the post-transmission analysis such as the number of bits/bytes received (whether there was any data lost).

2.3.1.2 Data and Image Handling

NumPy is a package used for scientific computing in Python. Its main feature is an N-dimensional array object with various functions for reshaping and efficiently computing on large arrays. These arrays are used to store the data to be transmitted while it is converted from byte-wise data into bitmasks which can be transmitted by the DACs and vice versa on the receiving end. Scalar operations can be performed on an entire NumPy array which allows the inverting of all of the bitmasks of the transmission data by XOR to be expressed simply as `masks ^ DAC_MASK_1`. The arrays also have stronger typing than regular Python, so an array can be set to have data types such as unsigned 8-bit integer (byte, `dtype='uint8'`) array or unsigned 32-bit integer (bitmask, *dtype='uint32'*) array and Numpy will ensure that its elements remain in that type. The library is imported *import numpy as np* so all of the library functions can be called with the standard *np.* shorthand such as *np.array()*. Functions of particular interest are *np.unpackbits()* and *np.packbits()* which allow you to unpack a size-N array of M-bit integers into a size-($N \times M$) binary array of the bits of each integer, which makes switching from byte-wise data to bit-wise operations and back very simple. NumPy also has *np.fft.fft()* and *np.fft.ifft()* for computing Fast Fourier Transforms and Inverse Fast Fourier Transforms on data which makes OFDM easier to implement.

The library imageio is designed as an easy interface to read and write images in Python. Reading in an image is as simple as *img = imageio.imread(path, pilmode = 'RGB')* which saves the pixel values of the image at *path* into a size-(x, y, 3) NumPy array in the variable *img*. Different *pilmode* settings can be used to read the image as black and white *pilmode = 'L'* or in another colour palette. Saving an image is just as simple with*imageio.imwrite(path, img)*. Using imageio the transmitted data can be image files which are more interesting to work with than random bit-streams, and any significant number of errors will be visible in the output.

2.3.2 Advanced Modulation Schemes

This section will start by describing the advanced modulation schemes used, and will then it will go on to how the transmitter and receiver implement the schemes in code.

The modulation schemes used are 4-level Pulse Amplitude Modulation (4PAM), 256-level Pulse Amplitude Modulation (256PAM) and 16-Quadrature Amplitude Modulation (16QAM). Orthogonal Frequency Division Multiplexing (OFDM) is discussed as a possibility but not implemented. 4PAM expresses each bit pair as one of four voltage levels which is expressed with use of a Digital Analogue Converter (DAC). 16QAM uses a similar concept except it transmits two voltages at once to describe each set of four bits. This is achieved using two DACs, and by multiplying the output of one by a sine wave and the other by a cos (or 90^o phase-shifted sine) and adding them, using the orthogonality of the two waveforms to extract each level independently at the other end. OFDM is a scheme which uses sub-carrier modulation in the digital domain, where each sub-carrier is modulated with a conventional modulation scheme such as Phase Shift Keying or QAM. It then transmits the Inverse Fast Fourier Transform (IFFT) of the combination of these sub-carriers as a complex carrier-modulated signal.

The communications test bed could be implemented with OFDM, however there were certain issues and time constraints which meant that this was not realised. The details of how the test bed would be extended are included. As mentioned in Section 2.2.3, the Raspberry Pi is unable to output negative voltages, and so all symbol values are in a positive voltage range between $V_{min} = 0V$ and $V_{max} = 3.3V$. Therefore, 4PAM and 16QAM use values from the range $\{0, 1, 2, 3\} \times \frac{3.3}{3}V$, output from one and two DACs respectively. Similarly 256PAM uses 256 values from $\{0, 1, ..., 255\} \times \frac{3.3}{255}V$.

Figure 2.11 (see Page 31) is a flow chart of the operation of the transmitter code. 256PAM is omitted in the flow chart as it is similar to but more easily realised than 4PAM, and the space is better used conceptualising the addition of OFDM. 256PAM is also not a viable modulation scheme unless the DAC acts perfectly, and so it is implemented mostly as a way of testing the accuracy of the DAC's granularity. Tests using 256PAM to output monotonically increasing values revealed that the converters were not working exactly as expected, and that there were spikes at certain values and a non-continuous analogue response was measured for continuous digital input. This problem is investigated in Section 3.3, but the solution was to use an empirically derived lookup table to get the four levels for 4PAM and 16QAM as close as they could be to the correct values – this would be removed from the code if a successful replacement digital analogue converter was used.

Make sure the corresponding section (USE THE RIGHT SUB-SECTION) does reference the fact that the DAC doesn't work exactly as it's supposed to and how I dealt with it

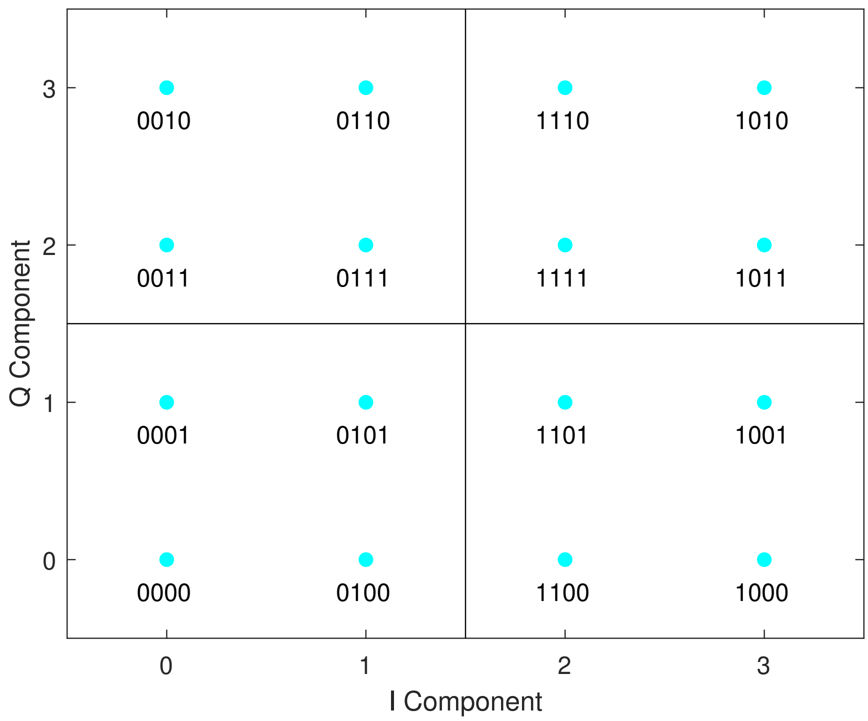


Figure 2.10: 16-QAM Constellation with Grey Code

The transmitter can be run from the command line with transmission frequency and modulation scheme as command line variables, otherwise it uses the currently coded in values. The data is taken as NumPy arrays of bytes (8-bit unsigned integers). This is either pseudo-randomly generated or taken as pixel values from an image file using imageio. For 4PAM, each byte is split into four symbols, and each symbol (of value 0, 1, 2 or 3) is converted into an 8-bit digital value. As mentioned above this is done with a lookup table, but would ideally be done by multiplying each symbol by $\frac{255}{3} = 85$. The values are then expanded into bitmasks which can be used by the bank write operation in the C code. This is explained in below in Section 2.3.2.1. For 16QAM each byte is split into two symbols, where each symbol consists of an I and a Q component (of value 0, 1, 2 or 3). These symbols are chosen using the grey coded 16QAM constellation in Figure 2.10. Grey code is a binary system where any symbol has only one bit difference to all directly adjacent symbols, and it is chosen because if a symbol is incorrectly decoded, the next most likely symbols are directly adjacent and so only one bit will be incorrect, minimising the bit error. By inverting the constellation, a dictionary of 4-bit values to symbols is used to easily convert each byte to two symbols. This is done by unpacking each byte into corresponding bits and reshaping into a matrix with four columns, then each row is taken as a dictionary key to decode the symbol, shown in Listing 2.4. Each symbol is then converted into an 8-bit DAC value and expanded onto a bitmask for the C code, with the I component mapped to the first DAC's pins, and the Q component mapped to the second DAC's pins.

```
mapping_table = {
    (0,0,0,0) : 0 + 0j,
    (0,0,0,1) : 0 + 1j,
    (0,0,1,0) : 0 + 3j,
    (0,0,1,1) : 0 + 2j,
    (0,1,0,0) : 1 + 0j,
    (0,1,0,1) : 1 + 1j,
    (0,1,1,0) : 1 + 3j,
    (0,1,1,1) : 1 + 2j,
    (1,0,0,0) : 3 + 0j,
    (1,0,0,1) : 3 + 1j,
    (1,0,1,0) : 3 + 3j,
    (1,0,1,1) : 3 + 2j,
    (1,1,0,0) : 2 + 0j,
    (1,1,0,1) : 2 + 1j,
    (1,1,1,0) : 2 + 3j,
    (1,1,1,1) : 2 + 2j }
# Unpack data list into N bits and reshape to (N/4 x 4) bit matrix
data_list_bits = np.unpackbits(data_list)
```

```
data_list_bits = data_list_bits.reshape( (data_list_bits.size//4, 4)
)
# Convert (N/4 x 4) matrix into N/4 symbols
def Mapping(bits):
return np.array([mapping_table[tuple(b)] for b in bits])
symb = Mapping(data_list_bits)
```

Listing 2.4: Inverted QAM Constellation and its use to encode each byte as two symbols

OFDM is also conceptualised here but due to the DAC continuous value issue mentioned above, a better-functioning DAC would need to be found before this could be implemented. For OFDM, the data stream is split into N sub-streams, each transmitted on a separate sub-carrier. These sub-carriers are chosen to be orthogonal so they do not interfere with each other, normally achieved by having each sub-carrier centred at integer multiples of the desired frequency gap δF. The transmitter takes N complex symbols from the conventional modulation scheme - 16QAM for example - at a time, and puts them through an IFFT with N inputs. The complex output is then transmitted as a complex carrier-modulated signal, meaning that the real output of the IFFT would be modulated by a sine wave and the imaginary output by a cos wave at a chosen carrier frequency in the same manner as the I and Q components of QAM. NumPy has functions *numpy.fft()* and *numpy.ifft()* which make the transition from 16QAM to OFDM as simple as splitting the signal into multiple streams before converting the data into QAM symbols, and then using the symbols as inputs to the IFFT. Adding a cyclic prefix to each block of IFFT samples to prevent inter-block interference, ensuring the outputs are expressed as 8-bit digital outputs and mapping the values to a set of bitmasks would still need to be included.

For all modulation schemes considered, once the output is expressed as an array of bitmasks - essentially 32-bit binary numbers where each bit corresponds to a pin to set - each element in this array is exclusive bitwise OR-ed (XOR-ed) with the bit mask of the two DACs' address pins. This results in another array of bitmasks where all of the pins in the DAC bitmasks are inverted, and all the other pins remain zero. This is necessary because the C library has two bank operations (which work on a bank of pins), *gpioWrite_Bits_0_31_Set()* and *gpioWrite_Bits_0_31_Clear()*. As the name suggests, one is required to set all of the pins which will be set to '1', and one is required to clear all of the bits which will be set to '0' but may have been '1' before. Both arrays are saved as binary files, so they can be read in by the C transmitter code. At this point the command to start the receiver is passed through an SSH link to the receiver and the connection is terminated. The receiver is started with command line arguments of number of masks (symbols) and the transmission type. After this, the C transmitter is started with one command line argument, mask (symbol) transmission frequency, otherwise known as baud rate. The C transmitter sets up the pins for the transmission, reads in the binary files corresponding to the *Set* and *Clear* bitmasks, and then loops through, outputting the values to the DACs at the specified baud rate.

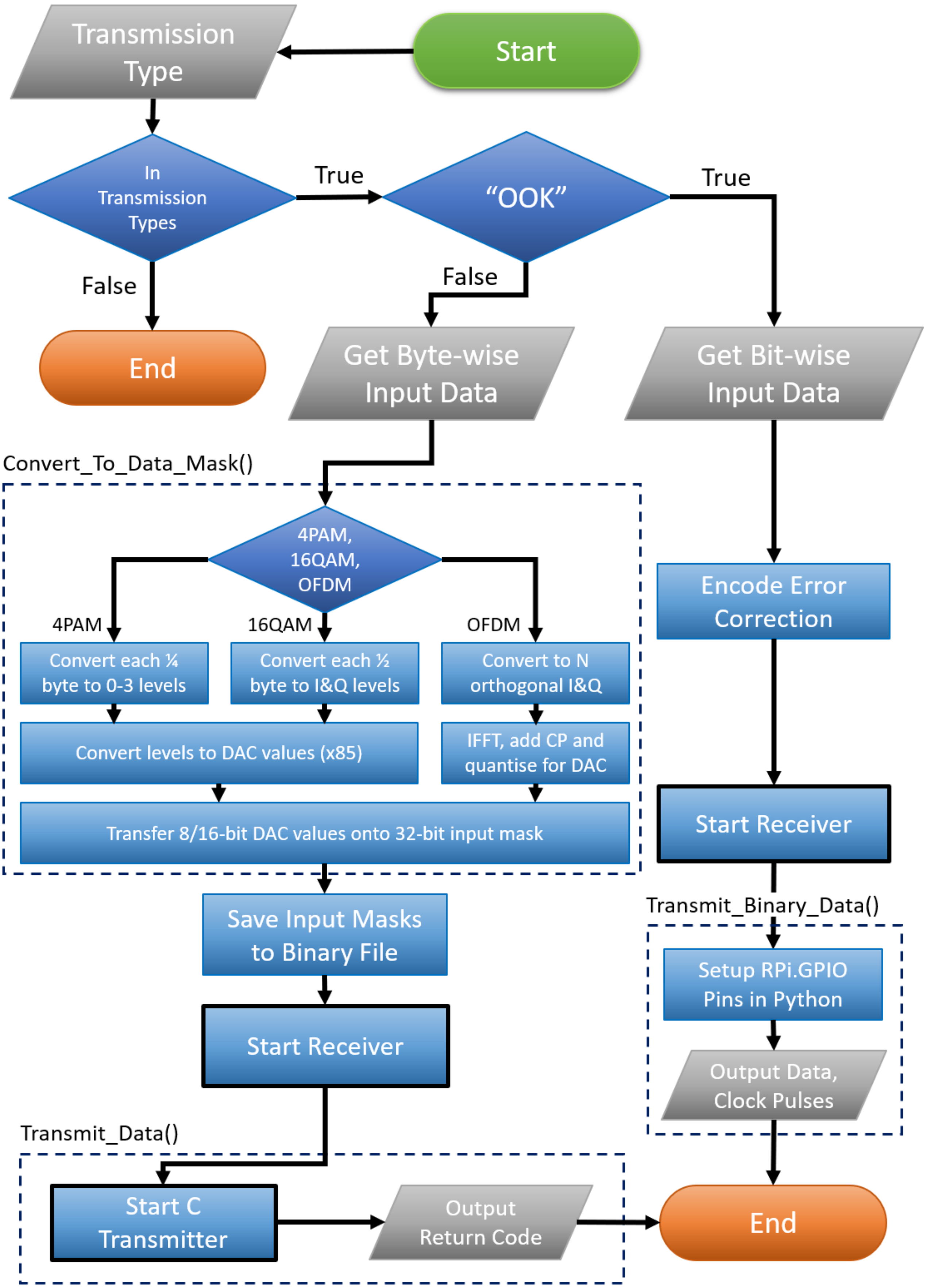


Figure 2.11: Flow Chart for Transmitter Code

Figure 2.13 (see Page 34) is a flow chart of the receiver code's execution path. The receiver starts, and if the transmission type is not OOK and thus an advanced modulation scheme, it immediately starts the C receiver code passing the command line argument of number of masks so that it knows how much data to expect. This part of the receiver saves one bitmask per clock pulse until it has received the expected amount of data. There is also a way of checking if the transmission has finished, when the clock stops, so that if data was missed at the beginning of or during the transmission, the receiver will still stop listening and complete execution. The C code then saves the bitmasks received to a binary file and terminates successfully.

If the C receiver is successful, the binary file is read into a NumPy array to be decoded. For all modulation schemes, the first step is to de-map the data from the masks to their DAC value(s) using the list *DAC_Pins_1* as well as *DAC_Pins_2* for quadrature-carrier modulation schemes. The receiver then measures the peak values of the signal and if there has been any attenuation of the signal, it is scaled to the full input range and the attenuation is logged. PAM and QAM then use Maximum Likelihood (ML) decoding to figure out the correct symbol values. This equates to using partitions which are half way between possible symbols to decode them. For example in 16-QAM, any value within the box around each symbol in Figure 2.12 would be mapped to the symbol at its centre.

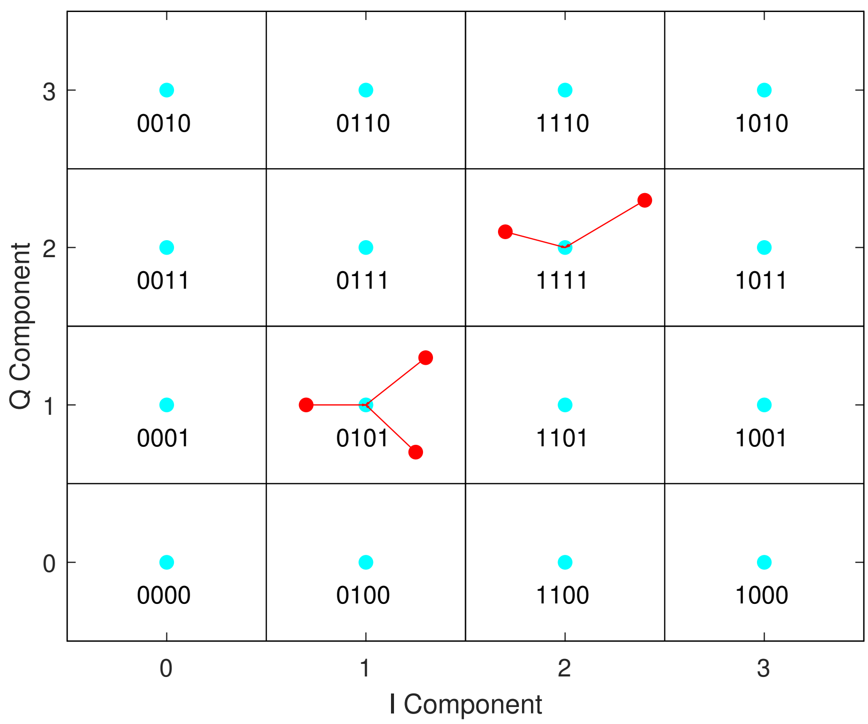


Figure 2.12: 16-QAM Constellation with Maximum Likelihood Boundaries

Once the input has been decoded into the ML estimated symbols, it must be recombined into byte-wise data. For 4PAM this constitutes an OR of the four symbols of a byte, each symbol bit-shifted two bits to the left (multiplied by four) of the next. For QAM, the mapping table of Listing 2.4 is inverted and each symbol de-mapped into four bits. The bits are then packed into 8-bit numbers with the command *np.packbits(output_bits)*. If the transmission was of an image file, the receiver saves the received data into an image file of the same size, otherwise it saves it to a file where it can be compared to the original data.

Write the C Sections (and if time, the test masks code)

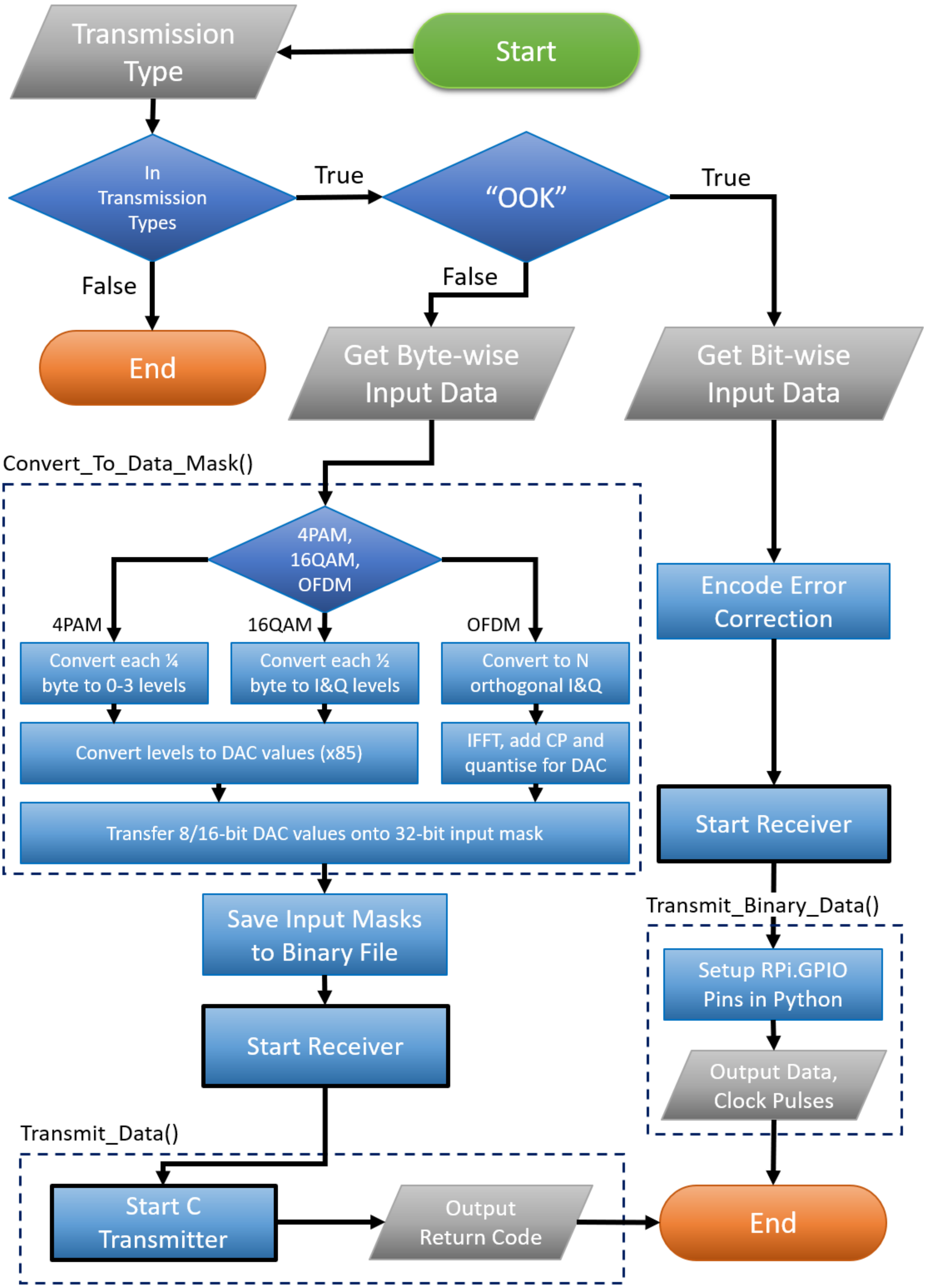


Figure 2.13: Flow Chart for Receiver Code

2.3.2.1 C Transmitter and Receiver

The C library used for manipulation of the GPIO pins is pigpio [15]. This library uses direct memory access to read the values of the pins, and has functions for reading in or writing to a bank of values as a bitmask. A bitmask is an unsigned 32-bit integer (in this case) where each bit corresponds to a pin value being '1' or '0'. The mask uses BCM numbers, and although there are 32 pins which can theoretically be accessed, only BCM pins 2-27 are available for user purposes. The Python transmitter expands the required DAC bit values (0-7 for 8-bit output) onto the 32-bit mask depending on which GPIO pins the DAC is connected to. An simplified example of this mask expansion is shown in Figure 2.14. As Section 2.3.2 explained, an inverted mask is also required to clear the '0'-value bits from previous values. Section 3.2 compared the speed of the individual pin read to the bank read operation, and the speed of the individual pin write to a bank set and clear, and in both cases the bank operation is faster - likely because of the direct memory access - as well as reading or setting all required pins at once. As a result, all of the C code uses these mask operations to set and read the pins.

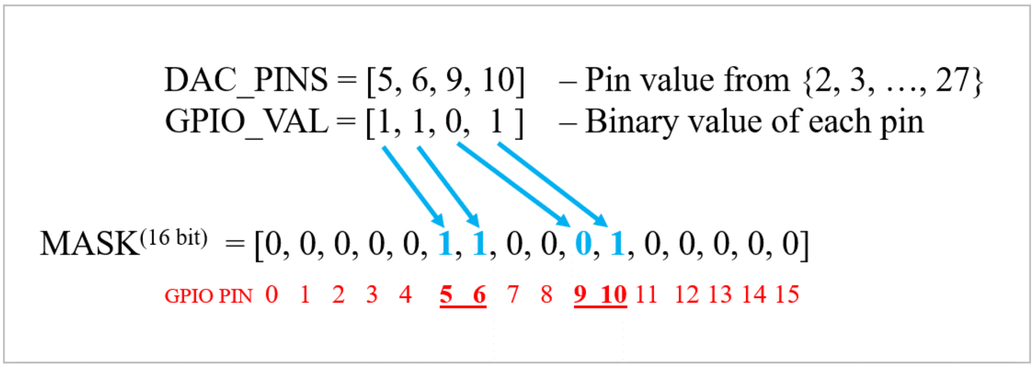


Figure 2.14: Explanation of the concept of mask expansion

Another advantage of using bitmasks is that *PiTransmit_3* and *PiReceive* can operate entirely independently of the choice of which GPIOs the DACs and ADCs use, or even whether the modulation scheme is using one or two of each. The transmitter reads in the array of bitmasks (bits to set) from a binary file, which is calculated and saved in the Python before transmission. It also reads in inversion mask (bits to clear) so this calculation is not done during transmission. The clock pulse goes low with the data and then high 1 µs later, going high again after the rest of the clock cycle. Listing 2.5 from the transmitter comments explains the reason for the clock shape. The transmitter loops through the data masks, bank setting (masks) and clearing (inversion masks) the values for the DAC input and outputting the clock signal, then it closes on completion.

```
/* Clock cycle will have shape:
* |_|-------------|_|-------------|_|-------------|_|-------------
* The pulses will be low for 1us ^ and high ^ for the rest of the clock period
* ADC:
* Rising _CS_ latches data but read event requires falling then rising edge
* It holds the output at the latched value until the next falling edge
* DAC:
* _WR_ going low clears the converter and _WR_ high starts the conversion
* _BUSY_ goes high when conversion is complete, and is used as Rx clock pin
*
* A 1us pulse is long enough for both converters to use as a trigger (100 ns at least)
* It is also short enough that at max frequency (100kHz) it is still a fraction of clock
*/
```

Listing 2.5: Transmitter documentation explaining the clock signal

The receiver starts by allocating a memory block for the number of masks expected. It then sets up two callback functions, *readPins()* and *checkPins()*. *checkPins()* is set up as a callback on the change of state of all of the ADC pins, and all it does is update a global variable *pin_state* to the output bitmask of *gpioRead_Bits_0_31()*. *readPins()* is setup as a callback on the change of state of the clock (ADC $\overline{BUSY}$) pin. It saves *pin_state* to a local variable, then if the event triggered was a rising edge, it saves this value into the output array. A static counter keeps track of the data received, and when it reaches the number of masks expected, closes the callbacks on all pins and changes the global *pin_state* to a specific value which it can't reach on its own. The callbacks are also closed and this specific value set if there is a watchdog time-out – the callback is called because there has been change in the clock pin for a certain amount of time.

Once the receiver has set up the callbacks, it uses a while loop to wait until the specific value is set to the *pin_state* signifying the end of the program. After this the received data masks are saved to a binary file which can be opened by the Python code and mapped to ADC values to decode the transmitted data.

Chapter 3

Electronic Testing

Assessing the electronic capabilities of the test bed is split into three main sections. The first and second sections deal with the electrical and computational characteristics respectively of the Raspberry Pi, and the third deals with the components used in the test bed. The electrical section pertains to the GPIO pins and their physical capabilities, whereas the computational section considers how fast the pin values can be changed and read by the underlying code, and how most efficiently to achieve this. The final section discusses limitations to the operation of the test bed which are posed by the components used.

3.1 Electrical Characteristics of the Raspberry Pi

The Raspberry Pi 3 Model B+ has 40 pins, including eight ground pins, two $3.3\,\mathrm{V}$ power pins and two $5\,\mathrm{V}$ power pins. 26 of the remaining pins (BCM pins 2 to 27) are free to be used as General Purpose Input/Output (GPIO). The GPIO pins can output low ($0\,\mathrm{V}$) and high ($3.3\,\mathrm{V}$) levels, and they are powered by the same $3.3\,\mathrm{V}$ rail as the power pins of the same voltage. As a result, there is a maximum current that can be drawn from all of these pins together as well as from each GPIO pin individually.

The Embedded Linux Wiki [19] claims that the $5\,\mathrm{V}$ pins can provide a maximum current equal to, ”The USB [power cable] input current (usually $1\,\mathrm{A}$) minus any current draw from the rest of the board.” It also provides the maximum current to be drawn from all $3.3\,\mathrm{V}$ power pins as $50\,\mathrm{mA}$ (this would apply to the power pins and the GPIO pins combined). However that specification was actually a design value for the original Pi, designed to supply $3\,\mathrm{mA}$ for each of its17 pins for $\approx 51\,\mathrm{mA}$ total, according to Gert Van Loo, the hardware engineer of the first Pi's boards [20]. There is no current-limiting on the pins so they will attempt to drive the current pulled they stops working, however multiple sources including Gert Van Loo suggest that the maximum current that should be drawn from any one pin for safe operation is 16 mA as this is the current to which the electronics on each pad are rated. Mosaic Documentation Web also has a page attempting to define the electrical specifications of the Raspberry Pi [21], and it also suggests that one shouldn't attempt to source or sink more than 16 mA on an output pin. The pins actually have a set drive strength from 2 mA – 16 mA in 2 mA increments which is set for a bank of pins and usually set to 8 mA but even a pin set to 2 mA drive strength and then loaded so as to draw 16 mA will not be damaged [22].

The maximum current which can be drawn from an output pin is a useful detail. This is both in order to ensure that none of the attached components draw too much current, as well as to decide, along with the input impedances of the pins, whether or not the GPIO pins of one Pi can be connected directly to another without a protective resistor between them. It is important that the ground pins of the two Raspberry Pis must be connected together so that they share a common reference for the data levels. The input and output impedances of the pins in various set-up modes are shown in Table 3.1. All values were measured on a multi-meter and using the pin GPIO26 and compared to other pins to check consistency. Pins 2 and 3 have permanent internal pull-up resistors whereas all of the rest have software-controllable pull-up or pull-down resistors which can be set for inputs.

Input/Output Mode	Impedance to Ground (kΩ)
Input	Open Loop
Input with Pull Up Resistor	53 250
Input with Pull Down Resistor	49.24
Output	0.0329
Raspberry Pi ON (No Mode)	49.23
Raspberry Pi OFF	606.14

Table 3.1: Table of GPIO Pin Impedances for Different Operating Modes

These impedances show that even the lowest input impedance of 48.93 kΩ will only draw 67.44 µA of current from a 3.3 V GPIO pin, and this is three orders of magnitude lower than the maximum current these pins can supply. The impedance of pins set to be outputs is the only value lower than this at 32.9 Ω, and so if an output at 0 V were connected to another pin at 3.3 V it would potentially draw about 100 mA which would damage both pins or at least (the Raspberry Pi has safeguards excessive currents) restart the Pi. This is unlikely to happen in the test bed setup however, unless an output pin was directly connected to a 3.3 V power pin, or both Pis had directly connected pins set as outputs. In conclusion, the pins of two Raspberry Pis can be directly connected together, as long as care is taken not to have both devices setting these pins to outputs at the same time .

3.2 Computational Characteristics of the Raspberry Pi

Write Computing and Components Sections - I have all the data, screen shots and component characterising values

The Raspberry Pi is not a real-time device. Running under an operating system, there will always be interrupts due to scheduling of different threads, which mean that code execution may not always be as perfectly timed as it would on a dedicated embedded device like an Arduino. The Raspberry Pi has certain advantages however; Python is significantly more versatile for signal processing, and the Raspberry Pi B+ has a 1.4 GHz processor compared to the Arduino Uno's 16 MHz clock [23]. There are also ways of ensuring the Pi operates as close to real time as possible.

The time-sensitive parts of the code are the actual pin manipulations in the transmitter, and the reading in of the pin values in the receiver. This means that these need to be the parts optimised for speed.

System interrupts can be turned off in the code, however this results in nothing else being run on the computer, essentially freezing it until the interrupts are turned back on [24]. This should not be done for too long as it is bad for the Pi, and as the transmitter and receiver may be used to transmit a significant amount of data, this is not suggestible without periodically turning on interrupts to allow them to run, losing samples at the receiver to the backlog of interrupts. However, there is one particularly inhibiting interrupt - adjusting of the refresh rate of RAM every 500ms - which can be turned off [25]. This is achieved with the terminal command in Listing 3.1.

```
sudo sed -i '$s/$/\ndisable_pvt=1/' /boot/config.txt
```

Listing 3.1: Turning off the RAM refresh rate adjustment

DIWINE White Paper 2 [26] at the end of the second year in a three year project, they still had "'time zero' reference" between the terminal nodes for synchronisation.

time.time() in for loop - 3.32 us time.time() in while loop with i++ - 4.02 us = slower time.time() sequential - 2.81 us therefore for loop time - 0.51us

100Hz (excluding the 10ms) sleep(1/freq) - 104.13 us sleep(1/freq-timeDif) - 90.08 us == REMOVES DELAY OF 15us - GPIO AND A BIT SAME without sleep(1/freq) (just GPIO) - 5.18 us SAME without GPIO.output (just sleep(1/freq)) - 97.18 us == SLEEP IS THE INEFFICIENT FUNCTION LOSING ME TIME You can hard-code in the 104e-6 value to get offset of 5us instead of 104us but this is not ideal - need to compare to coding in c...

Table from interim report and two screen shots in particular.

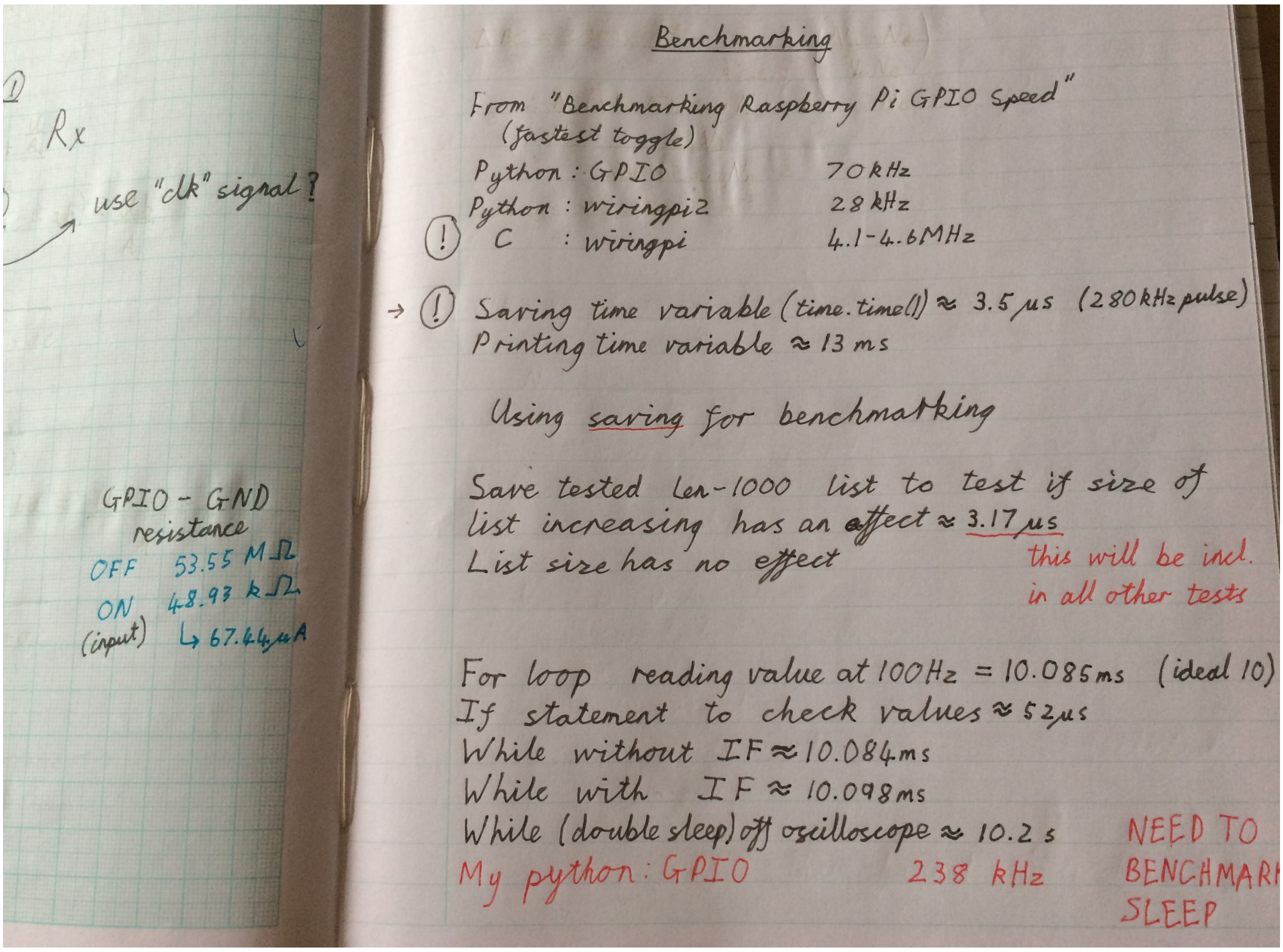


Figure 3.1: Benchmarking initial data collection

3.2.1 Comparing Python and C

Python is implemented as an interpreted language, meaning that the code (or at least an intermediate byte-code representation) is interpreted by a virtual machine at runtime. This means that, although a lot more powerful, Python can have drawbacks in performance relative to C, which is compiled into executable machine code prior to being run. An article from 2015 benchmarking the frequency achieved by switching a GPIO on and off continuously suggests that the Python library RPi.GPIO achieved the highest frequency of Python libraries, at 70 kHz [27]. It also suggested that all of the C libraries achieved frequency ranges in the Megahertz. The libraries have been updated to improve efficiency, but if C is found to be a significantly better option, then the pin manipulations of the transmitter and receiver themselves should be written in C and compiled into executables which the main Python code-base can run.

3.2.2 Maximum Frequency

Comparison of transmitter bank vs individual write and receiver read (callback) and read vars on clock pulse vs bank read on clock pulse – Also mention

```
pi@raspberrypi2:~/Documents/4YP_PiCom/4YP_PiCom_Transmitter/Part_Tests $ sudo ./
PiCom_WriteTimeTest
----- Single Pin vs Bank On-Off Write Test -----

EMPTY FOR LOOP
Time per for loop: 0.0060us
Frequency: 166.6667MHz

SINGLE TRANSITION
Time per for loop (on, off): 0.3460us
Individual Write Frequency: 5.7803MHz

BANK TRANSITION
Time per for loop (all on, all off): 0.1800us
Individual Write Frequency: 11.1111MHz
```

Figure 3.2: Write Time Test

```
pi@raspberrypi2:~/Documents/4YP_PiCom/4YP_PiCom_Transmitter/Part_Tests $ sudo ./
PiCom_ReadTimeTest
----- Single Pin vs Bank On-Off Read Test -----

SINGLE READ
Time per for loop (single read): 0.2280us
Read Frequency: 4.3860MHz

BANK READ
Time per for loop (bank read): 0.1320us
Read Frequency: 7.5758MHz
```

Figure 3.3: Read Time Test

3.3 Characterising Components of the Test Bed

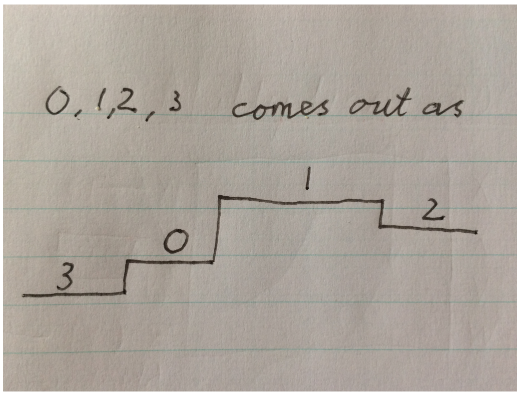


Figure 3.4: Non-continuous Output for Continuous Input Values of the DAC

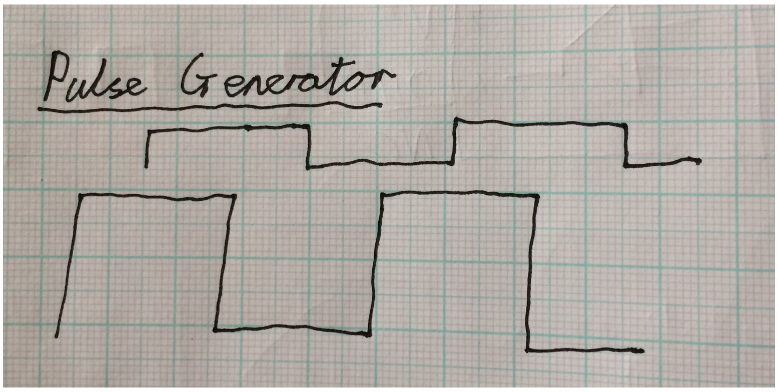


Figure 3.5: Output of pulse generator

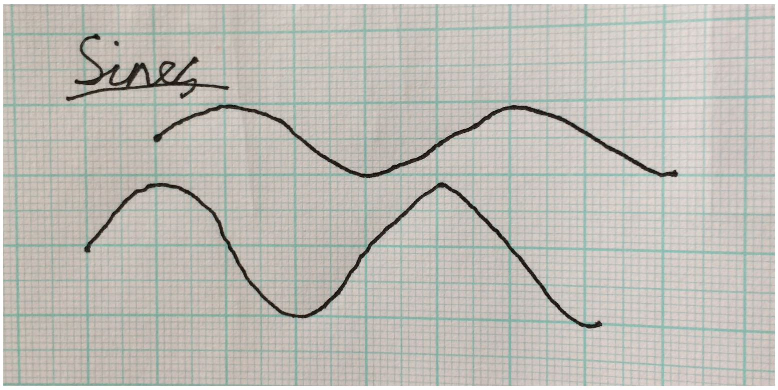


Figure 3.6: Output of pulse generator once filtered to be sine waves

Electrical Components:

- Analogue Digital Converter
- Digital Analogue Converter
- Quadrature Sinusoid Generator
- Multiplier/Mixer

• Low Pass Filters

3.3.1 Overclocking Components

The *pigpio* library has access to the hardware clocks of the Raspberry Pi. Specifically on the Pis used, it has the ability to set a hardware clock which is not reserved for system use to a specified frequency between 4.7 kHz and 250 MHz on pin 4, although the library documentation suggests that frequencies above 30 MHz are unlikely to work [28].

There are certain components such as the Analogue Digital Converter and the pulse generator which work using an internal clock set by an external resistor, but which may be overclocked by an external clocking signal, and this functionality may be used. This would allow the frequencies of these devices to be defined in software with less reliance on external physical components. The pulse generator isn't as good an example as it still uses external filtering to generate sinusoidal outputs, but this is particularly relevant for the ADC as it requires eight clock pulses per conversion, and setting a hardware clock for (at least) eight times the transmission frequency is one way of ensuring this in software.

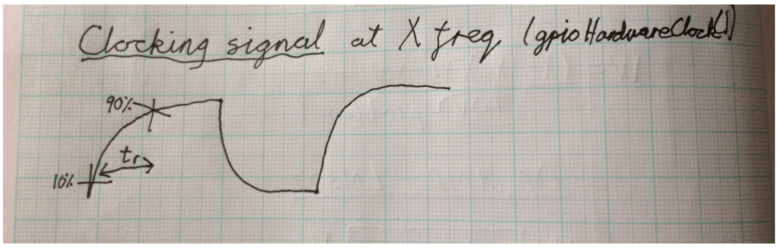


Figure 3.7: This is a high frequency view of the hardware clock for interest and rise-time (max frequency achievable actually) - will analyse screen grab from Oscilloscope and include in report

Chapter 4

Communications Testing

Neet to write up the Comms Testing – Mention the fact that we had to deal with deletion channel vs error channel, clocked vs unclocked, Justin's project - http://diwine-project.eu/public/publications.html

PAM vs QAM in terms of plus QAM has faster rate, grey coding, but DAC lookups are slightly off each other.

Testing Communications, remember project levels:

Easily Attainable: Construct a basic wired unidirectional communication test bed complete with a transmitter and a receiver. These units should be synchronised and an appropriate line code (i.e., baseband modulation scheme) should be exploited to convey test data from one device to another.

Medium Complexity: Characterise the performance of the test bed, identifying bandwidth limitations, noise characteristics, and reliability for different modulation and coding schemes. Test specific state-of-the-art modulation techniques recently published in the research literature. (These will be identified by the supervisor).

Advanced: Develop design enhancements that will enable the test bed to be extended to wireless scenarios, including RF and optical wireless systems. Implement these modifications if the budget permits.

4.1 SNR for Different Modulation Schemes

I'm not entirely clear on how you would calculate the SNR for these signals, considering we end up with a bit stream on either end and so bit error rate is easy to understand. Some insight as I'm writing up the test results would be handy – An SNR measurement assumes there is noise in the system. If you measure noise naturally, or otherwise artificially generate it, then the SNR would be E[wanted signal power]/E[noise signal power].

4.2 Error Rate

Bit error rate of transmission.

4.3 Channel Coding

Syndrome decoding - not a large amount of it covered but rather interesting stuff! CommPy - [29]

Chapter 5

Conclusion

Here we shall have our conclusion.

Write Conclusion

Summary, what has been achieved, recommendations for future work.

Make sure each figure is referenced explicitly in text and has a title, maks sure section references have the title where ambiguous otherwise, check for inline code consistency

Bibliography

[1] J. Mitola. “Software radios-survey, critical evaluation and future directions”. In: *[Proceedings] NTC-92: National Telesystems Conference*. IEEE. DOI: 10.1109/ntc.1992.267870.

[2] Mihir Patkar. *9 Things You Wanted to Know About Raspberry Pi*. [Online: Accessed ...] Aug. 12, 2016. URL: https://www.makeuseof.com/tag/9-things-wanted-know-raspberry-pi/.

[3] JFW Engineering Dept. *Understanding The Limitations Of Modern Military Radio Testbeds*. Tech. rep. JFW Industries, Inc. URL: https://www.jfwindustries.com/pdf/Understanding%20The%20Limitations%20of%20Modern%20Military%20Radio%20Testbeds.pdf.

[4] Justin P. Coon et al. *A wireless cloud network for the Internet of Things*. Tech. rep. DIWINE, Mar. 23, 2016. URL: http://diwine-project.eu/public/content/files/public/DIWINE_WhitePaper_4.pdf.

[5] Yijun Qiao, Harald Haas, and Edward Knightly. “A Software-defined Visible Light Communications System with WARP”. In: ().

[6] Abdelhamid Attaby and Moustafa Youssef. “WiPi: A Low-Cost Heterogeneous Wireless Testbed for Next Generation Applications”. In: IEEE-VTC Spring. Vol. 8. Wireless Networks: Protocols, Security, and Services Papers.25. 2018.

[7] Raspberry Pi Foundation. *Raspbian*. [Online: Accessed ...] 2018. URL: https://www.raspberrypi.org/downloads/raspbian/.

[8] James Mackenzie. *Headless Raspberry Pi Setup*. [Online: Accessed ...] Jan. 2, 2017. URL: https://hackernoon.com/raspberry-pi-headless-install-462ccabd75d0.

[9] Simon Tatham. *PuTTY: a free SSH and Telnet client*. [Online: Accessed ...] June 8, 2017. URL: https://www.chiark.greenend.org.uk/~sgtatham/putty/.

[10] RealVNC. *Download VNC Viewer*. [Online: Accessed ...] 2018. URL: https://www.realvnc.com/en/connect/download/viewer/.

[11] Eduard Bloch and Christian Lackas. *System Administration Utilities. VPNC*. [Online: Accessed ...] Nov. 23, 2016. URL: https://manpages.debian.org/testing/vpnc/vpnc.8.en.html.

[12] Jeff Forcier. *Welcome to Paramiko!* [Online: Accessed ...] 2018. URL: http://www.paramiko.org/.

[13] NumPy Developers. *NumPy*. [Online: Accessed ...] 2017. URL: http://www.numpy.org/.

[14] Almar Klein. *imageio. Python library for reading and writing image data*. [Online: Accessed ...] 2014. URL: https://imageio.github.io/.

[15] joan2937. *The pigpio library*. [Online: Accessed ...] Apr. 5, 2018. URL: http://abyz.me.uk/rpi/pigpio/index.html.

[16] Scott Chacon. *Git*. [Online: Accessed ...] URL: https://git-scm.com/.

[17] Python Software Foundation. *RPi.GPIO 0.6.3*. [Online: Accessed ...] Oct. 30, 2016. URL: https://pypi.org/project/RPi.GPIO/.

[18] Sebastian Dahlgren. *Using paramiko to send SSH commands*. [Online: Accessed ...] Oct. 22, 2012. URL: http://sebastiandahlgren.se/2012/10/11/using-paramiko-to-send-ssh-commands/.

[19] Embedded Linux Wiki. *RPi Low-Level Peripherals*. [Online: Accessed ...] Jan. 19, 2017. URL: https://elinux.org/RPi_Low-level_peripherals.

[20] Raspberry Pi Stack Exchange. *What are the Electrical Specifications of GPIO pins?* [Online: Accessed ...] Jan. 11, 2017. URL: https://raspberrypi.stackexchange.com/questions/60218/what-are-the-electrical-specifications-of-gpio-pins/60219#60219.

[21] Mosaic Industries. *GPIO Electrical Specifications*. [Online: Accessed ...] URL: http://www.mosaic-industries.com/embedded-systems/microcontroller-projects/raspberry-pi/gpio-pin-electrical-specifications.

[22] Gert Van Loo. *GPIO Pads Control. Errata*. [Online: Accessed ...] Aug. 2, 2012. URL: https://www.scribd.com/doc/101830961/GPIO-Pads-Control2.

[23] Arduino. *ARDUINO UNO REV3*. [Online: Accessed ...] 2017. URL: https://store.arduino.cc/arduino-uno-rev3.

[24] petzval. *Accurate timing for real time control*. [Online: Accessed ...] Aug. 10, 2013. URL: https://www.raspberrypi.org/forums/viewtopic.php?t=52393.

[25] digibird1. *Raspberry Pi as an Oscilloscope @ 10 MSPS*. [Online: Accessed ...] 2014. URL: https://digibird1.wordpress.com/raspberry-pi-as-an-oscilloscope-10-msps/.

[26] The DIWINE consortium. *Dense cooperative wireless cloud networks (DIWINE) – 5G experimental facilities: Smart meter network demonstrator testbed*. Tech. rep. DIWINE, Jan. 19, 2015. URL: http://diwine-project.eu/public/content/files/public/DIWINE_WhitePaper_2.pdf.

[27] Joonas Pihlajamaa. *Benchmarking Raspberry Pi GPIO Speed*. [Online: Accessed ...] Feb. 15, 2015. URL: https://digibird1.wordpress.com/raspberry-pi-as-an-oscilloscope-10-msps/.

[28] joan2937. *pigpio C Interface. gpioHardwareClock*. [Online: Accessed ...] Mar. 20, 2018. URL: http://abyz.me.uk/rpi/pigpio/cif.html#gpioHardwareClock.

[29] Veeresh Taranalli. *CommPy: Digital Communication with Python*. Version 0.3.0. [Online: Accessed ...] 2015. URL: https://github.com/veeresht/CommPy.

Format for web pages should be:

Creator's surname, creator's first name. Title. Date of publication. Name of Institution associated with site. View date. ¡http://address/filename¿.

Appendix A

Risk Assessments

A.1 General Risk Assessment

Department of Engineering Science **4YP Risk Assessment** UNIVERSITY OF OXFORD

Description of 4YP task or aspect being risk assessed here: *(Read the Guidance Notes before completing this form)* PiCom: A Digital Communications Test Bed Based on Raspberry Pi – Use of Raspberry pi and wireless breadboard		4YP Project Number: 11410
Site, Building & Room Number: Thom Building, Electronics Lab, 5th Floor	Approx size of equipment/apparatus used or built (in metres): Height: …0.03……. Width…0.08….. Length……0.15…….	Photo provided? YES/NO
Assessment undertaken by: Cameron Eadie	Signed: Eadie	Date: 02/11/17
Assessment Supervisor: Justin Coon	Signed: Justin P Coon	Date: 2.11.17

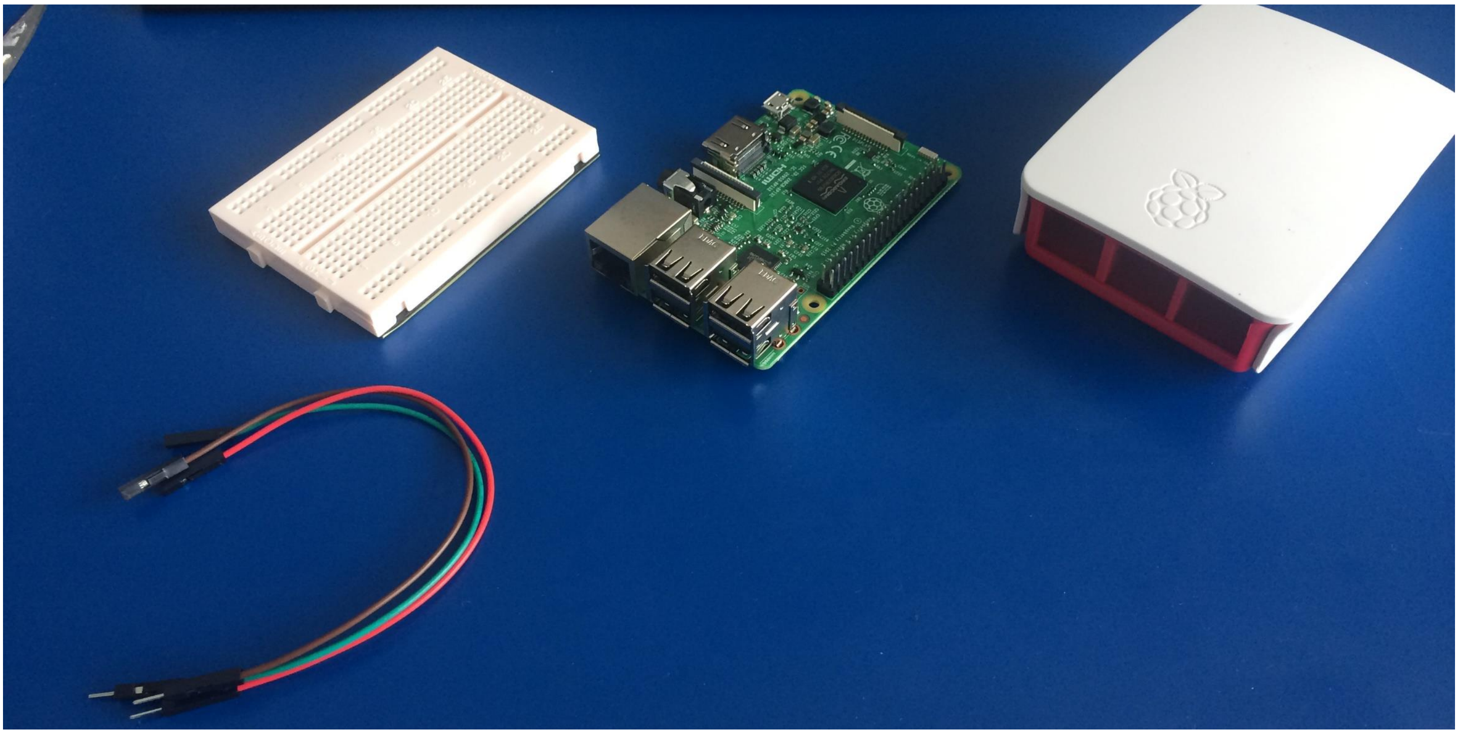
Assessing the Risk*
You can do this for each hazard as follows:

- Consequences: Decide how severe the outcome for each hazard would be if something went wrong (i.e. what are the Consequences?) Death would be "Severe", a minor cut to a finger could be regarded as "Insignificant".
- Likelihood: How likely are these Consequences to actually happen? Highly likely? Remotely likely, or somewhere in between?
- Risk Rating: Start at the left of the coloured Matrix. On your chosen Consequences row, read across until you are in the correct Likelihood column for the hazard in question. For example, an outcome with Severe consequences but with a Low probability of actually happening equates to a Medium risk overall. In this case "Medium" is what should be written in the Risk.

RISK MATRIX		LIKELIHOOD (or probability)			
		High	Medium	Low	Remote
CONSEQUENCES	Severe	High	High	Medium	Low
	Moderate	High	Medium	Medium/Low	Effectively Zero
	Insignificant	Medium/Low	Low	Low	Effectively Zero
	Negligible	Effectively Zero	Effectively Zero	Effectively Zero	Effectively Zero

Hazard *(potential for harm)*	**Persons at Risk**	**Risk Controls In Place** *(existing safety precautions)*	**Risk***	**Future Actions identified to Reduce Risks** *(but not in place yet)*
Electrical shock from 3.3V powered I/O pins or open circuit board on the Raspberry Pi	Student using the Raspberry Pi	• Use Raspberry Pi within its case whenever possible and avoid contact with I/O pins • Remember to call GPIO.cleanup() function in python code to turn off any active I/O pins used by a program. Call this in the 'finally' section of a try-except block so that it always executes before program exit • Do not connect I/O pins directly together – use resistors to prevent potential short circuit • Subject power suppy of Raspberry Pi to Portable Appliance Test (PAT) at regular intervals	**Low**	

Hazard *(potential for harm)*	**Persons at Risk**	**Risk Controls In Place** *(existing safety precautions)*	**Risk***	**Future Actions identified to Reduce Risks** *(but not in place yet)*
Electrical shock constructing and prototyping electronics on wireless breadboard	Student prototyping electronics to connect to Raspberry Pi	• Never build or rearrange electronics on wireless breadboard while Raspberry Pi is powered and I/O pins are connected to the board • Ensure that both Pi's are grounded together • Don't touch electronics while I/O pin connections are active	**Low**	



A.2 Computer Risk Assessment

Department of Engineering Science

Supplementary Questions for 4th Year Project Students

UNIVERSITY OF OXFORD

Risk Factor	Answer	Things to Consider	Record details here
Has the checklist covered all the problems that may arise from working with the VDU?	☑ Yes ☐ No		
Are you free from experiencing any fatigue, stress, discomfort or other symptoms which you attribute to working with the VDU or work environment?	☑ Yes ☐ No	Any aches, pains or sensory loss (tingling or pins and needles) in your neck, back shoulders or upper limbs. Do you experience restricted joint movement, impaired finger movements, grip or other disability, temporary or permanently	
Do you take adequate breaks when working at the VDU?	☑ Yes ☐ No	Periods of two minutes looking away from the screen taken every 20 minutes and longer periods every 2 hours Natural breaks for taking a drink and moving around the office answering the phone etc.	
How many hours per day do you spend working with this computer?	☐ 1-2 ☑ 3-4 ☐ 5-7 ☐ 8 or more		
How many days per week do you spend working with this computer?	☐ 1-2 ☑ 3-5 ☐ 6-7		
Please describe your computer usage pattern	Use of laptop for a number of hours a day, most days, in deparment or at home		