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

Software-defined radio test beds are valuable resources for verifying theoretical work on digital communications, however they are often too expensive to justify putting together. This paper investigates the design of a software-designed raido (SDR) test bed using two Raspberry Pis and a number of commercially available off-the-shelf chips. The test bed is designed to be significantly lower cost than current SDR platforms, which use expensive custom software radio boards as the radio interface. The result is a system with a retail price of $£165$, which constitutes a significant reduction in price in comparison to any of the other test beds discussed. The construction and operation of the test bed are investigated for the understanding of the reader. This architecture is analysed and the issues dealt with such that it may be extended or improved. The test bed is then tested for baseband transmission of On-Off Keying, 4-level Pulse Amplitude Modulation and 16-Quadrature Amplitude Modulation, however a problem encountered with the digital analogue converter used made it unreliable for Orthodonal Frequency Division Multiplexing. The limits of the test bed are analysed, and although it performs reasonably well given the low budget, the prototyped design prevents the test bed from working well at high frequencies where exposed wires emit interfering radiation to adjacent wires causing impairments. Suggestions to mitigate these problems and improve on the architecture are made.

Contents

Todo list

Chapter 1

Introduction

The idea of a software radio is attributed to Dr. Joseph Mitola III in the early 1990s, referring to radios which could be reconfigured by changing the software, allowing for changes to communications protocols without altering the underlying hardware [1]. The ideal software radio would consist of a computer connected to a Digital-Analogue Converter (DAC) connected to an antenna as a transmitter, and an antenna connected to an Analogue-Digital Converter (ADC) connected 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 and 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 or boards to construct them, however most of these can be very expensive. Although these test beds supply advanced tool kits, the basic functionality is all that is required to successfully test new communication schemes. The goal of this project is to develop a low-cost software-defined radio test bed using a Raspberry Pi to encode and transmit data, and another to receive and decode it. This uses the Raspberry Pis themselves, as well as a

small number of external components to provide functionality that the Pis alone cannot. Once this test bed is constructed and analysed, the next step is to characterise the performance of the test bed.

1.1 Motivation

The Raspberry Pi is a low-cost computer board which runs on 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 and inexpensive way of developing an alternative SDR test bed (see Section 1.2). This would make such test beds significantly more available, such that one might be built by anyone wanting to physically test 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 to attempt to characterise the performance of the test bed. This entails measuring the Raspberry Pi and other components used, as well as describing test bed in terms of its success at achieving the communications goals. This includes testing its reliability for different modulation and coding schemes, and identifying key characteristics such as noise and bandwidth limitations.

1.2 Literature Review

There are a number of software radios and SDR test beds which have been developed in recent years. A large number of them make use of one or more of the 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 $\pounds$880 to $\pounds$3,510. The reconfigurable SDR peripherals for rapid prototyping are even more expensive, with prices starting around $\pounds$5,000 [4]. Some

free, open source software which provides a number of communications libraries and tools, and is used by a number of projects, is GNU Radio [5]. While this is an excellent option for a number of test beds, it does require a lot of processing power and memory, which earlier models of the Raspberry Pi were not able to handle, but the Model 3+ which is used is capable of running it. However, this would significantly impact the ability of the Raspberry Pi to transmit or receive signals at the same time. This is unlike USRP-based test beds where the Peripheral handles the load of transmitting/receiving and the computer separately manages the data conversion load of GNU Radio. There have also been problems with compiling GNU Radio on Raspberry Pis, but this can be solved [6]. Because of the drawbacks, and the significant code base which would take time to master, the decision to develop independent software was made. GNU Radio could possibly be included in an alternative iteration of this project's test bed.

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 [7]. 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. In 2014, Ku et al. developed the first SDR test bed for the testing of a RF sub-sampling receiver using a USRP and GNU Radio [8]. This receiver sampled a $4\,\mathrm{GHz}$ signal at $100\,\mathrm{MS/s}$ with almost zero bit error and shows that radio systems are starting to approach the capabilities of the ideal software defined radio.

There are also projects which use a large number of 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. Its final White Paper describes the use of six Ettus USRPs in a testing setup, with two source, two relay and two destination nodes used to simulate a Smart Meter Network [9]. Another similar USRP-GNU Radio project described a distributed software defined radio test bed for localisation and tracking of an emitter in real time [10]. This used a complex combination of GPS synchronisation and Kalman filtering to track the target node (USRP and computer) based on readings from a number of distributed USRP-based sensors. An alternative to the Ettus USRP is the WARP (Wireless Open-Access Research Platform) Board developed at Rice University. It includes two programmable RF interfaces and a number of peripherals. A demonstration using WARP Boards for a software-defined visible light communications system was presented by Qiao et al. [11]. This system was used to demonstrate optical Orthogonal Frequency Division Multi-

plexing (OFDM) using a WARP Board connected to an LED as the transmitter and another connected to a photo-detector as the receiver. 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, so financial limitations may impede many researchers' ability to access SDR test beds. As a result, particularly in the last two years, a number of groups have looked to the Raspberry Pi as a low-cost alternative to develop these test beds. The first examples of this are not designed using the Raspberry Pi as the main component of the test bed, but as a low cost "cognitive engine" to interface with USRPs. El Barrak et al. implemented this idea for an SDR-based spectrum sensing prototype, using a Raspberry Pi 3 for the computation running GNU Radio [12]. This was then connected to a USRP which performed the spectrum sensing operation. This prototype was used to develop the idea of more effectively utilising the frequency spectrum available with "cognitive radio", sensing which frequencies are in use and dynamically using spectrum portions opportunistically as they become available. Another project - WiPi - which directly confronts the high cost of wireless test beds, approached this problem by designing a costly test bed and making it remotely accessible [13]. As the paper states, "Implementation on real devices is the optimal approach to capture the conditions in real scenarios." and this is why there is such a demand for the ability to use these test beds. This remote test bed is achieved with a number of Raspberry Pis each connected to a USRP, and these all connected to a central server which researchers can access, using the test bed to validate their results. Pasolini et al. proposed "A Raspberry Pi-Based Platform for Signal Processing Education" [14]. This platform inverts the concept of the previous Raspberry Pi test beds, and does the computation on a standard computer, using the Raspberry Pi as the transmitter. The communications part of the platform is computed using a free Simulink package provided by MathWorks, "Simulink defined radio" so there is no need to have programming experience to use it [15]. The output is generated through a USB audio card with an audio bandwidth of $48.000\,\mathrm{SPS}$ (samples per second), allowing for signals "within the interval *[0, 24]kHz*." This demonstrates the capabilities of the Raspberry Pi not as a computer but instead as a radio transceiver.

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 the components of its 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 to discover its limits.

The project combines the two aspects of extant Pi-based SDR test beds and utilises the full power of the Raspberry Pi as both the centre of computation and as the communications interface through the GPIO pins. It also provides a suitable architecture for the external hardware of the test bed using low-cost commercially available off-the-shelf components. This architecture is analysed and the concerns dealt with, such that it may be extended or improved by anyone wanting to investigate further possibilities. The retail price of the test bed (excluding tax) is $\pounds$165, which is very affordable compared to any of the test beds discussed. The performance of the test bed is moderate given the low budget, and a number of reasons for its downfalls are highlighted such that they may be addressed at a later stage. Finally, future directions of this test bed are discussed in the conclusion identifying where improvements could be made to get the most out of future implementations.

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 [16].

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), where the rest of pins are not assigned numbers as they are power or ground pins.

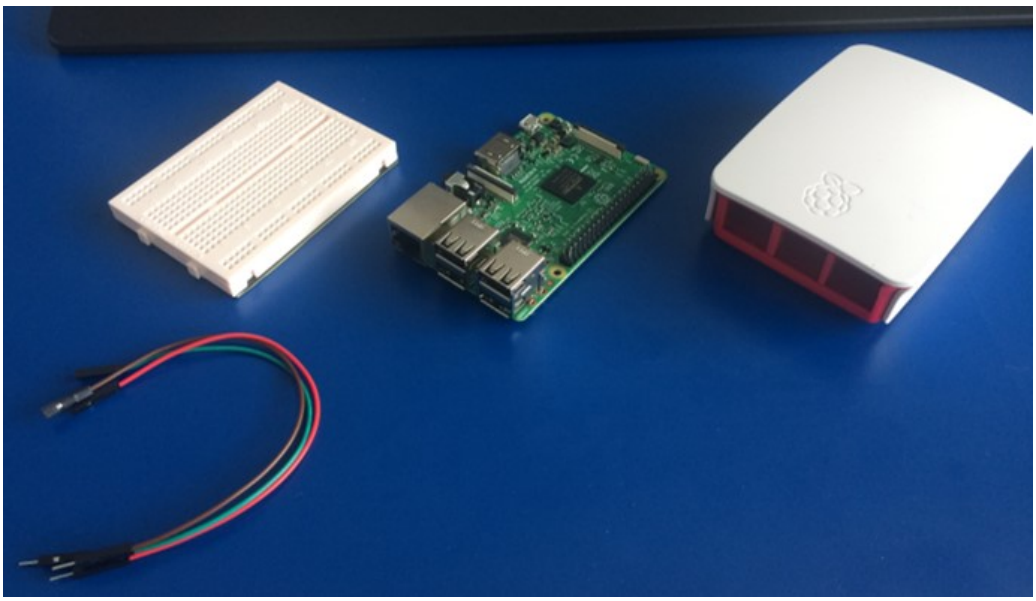


Figure 2.1: The Raspberry Pi with its case, a prototyping breadboard and some jumper cables

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 [17]. 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. Alternative remote acess is possible using Secure Shell (SSH) set up for the command line, or Virtual Network Computing (VNC) for the X Window interface. Secure Shell is necessary in this case, as it is used in the test bed execution. When actually using the test bed, both Raspberry Pis are run 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 is connected to a screen and keyboard, and then it starts the receiver via SSH, which runs headless (without a screen or control peripherals) and receives the data, processes and outputs it, and stores 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 for Windows can be used [18], with *pi@raspberrypi.local* as the host name so the IP address is not needed. The standard login is *username: pi* and *password: raspberry*.

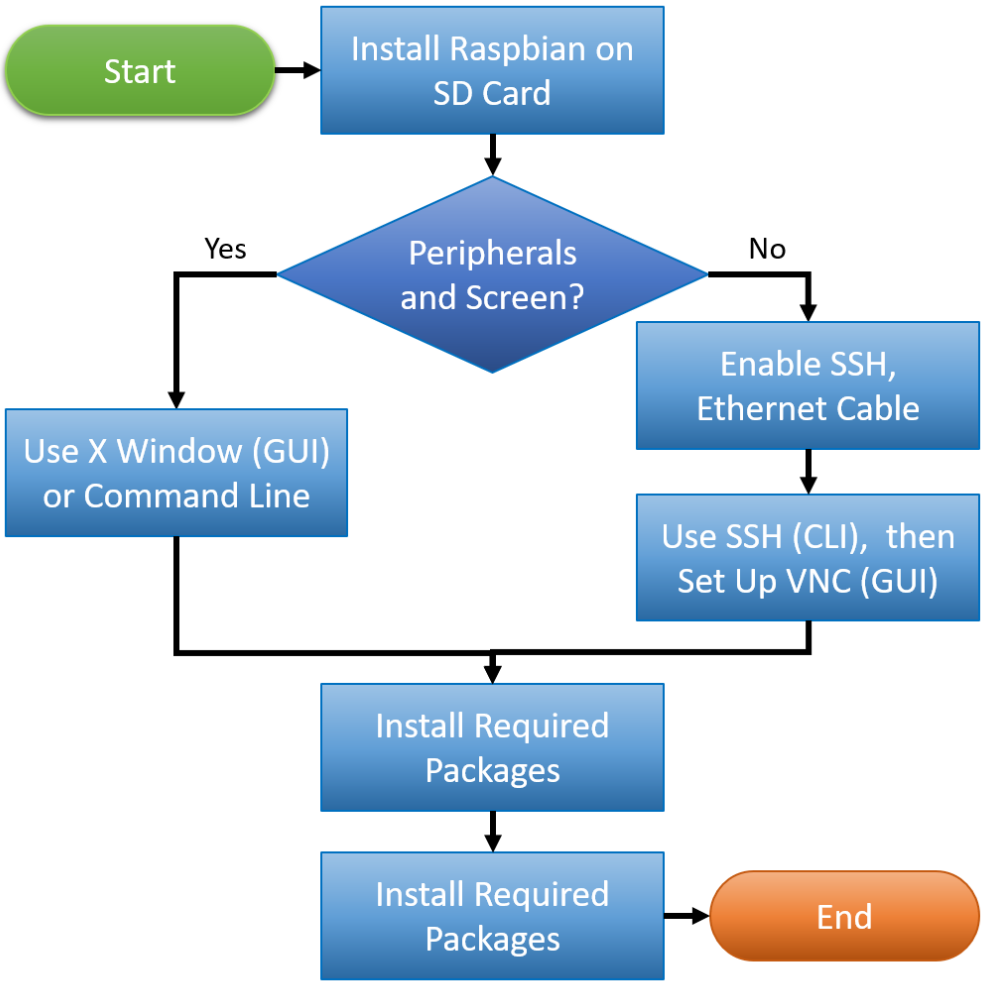


Figure 2.2: The setup procedure of the Raspberry Pi

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 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 [19]. 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 Python 3, `pip` - the Python package manager - is run as `pip3` for Python 3. Any dependencies not in this list give clear instructions in the terminal 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 pip3 install matplotlib
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 [20]
- *Paramiko* - Python library used for SSH to start the receiver [21]
- *NumPy* - Python library used for easier array manipulation as well as interface with images [22]
- *imageio* - Python library used to read and save image files as *NumPy* arrays [23]
- *matplotlib* - Python library used to plot graphs of data using a MATLAB-like interface [24]
- *pigpio* - C library used to interface with the GPIO pins [25]
- *Git* - Version control software which also allows for the cloning of the project code to any device [26]

2.2 Test Bed Architecture

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, eventually all integrated into the final setup. The first is the Pis connected together by three wires, a serial data line, a clock line and a connection between ground pins as shown in Figure Figure 2.3. This arrangement is similar to that used for the Inter-Integrated-Circuit (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 (at baseband, this can also be modulated using the next setup). The final arrangement extends the set-up to two DACs and two ADCs. These signals are then modulated by a sine wave and a cosine wave respectively and added, and can be separated at the receiver due to the orthogonality of the two signals. This arrangement is used for Quadrature Amplitude Modulation as well as Orthogonal Frequency Division Multiplexing.

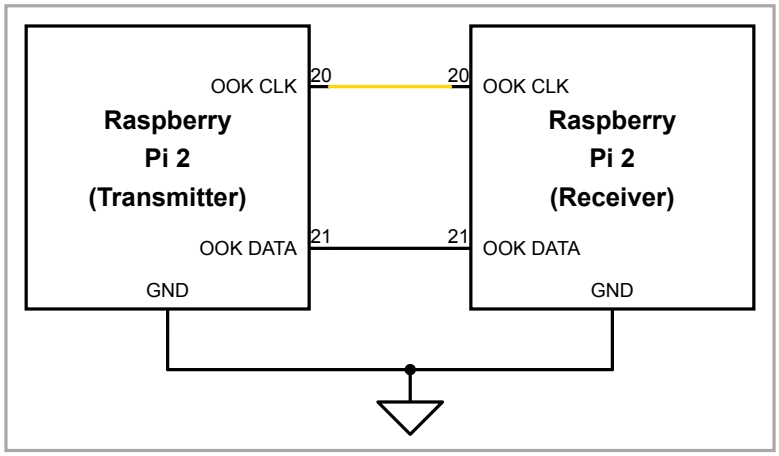


Figure 2.3: Test Bed Set-up For On-Off Keying

2.2.1 Digital Analogue Converter

The Digital Analogue Converter (DAC) 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. A parallel 8-bit DAC and equivalent Analogue Digital Converter (ADC) were chosen because they can achieve significantly higher data rates than a serial interface device which must

transmit all eight bits per sample from one GPIO pin. 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 could produce the 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. However, the latches are not transparent (the input is not visible while $\overline{CS}$ is low and the chip is selected), and so the output holds the analogue conversion of the latched digital byte only until the falling edge of $\overline{CS}$. 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. The same clock pulses are also fed to the $\overline{WR}$ (start conversion) pin of the ADC. Connections between the transmitter Raspberry Pi and the DAC are in Figure 2.4.

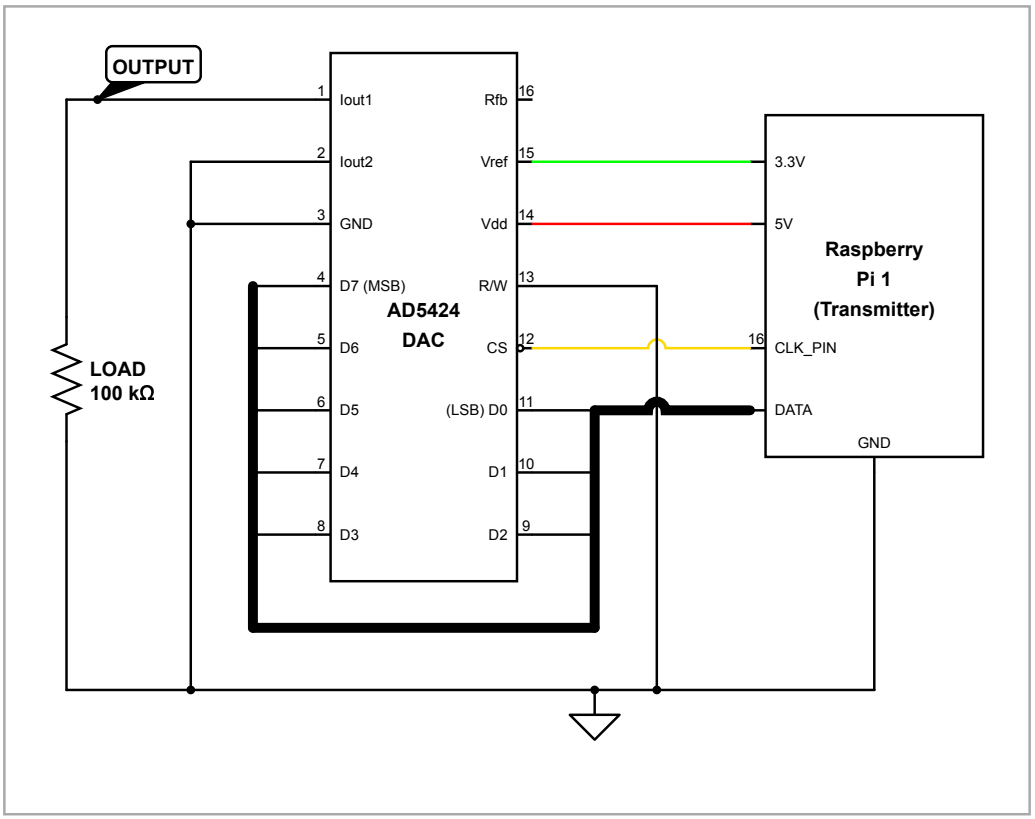


Figure 2.4: Layout of the Digital Analogue Converter

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 overdriven, 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 $\overline{WR}$ is fed from the clock line of 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. This is because there is some delay in the ADC's calculation of its digital value, but on the rising edge of $\overline{BUSY}$ the resulting digital value at the output is correct and stays correct until the next negative pulse on $\overline{WR}$. This works 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 permanent non-destructive readout of the digital output.

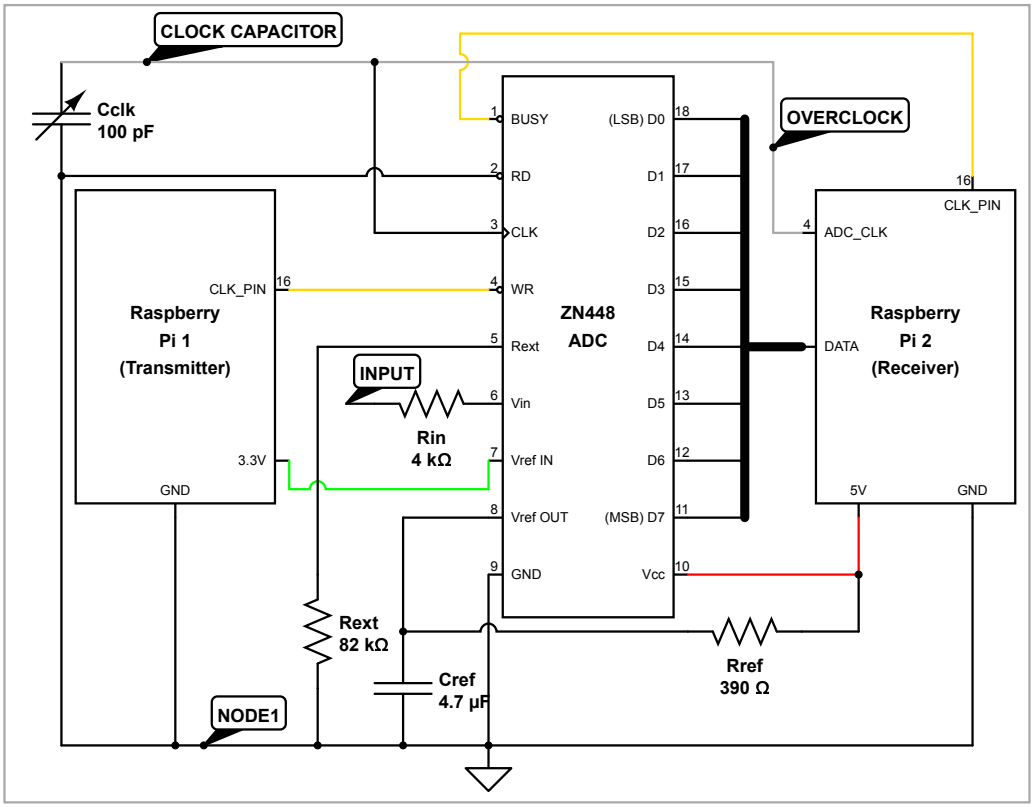


Figure 2.5: Layout of the Analogue Digital Converter

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 3.3.4). 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. The data sheet suggests that 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 the guaranteed 9 µs conversion time (0.11 MSPS); a clock frequency giving 9 pulses between conversions is suggested to absolutely ensure complete conversion. It is worth mentioning that although the maximum suggested capacitor-based or overdriving frequency of the internal clock is 1 MHz, the data sheet states that all tests were done with an internal frequency of 1.6 MHz suggesting this can be pushed. 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.

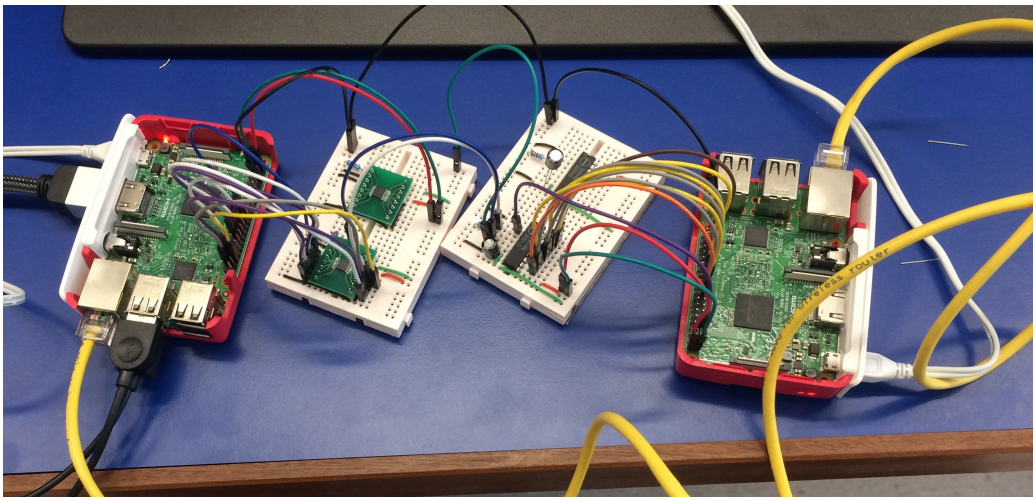


Figure 2.6: Layout of Test Bed for Pulse Amplitude Modulation

2.2.3 Quadrature Pulse Generator

The pulse generator is the LTC-9608-2 which provides 90^o shifted (quadrature) output square waves. The frequency is a function of a single setting resistor, with a value of $f_{out} = 10MHz \times \frac{10000}{R_{set}}$ and can operate up to 10 MHz. In order to produce sinusoidal output, a second order passive RC low-pass filter is used at each of the outputs. One output is multiplied by the baseband signal for PAM modulation, and both are multiplied by the I and Q components of the output respectively and

added together for QAM in order to quadrature-carrier modulate the signal.

2.2.4 Multiplier

The multiplier is the AD633, capable of four-quadrant multiplication and requiring no external components. It also has a summing input, allowing the output of one multiplier to be added to the output of another. 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 $V_{max/2}$ provided with a voltage divider. This means that the sine and cosine signals would be inverted when multiplied by the values equating to 0 and 1 in the same way they would be for -3 and -1. As a result the transmitted signal will be a regular PAM or QAM signal plus a DC component of $V_{max/2}$. The final layout of the test bed is shown in Figure 2.7.

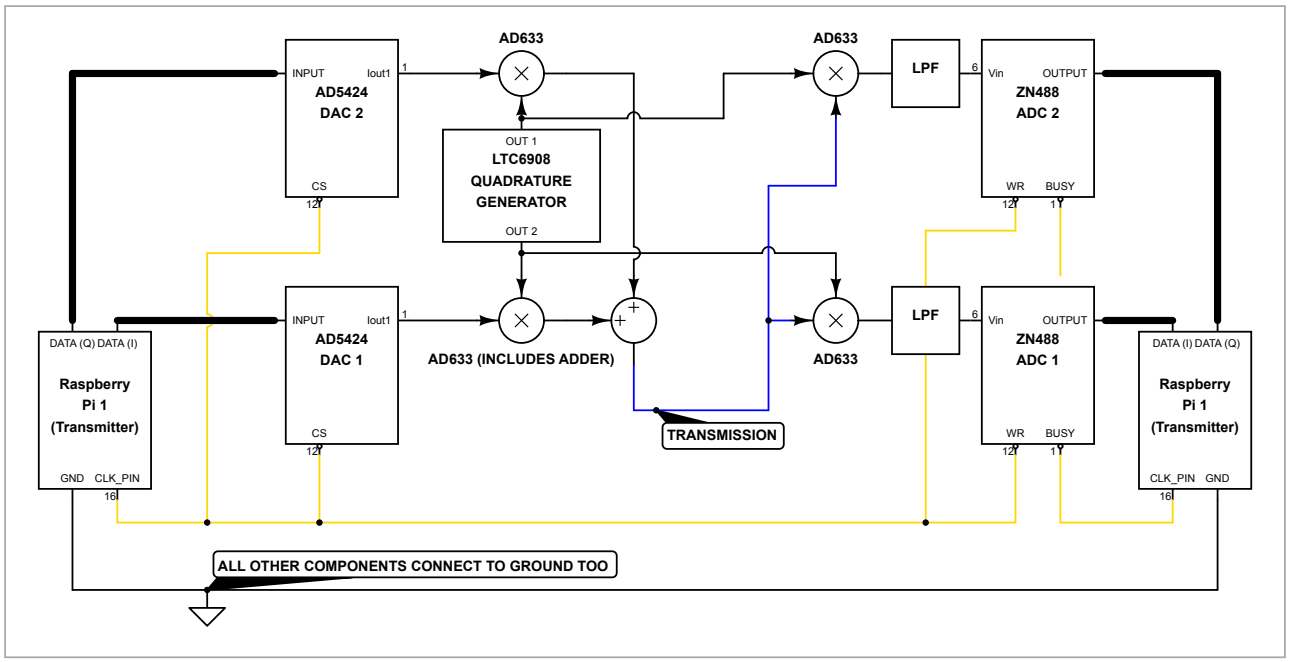


Figure 2.7: Layout of the Test Bed for QAM and OFDM

2.2.5 Parts Used

Figure 2.8 shows various chips and their on-board set-ups. 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 retail price - without VAT (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

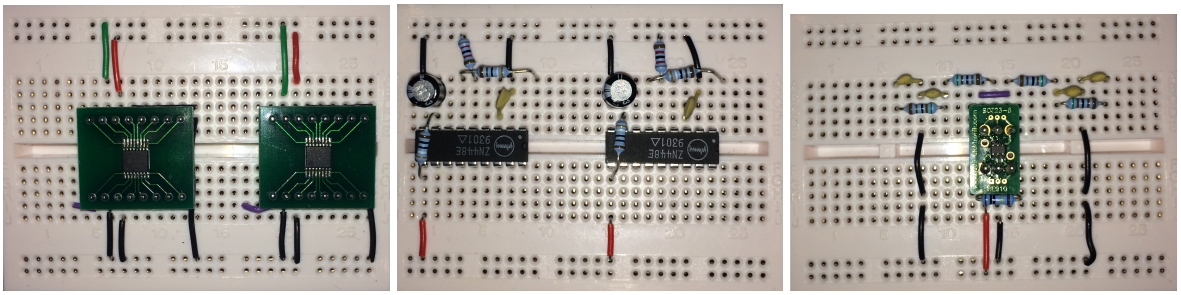


Figure 2.8: The DAC (left), DAC (centre) and Quadrature Pulse Generator (right) set-ups on individual breadboards

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 lower cost compared to purchasing or building a standard test bed architecture. Although these more standard test beds may provide certain advanced capabilities, the key functionalities required should be achievable with this design. The extent 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 (retail prices as available from RS, Digi-Key, Farnell and Mouser Electronics)

It is worth noting that there is a large number of available options for each component of the test bed. Each possibility has certain advantages and disadvantages, and many 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 the code.

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

- 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 converter had a problem with outputting correct value output (see Section 3.3.1) which impacts the effectiveness of all advanced modulation schemes, and would need to be rectified before OFDM could be implemented.
- The Digital Analogue Converter was also chosen for its easy interfacing with microcontrollers, but it has a current-output which may explain the non-full-range voltages. A chip intended for use as a voltage-output device with the same characteristics would be preferable.
- 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 similar. 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 specific chip could be found for this purpose, and chips that did anything in this area were expensive and overly complicated.
- 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 the design are again made using fixed-value components. If frequency response of a filter could be altered in software, all frequencies used could be hardware independent, although this DC low pass filter probably would not have a problem filtering out a carrier.

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 [27]. This is used for the On-Off Keying part of the communications test bed. However, Python is relatively slow, 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 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 of 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 out any changes to the hardware as the pins used for data transmission and clocking on OOK are different to those used by the advanced modulation schemes. This adheres better to the Software Defined Radio paradigm, allowing any transmission to be run on one hardware setup of the test bed (see Section 1.2).

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.

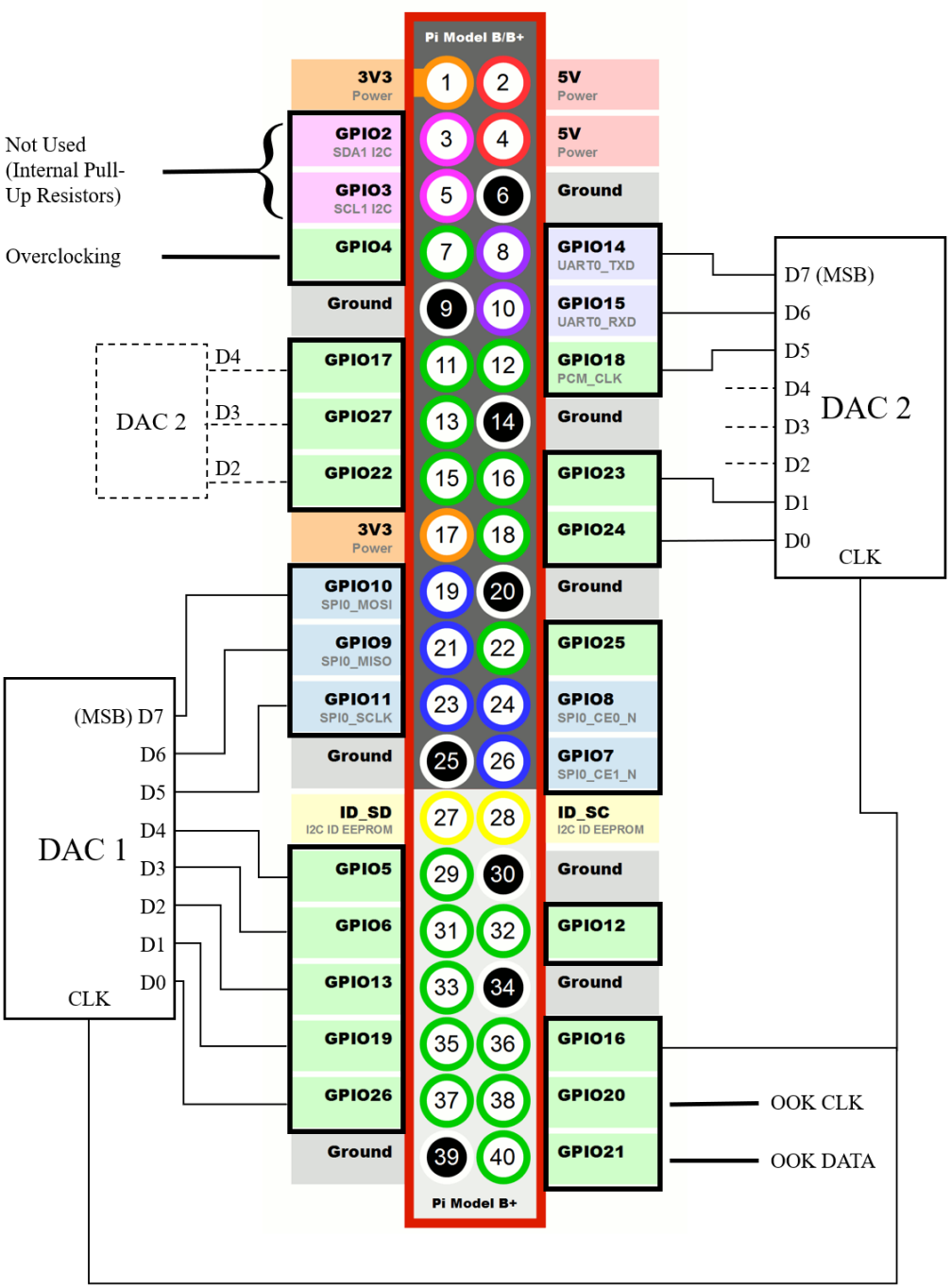


Figure 2.9: Pin diagram of connections to the Raspberry Pi (based on image from [28])

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 numbers (GPIO# in the boxes) rather than their BOARD numbers (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 are not 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 overdriving external components which can be fed a clock signal, which is discussed in Section 3.3.4. 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. Listing 2.2 shows these pin definitions except for the DAC clock pin (pin 16) which is defined in the C code.

```
1  # OOK Pins
2  CLK_PIN = 20
3  DATA_PIN = 21
4  # DAC Pins
5  DAC_PINS_1, DAC_PINS_2 = [10, 9 , 11, 5 , 6 , 13, 19, 26], \
6                           [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 (OOK) 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 Python as compared to C is addressed in Section 3.2.1. 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 an interrupt-driven solution is not ideal, but it was more easily written and so used for this early-stage modulation scheme.

Earlier versions of On-Off Keying included a function `Prep_Binary_Data()` which added initial and final padding of '1's to the data 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 as `Encode_Error_Correction()` was to be implemented, and 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 unfortunately was not implemented due to time constraints and unforeseen problems with other components, however in Section 4.3 the integration of error correction into the test bed is discussed.

2.3.1.1 Starting the Receiver

The receiver is started with the library *Paramiko* which is used to make an SSH connection, and then to execute a command on the device which it connects to [21]. The article "Using paramiko to send SSH commands" by Sebastian Dahlgren was particularly useful in understanding how the library is properly implemented [29]. 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 a program is run from a command in command line mode, it can have additional information passed 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 neither Raspberry Pi is expending these 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 statistics which can be generated quickly after transmission, 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 which will be transmitted, while it is converted from byte-wise data into bitmasks. The bitmasks are then transmitted by the DACs, and the ADCs save the data similarly at the other 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 as `import numpy as np` so all of the library functions can be called with the standard `np.` shorthand, for example `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 performing 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 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), adding them, and 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 expressed in the frequency domain. It then transmits the Inverse Fast Fourier Transform (IFFT) of the combination of these frequency-domain sub-carriers as a complex carrier-modulated signal.

The communications test bed could be implemented with OFDM, however there were certain component malfunctions and time constraints which meant that this was not realised. The details of how the test bed would be extended are included in this section. As mentioned in Section 2.2.4, 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 26) 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, that there were spikes at certain values, and that a non-continuous analogue response was measured for continuous digital input. This problem is investigated in Section 3.3.1 but the solution chosen 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 found.

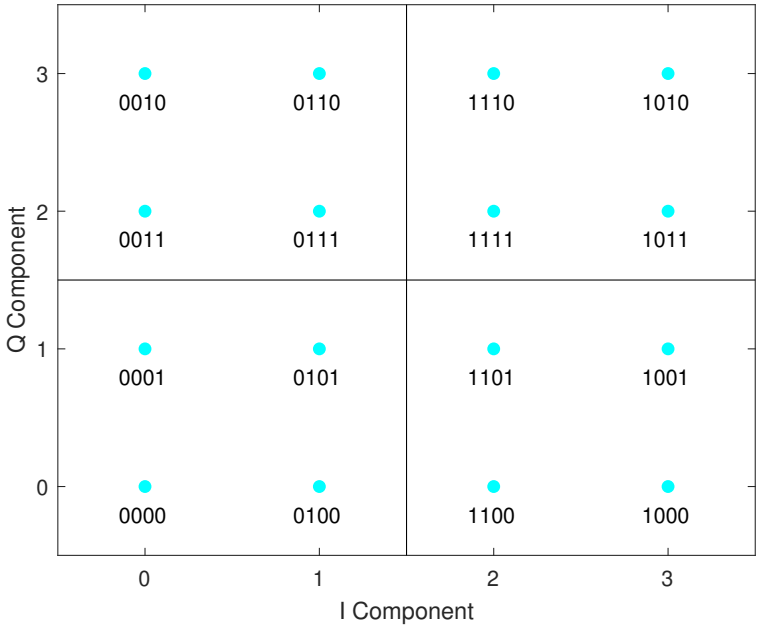


Figure 2.10: 16-QAM Constellation with Grey Code

The transmitter can be run from the command line with transmission frequency and modulation scheme as command line variables, otherwise it uses the hard coded values. The data is taken in 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 implemented 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 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 into two symbols. This is done by unpacking each byte into its corresponding bits and reshaping them 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 two 8-bit DAC values, 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 it 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 that 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 time-domain 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 cosine 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 - 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 names suggest, one function is required to set all of the pins which will be set to '1', and one is required to clear all of the pins which will be set to '0'. 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.

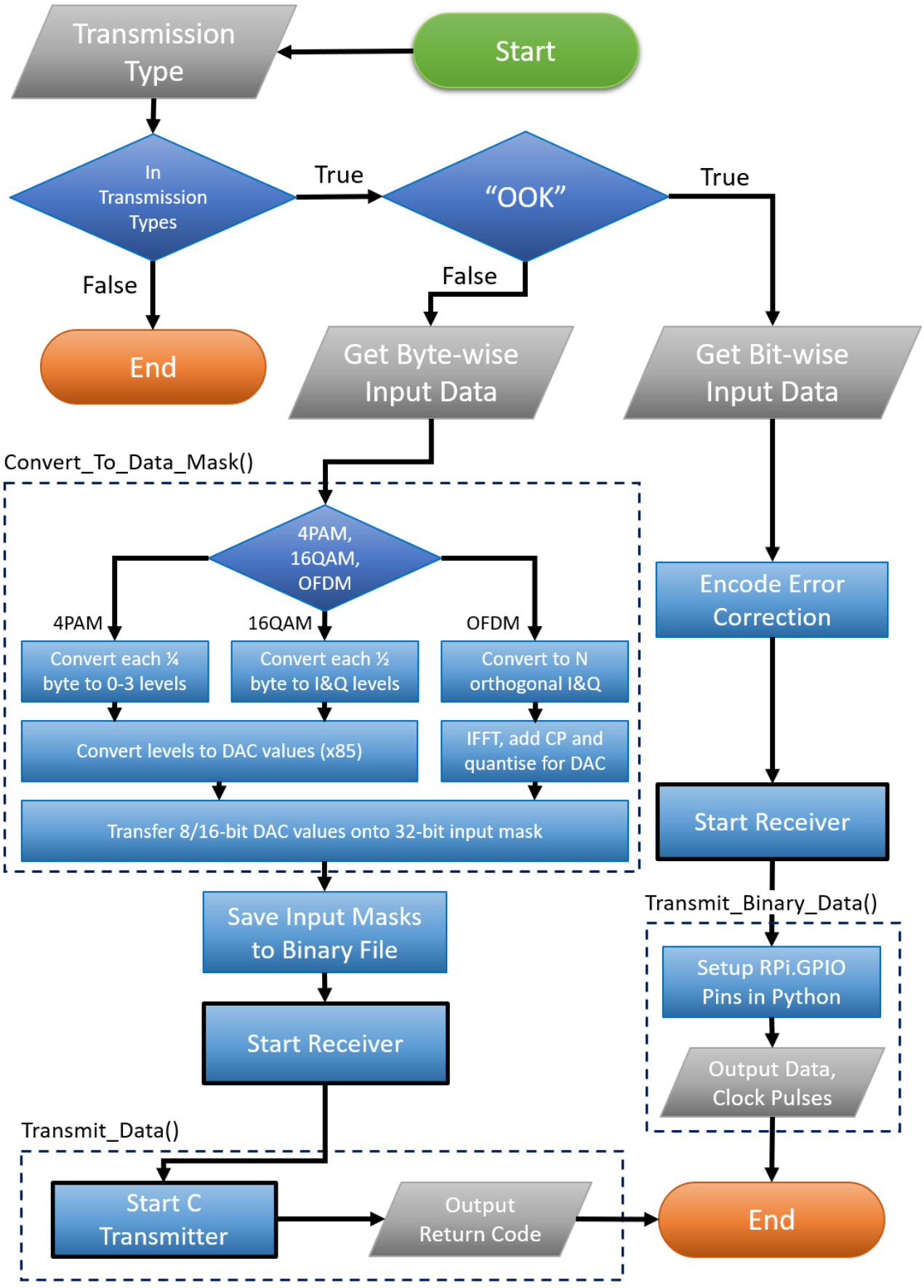


Figure 2.11: Flow chart for transmitter code

Figure 2.13 (see Page 28) 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. The transmission is also monitored for time out by setting a watchdog on the clock pin (waiting for a no-action time out), 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.

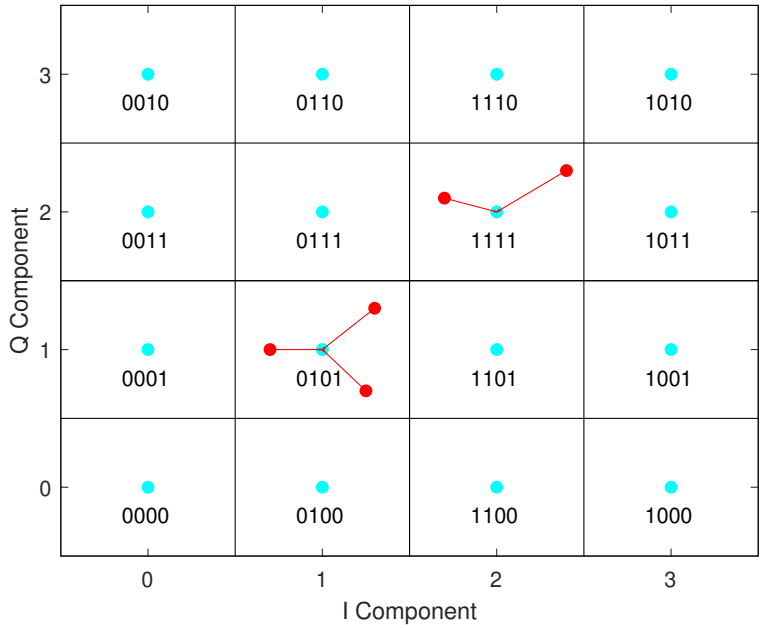


Figure 2.12: 16-QAM constellation with Maximum Likelihood boundaries

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 halfway 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.

Once the input has been decoded into the ML estimated symbols, it must be recombined into

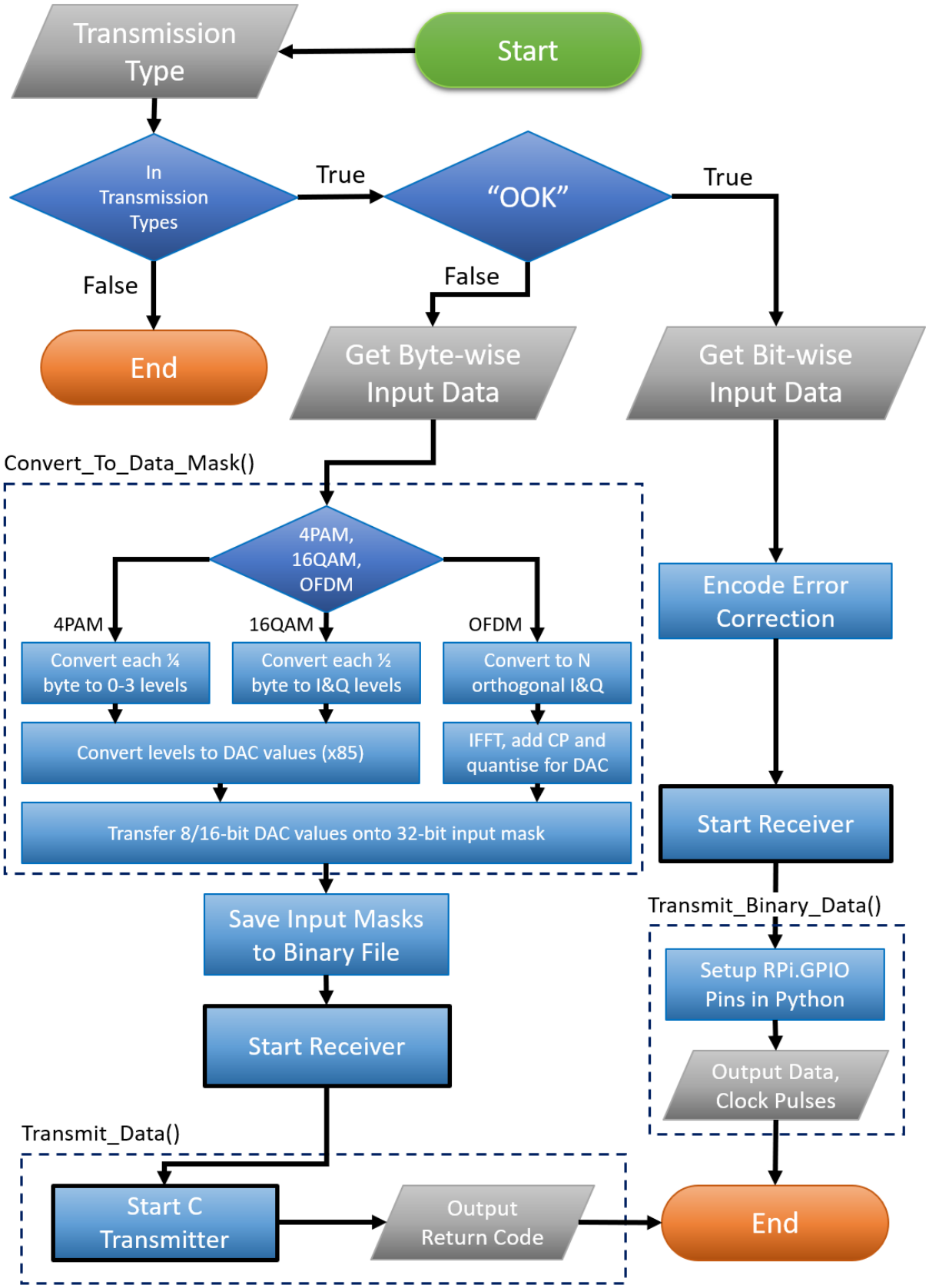


Figure 2.13: Flow Chart for Receiver Code

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.

2.3.2.1 C Transmitter and Receiver

The C library used for manipulation of the GPIO pins is *pigpio* [25]. 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 of their previous values. Section 3.2 compares 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. 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.

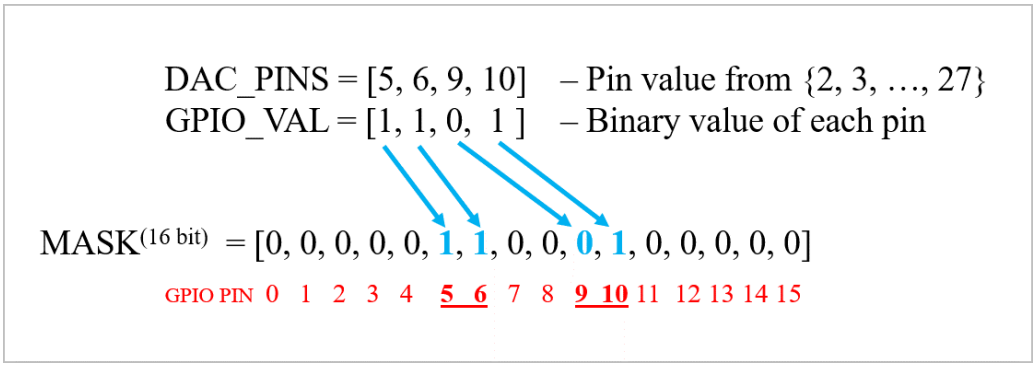


Figure 2.14: Explanation of the concept of mask expansion

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 the inversion masks (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 low again at the end of the clock cycle with the next parallel data. 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 upon 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 (100ns at least)
* It is also short enough that at max frequency (100kHz) it is still a fraction of
* a clock pulse */
```

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 the callback function `readPins()` on the change of state of the clock (ADC $\overline{BUSY}$) pin. If the event triggered was a rising edge, it saves the value of `gpioRead_Bits_0_31()` into the output array. A static counter keeps track of the data received, and when it reaches the number of masks expected, closes the callback and changes a global variable `pin_state` to a specific value to signal ending the program. The callbacks are also closed and this specific value set if there is a watchdog time out – when the callback is called because there has been no change in the clock pin for a certain amount of time.

Once the receiver has set the callbacks on startup, it uses a while loop to wait until the specific exit value is set on `pin_state` to exit the loop and end the program. Finally, the received bitmasks are saved to a binary file which is opened by the Python code and mapped to ADC pins to decode the received data.

Chapter 3

Electronic Testing

The assessment of the electronic capabilities of the test bed is split into three main sections. The first section deals with the electircal characteristics of the Raspberry Pi and pertains to the GPIO pins and their physical capabilities. The second section deals with the computational characteristics of the Raspberry Pi, and particularly how fast the pin values can be changed and read by the underlying code, and how most efficiently to achieve this. The third section deals with the components used in the test bed, discussing limitations to its operation 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 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." [30] 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 [31]. There is nothing lim-

iting the current-output on the pins, so they will attempt to drive whatever current is drawn until they stop 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 $16\,\mathrm{mA}$ 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 [32], and it also suggests that one should not attempt to source or sink more than $16\,\mathrm{mA}$ on an output pin. The pins actually have a set drive strength from $2\,\mathrm{mA} - 16\,\mathrm{mA}$ in $2\,\mathrm{mA}$ increments which is set for a bank of pins and usually set to $8\,\mathrm{mA}$ but even a pin set to $2\,\mathrm{mA}$ drive strength and then loaded so as to draw $16\,\mathrm{mA}$ will not be damaged [33].

The maximum current which can be drawn from an output pin is a useful detail. This is used to ensure that none of the attached components draw too much current. It is also used to decide, along with the input impedances of the pins, whether or not the GPIO pins of one Raspberry 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 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 ($\mathrm{k\Omega}$)
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 $48.93\,\mathrm{k\Omega}$ will only draw $67.44\,\mu\mathrm{A}$ of current from a $3.3\,\mathrm{V}$ 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 $32.9\,\Omega$, and so if an output at $0\,\mathrm{V}$ were connected to another pin at $3.3\,\mathrm{V}$ it would potentially draw about $100\,\mathrm{mA}$ which would damage both pins or at least (the Raspberry Pi has safeguards excessive

currents) restart the Pi. However, this is unlikely to happen in the test bed setup unless an output pin is directly connected to a 3.3 V power pin, or both Pis have 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

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 be 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 [34]. 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 [35]. This should not be done for a long period of time as it is bad for the Pi, and in this case the transmitter and receiver may be used to transmit for a significant amount of time. This means that disabling all interrupts is not a good idea without periodically turning on interrupts to allow them to run, and this would result in 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, and should be for any processor-intensive work [36]. This is achieved with the terminal command in Listing 3.1.

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

Listing 3.1: Turning off the RAM refresh rate adjustment

The choice to keep the common clock and carrier was to simplify the rest of the setup, after a number of attempts at synchronisation were made. This becomes particularly important when dealing with carrier modulated signals as a phase-locked loop would be required to recover the phase of the carrier. This decision is also made based on precedent – the DIWINE project in their second White Paper, two years into a three year project, still describe a "'time zero' reference" between the terminal

nodes of their test bed for synchronisation [37].

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 has drawbacks in performance relative to C, which is compiled into executable machine code prior to being run. An article by Joonas Pihlajamaa from 2015 benchmarked the frequencies which could be achieved by switching a GPIO on and off continuously. It suggests that the Python library RPi.GPIO achieved the highest frequency of the Python libraries, at 70 kHz [38]. It also suggests that all of the C libraries tested achieved frequency ranges in the Megahertz, although it did not benchmark *pigpio* which this test bed uses. The libraries have been updated to improve efficiency since 2015, 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.

```
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 3.1: Results of PiCom_Write_Time_Test

The test in the benchmarking article was performed with Python *RPi.GPIO* and C *pigpio*; an endless while loop writing a '1' then a '0' was started on a single pin, and the output frequency measured on an oscilloscope. The result was conclusive: Python produced a clock frequency of about 238 kHz, whereas the C code produced a clock frequency of about 2.87 MHz. Having decided that the pin manipulations were to be done using *pigpio* in C, the next step was to decide whether to use the bank read and write functions or the single-pin alternatives. This may seem an obvious choice, but depending on the speed of the read functions in particular, it may be more efficient to read each pin individually into a global variable holding their values as they are updated than to read all of them at

once.

The results comparing the single pin write and bank (setting eight values corresponding to the DAC mask) write functions are in Figure 3.1. This test was done using the timer function `gpioTick()` to average over the time of 1000 iterations of a for loop. An empty for loop is included for comparison. Importantly, the single pin write loop consists of setting the pin on and then off, whereas a bank write consists of a bank clear and set, so the equivalent loop is clear 0 and set the DAC mask, followed by clear the DAC mask and then set 0. Even though this is the case, the bank write is still approximately twice as fast as a single pin write, probably due to lower level direct memory access. The bank write can also set all 8 or 16 DAC pins at once and so is the clear favourite.

```
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 3.2: Results of PiCom_Read_Time_Test

The results comparing the single pin read and bank read functions are in Figure 3.2. This time there is only one function call in each loop, and again the bank function is about twice as fast. This suggests that reading each pin individually is a waste of time, as even if this was required, the bank function could read all of the pins in less time. As a result, all C code uses the bank functions except for the second (rising edge) clock transition in the transmitter which needs to happen after the DAC write and falling clock edge happen simultaneously in the bank write.

3.3 Characterising Components of the Test Bed

In order to use components in the test bed, it must be ensured that they are working correctly, such that any problems can be solved or mitigated. As there are a large number of components available, it is possible that some may be unsuitable for the intended use or may not work exactly as expected, therefore it is important to ensure that all components are working correctly and performing appropriately prior to including them in the test bed.

3.3.1 Digital Analogue Converter

The Digital Analogue Converter uses a resistor ladder connected to V_{ref} to determine the output. Switches determined by the digital values latched at the input of the device connect the corresponding resistor to either I_{out1} (the output) or I_{out2} (which is connected to ground). The ladder is designed such that each digital bit corresponds to a voltage twice the value of the bit before it is switched to the output, therefore giving an output proportional to the digital input value. In order to confirm that this is indeed the case, the transmitter is run with 256PAM modulation with an input of monotonically increasing byte values 0 to 255 repeating. This should ideally produce a monotonic linear output. Figure 3.3 shows the output of the DAC, with the output on probe 1 (yellow) and the data value of 3.3 V on probe 2 (green).

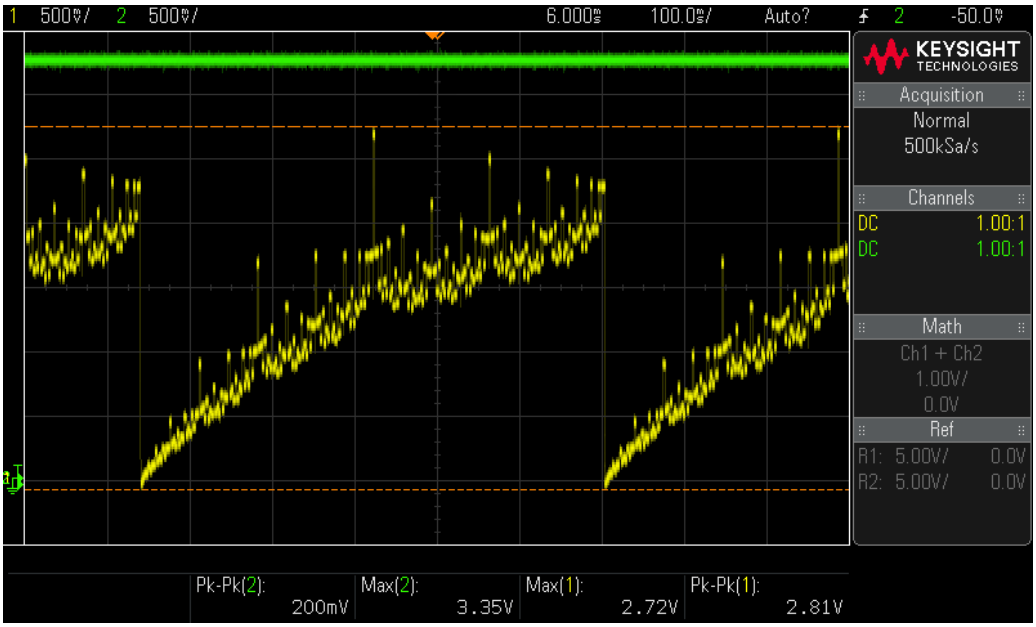


Figure 3.3: DAC non-linearly increasing output voltage for linearly increasing digital input values

The output is clearly not as expected which represents an unfortunate shortcoming if not rectified. The values seem to jump around the general trend expected, with peaks particularly at values of 64, 128 and 192. Decoupling the output from the power supply was attempted but this did nothing to affect the output, and lab technicians were consulted to no avail. The program was rerun with a pause point in the code between each output value to confirm that this behaviour was not to do with the output frequency, to verify that the bitmasks were giving the correct digital outputs (which they were), and to check that the peak points were the aforementioned values, corresponding to binary inputs of $0b01000000$, $0b10000000$ and $0b11000000$ respectively (where $0b$ signifies the binary representation). The behaviour is also consistent for both of the DACs purchased.

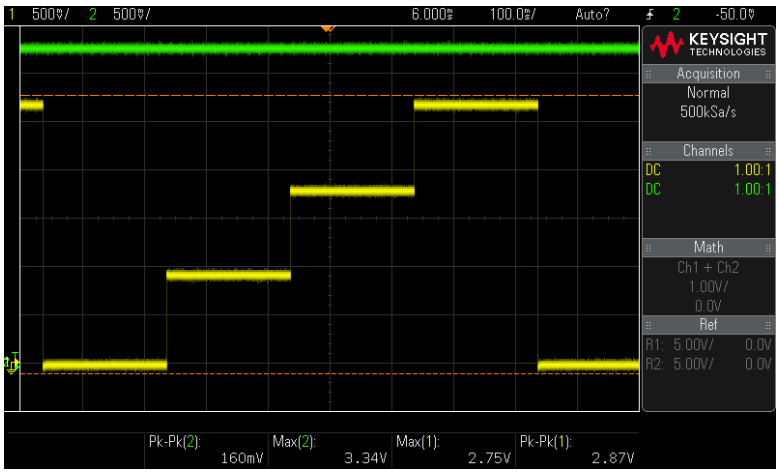


Figure 3.4: 4PAM output with lookup table

The solution chosen for this problem was to use a lookup table for empirical digital values which as closely as possible correspond to equidistant output values. This lookup table is used in the code to translate each PAM symbol or the real and imaginary parts of each QAM symbol from the range $[0, 3]$ to particular digital values, rather than as previously simply multiplying by 85 to produce values from 0 to 255. Figure 3.4 shows the output of 4PAM with such a table. Values are close, but there are some slight discrepancies, and this lookup takes advantage of the 128 maximum peak for larger output range but this could be dangerous if either DAC stopped exhibiting this peak for some reason.

3.3.2 Analogue Digital Converter

For the Analogue Digital Converter, the conversion is not instantaneous; it contains a DAC, a series of dividers and a comparator, and is constantly comparing the most significant bit (MSB) to $\frac{V_{ref}}{2}$ while $\overline{WR}$ is low. When the trigger goes high the conversion starts, and runs for eight clock pulses of the ADC's internal clock. On completion of this conversion the $\overline{BUSY}$ pin goes high, which is taken as the read clock for the receiver Raspberry Pi. The sampled output of a 5Hz wave sampled 5000 times at $10\,\mathrm{KSPS}$ is shown in Figure 3.5 with the expected value shown in red underneath the data, plotted with matplotlib [24]. The output has periods where it stops following the input signal and settles at the middle value instead but other than that follows the values fairly accurately.

It was originally thought that the ADC is on occasion jumping to the added DC value of the zero-centred sine wave, but it is also possible that it is occasionally missing timing and sampling during the time when the $\overline{WR}$ pin is low and the output is set to $0b10000000$ to compare the most significant bit. This data was collected when the write clock signal from the transmitter (on the $\overline{WR}$ pin of the

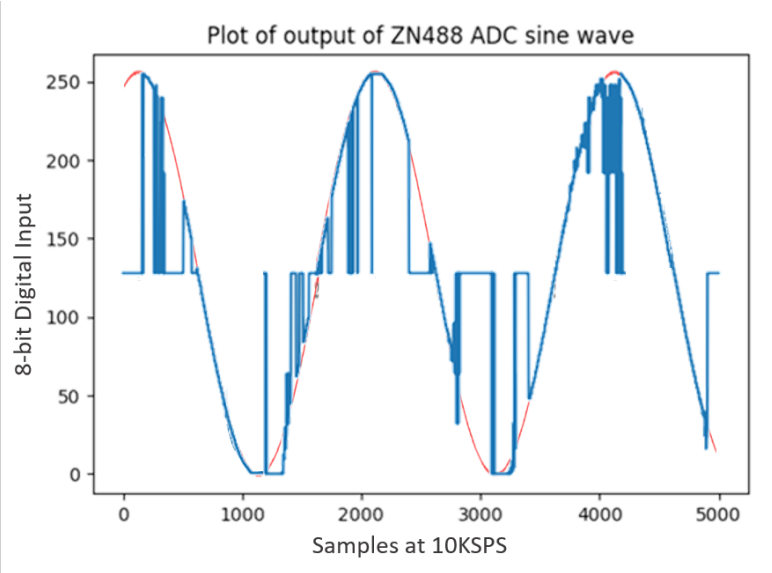


Figure 3.5: Input of ADC for a 5Hz sine wave generated with a signal generator

ADC) was positive for 50% of the clock cycle and 0 for 50%, but it was realised that a more logical signal for the operation of both converters is a small falling edge then rising edge pulse to latch the data in the DAC and to start the conversion of the ADC. If the problem was the DC value it will not affect the test bed, and if it was the delayed timing then the adjusted clock signal should make this far less likely. Unfortunately it was not possible to collect new data to verify this due to time constraints with the signal generator.

3.3.3 Quadrature Sinusoid Generator

The chip used to generate the sine waves is a quadrature pulse generator, outputting two square waves 90^o out of phase. Frequency output can be in the range $5\,\text{kHz} - 1100\,\text{kHz}$. Figure 3.6 shows the output of the pulse generator for resistor values of $1\,\text{M}\Omega$ ($100\,\text{kHz}$) and $100\,\text{k}\Omega$ ($1\,\text{MHz}$) respectively.

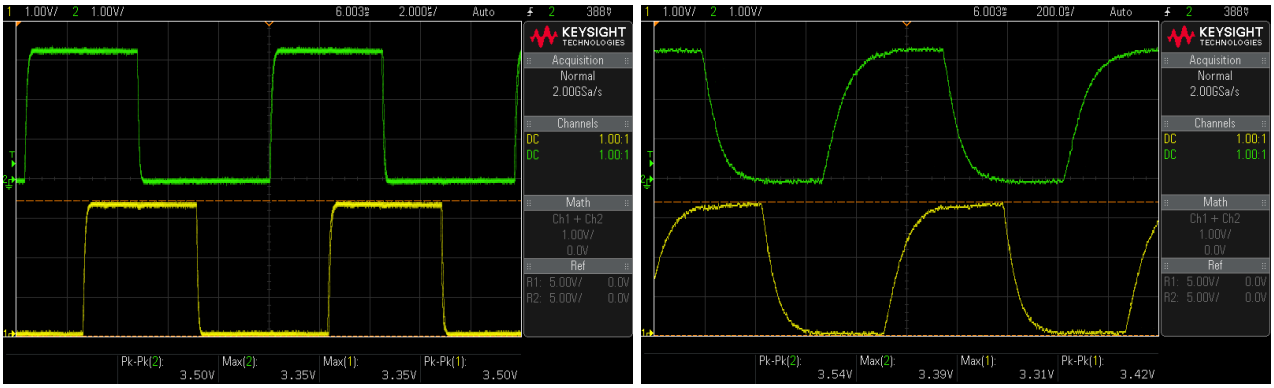


Figure 3.6: Output of quadrature pulse generator at 100kHz (left) and 1MHz (right)

A single carrier frequency needs to be selected in order to design the low pass filters used to convert the square wave outputs into phase-shifted sine waves. A frequency of $100\,\text{kHz}$ was chosen

as the wave form was a lot more stable at this frequency, but this is a design choice. Figure 3.7 shows the same outputs filtered with a second order passive RC filter.

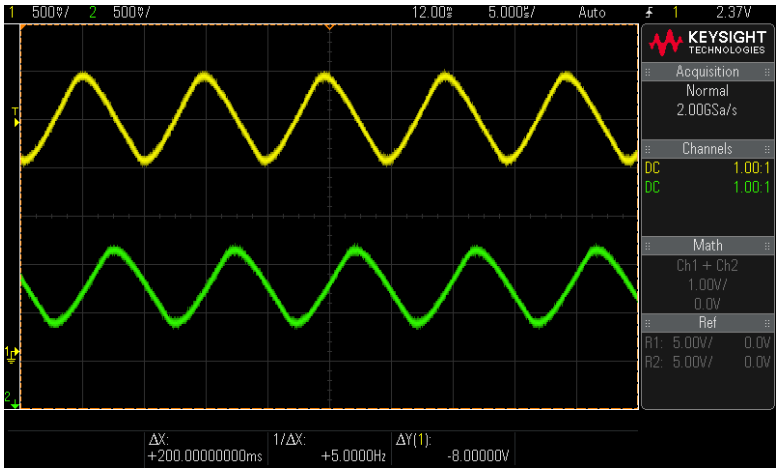


Figure 3.7: Output of pulse generator once filtered to be sine waves

3.3.4 Overdriving Component Clocks

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 [39].

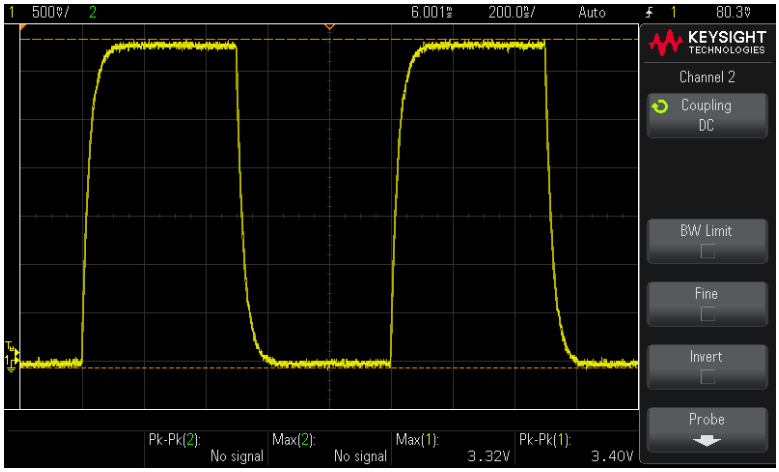


Figure 3.8: A 1MHz clock signal produced by the hardware clock

There are certain components which work using an internal clock set by an external resistor or capacitor, but which may be overdriven 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. This is particularly relevant for the Analogue Digital Converter 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 3.8 shows the stability of this hardware clock at 1 MHz which is the maximum clocking frequency of the ADC.

Chapter 4

Communications Testing

In order to determine the noise characteristics and error rate of each communication scheme on the test bed, data is transmitted through the system and analysed with respect to the input. For On-Off Keying, a random bit stream is generated once and then used for all of the tests so it can be compared consistently. Not many of these tests are included, because above about 500 Hz Python starts to miss clock edges with the speed of the transitions, and it becomes a deletion channel instead of an error channel. For the advanced modulation schemes, 4-Pulse Amplitude Modulation and 16-Quadrature Amplitude Modulation are tested at baseband frequencies. The data used for these tests is a black and white image of a kitten with enough gradient and contrast to be interesting informationally, because random bit sequences are less interesting to transmit and do not provide an intuitive understanding of the noise introduced or whether the data has transmitted correctly. The image is read into the transmitter, converted into bitmasks for the DAC, sent to the receiver, read in as bitmasks by the ADC and converted back into an image. The transmissions are referred to in terms of baud rate or symbol rate. This is also equivalent to the bitmask transmission rate, and is half the bit rate for 4PAM (2 bits per symbol/bitmask) and a quarter of the bit rate for 16QAM (4 bits per symbol/bitmask).

Figure 4.1: The original cat image (left) and the image transmitted at a baud rate of 500Hz (right)

Figure 4.1 shows the original cat image and the image transmitted at a low baud rate (500 Hz). Transmission was attempted for a number of slightly different lookup tables for the DAC symbols all at one frequency (including no lookup table - this produced the worst results), and this seemed to make a significant difference to the quality of the transmission – this is a large source of error in the system. The darkest areas of the image suffer the most from the transmission, and this could be due to the scaling of values being inclusive of noise, but it could also be due to more likely errors introduced by the components. There was a concern that at high frequencies the transmission would start to act as a deletion channel as it did with the OOK transmission, and miss some bitmasks. This was not a problem, however, as even at the highest frequency the ADC could handle (100 kHz), the number of bitmasks received was the number expected. Figure 4.2 shows the result of 4PAM transmission at different increasing baud rates from 1 kHz – 100 kHz. The 1000 Hz image shows addition of salt and pepper noise relative to the 500 Hz image, and then Gaussian noise seems to slowly fade out the image leaving only the basic structure intact. Interference is likely introduced by the connective wires from the GPIO to the components, which are capacitively coupled at high frequency and would explain the averaging out of the colours as frequency increases. The final 100 kHz image looks like it is almost entirely random noise.

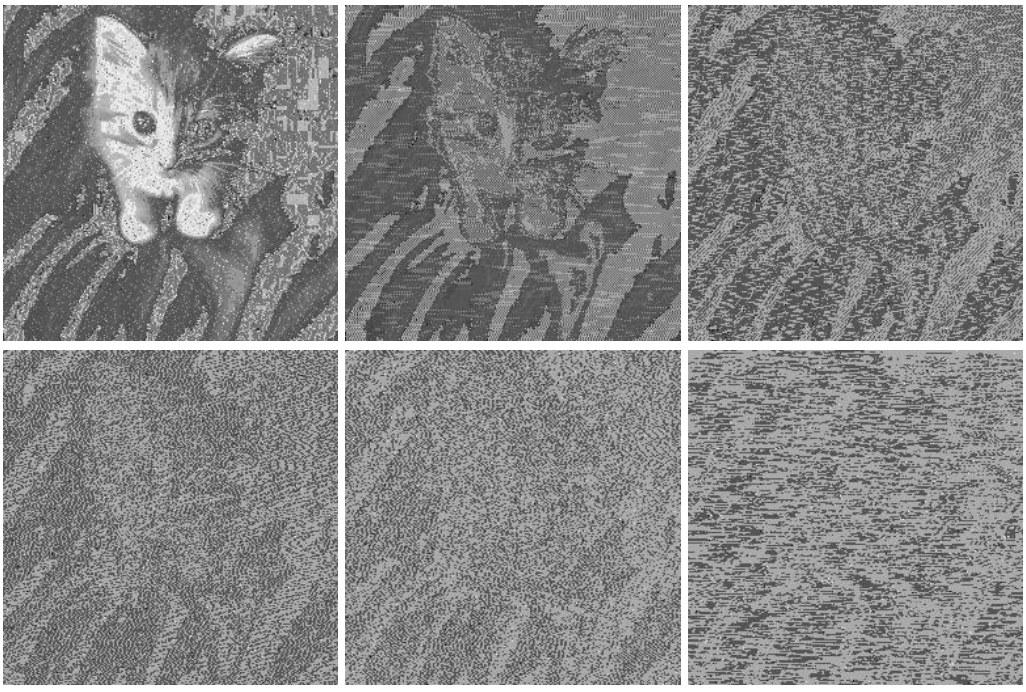


Figure 4.2: Rates of transmission for 4PAM: (top) 1kHz, 2KHz, 5kHz, (bottom) 10kHz, 50kHz, 100kHz

The images for the same baud rates as Figure 4.2 are shown in Figure 4.3 for 16 QAM transmission. These images, although different in quality, show roughly the same structural characteristics, particularly the difference between the 2 kHz image and the 5 kHz image in each, and the transition to random noise in the 100 kHz image in each. The symbol rates for corresponding 4PAM and 16QAM

being the same means that the bit rate for 16QAM is twice as high, although as the I and Q components of each symbol are transmitted in parallel the symbol rate is more relevant for comparison, seen in the similarities in same-baud-rate images for both sets. The 16QAM images appear lower in quality than the 4PAM images. There are two expected reasons for this: firstly, there are more exposed cables which can become capacitively coupled in this setup, compounding the problem, and secondly, although both DACs displayed very similar output values, the lookup table used is based on the first DAC and may not be as accurate in expressing the correct levels for the second. Grey coding implemented in this transmission scheme should have helped to reduce the bit error rate.

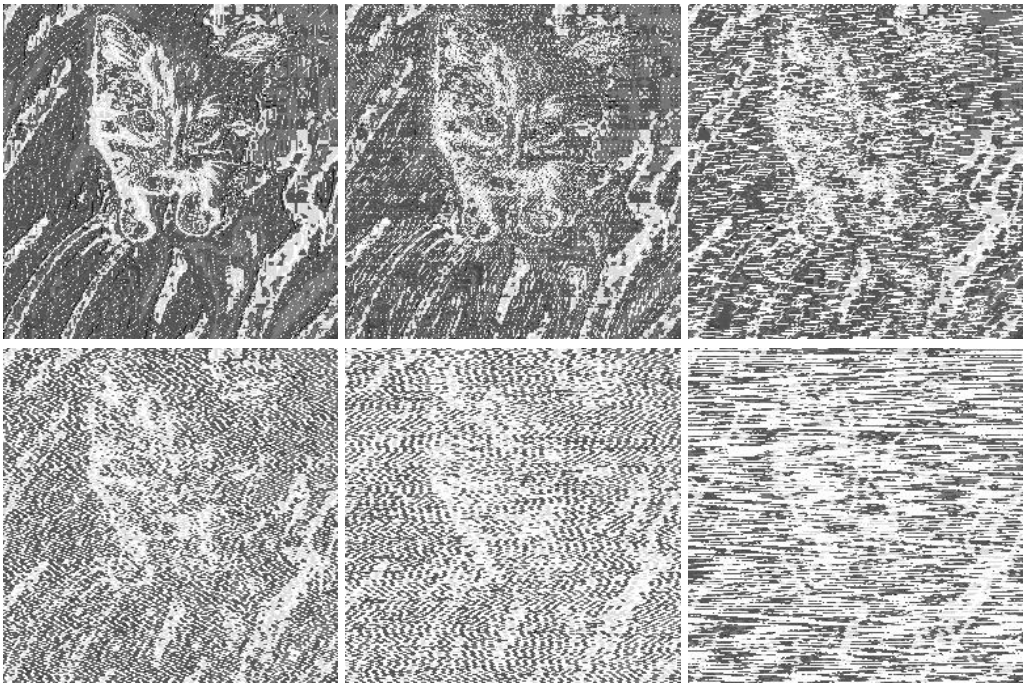


Figure 4.3: Rates of transmission for 16QAM: (top) 1kHz, 2KHz, 5kHz, (bottom) 10kHz, 50kHz, 100kHz

4.1 Bit Error Rate

The Bit Error Rate (BER) is simply the percentage of bits that are received in error with respect to the original transmission. This is computed for the image transmissions of the advanced modulation schemes, and for the only test of a random bit stream transmission which yielded a same-size output for On-Off Keying, in Table 4.1. The 4PAM and 16QAM results have a surprisingly high Bit Error Rate, seeming to suggest that the test bed did not perform particularly well despite visible success at lower frequencies. However they can be seen to approach a random $50\%BER$ as the frequency increases which agrees with a visual interpretation of the data.

Baud Rate (kHz)	OOK BER ($\%$)	4PAM BER ($\%$)	16QAM BER ($\%$)
1	5.06	46.20	45.92
2	–	45.59	46.31
5	–	48.83	46.44
10	–	48.96	46.65
50	–	50.59	46.88
100	–	50.89	46.80

Table 4.1: Table of Bit Error Rate for transmission of OOK, 4PAM and 16QAM at different baud rates

4.2 Signal to Noise Ratio

The Signal to Noise Ratio (SNR) is calculated as the ratio of the signal power to the noise power in the system, where power is the mean squared value of the amplitude of the signal.

$$SNR = \frac{P_{signal}}{P_{noise}} = \left(\frac{A_{signal(RMS)}}{A_{noise(RMS)}}\right)^2 \tag{4.1}$$

The Root-Mean-Squared (RMS) amplitude of the signal is calculated from the symbols which were transmitted. The amplitude of the noise is taken as the difference between the received values stored in the receiver's bitmasks (before Maximum Likelihood estimation is performed), and the symbols they represent. A graph of SNR against frequency is produced in Figure 4.4 for 4PAM and 16QAM. The SNR of the test bed as measured is not particularly large, with a large noise component. The SNR can also be seen to drop off at higher frequencies, which is expected from the unshielded wire setup, and this is clear in the images. The 4PAM SNR plot (red) is expected to be higher due to fewer wires with which to interfere, it would therefore be expected to drop off less quickly as the frequency increased, although whether the plot is characteristic of the system is unclear .

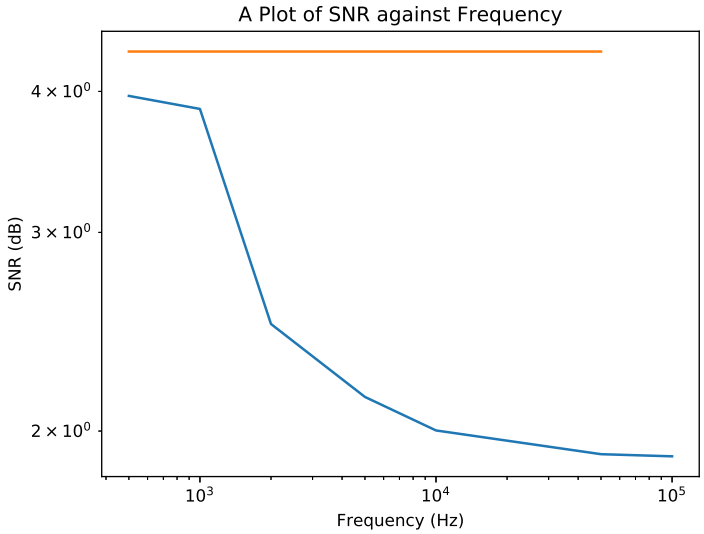


Figure 4.4: Figure of SNR against Frequency for 4PAM (red) and 16QAM (yellow)

4.3 Channel Coding

Channel coding was intended to be included as Hamming Codes, implementing forward error correction through syndrome decoding. Unfortunately due to time constraints and component issues, this was not included. However, there is a very interesting Python library which would allow for easy integration of more complex error correction, *CommPy* [40]. It includes convolutional codes with Viterbi and MAP decoders, Turbo Code encoding and decoding and random interlevers and de-interlevers.

Chapter 5

Conclusion

The project has been successful in designing and building a test bed with the ability to effectively communicate using a number of software-defined modulation schemes - the Rapsberry Pi is capable of being the sole computational and communications base of a communications platform. This test bed is also significantly less expensive than any of the investigated solutions. The communications testing suggested that there were significant problems with the Bit Error Rate achieved in communications, and it is suspected that the closely interacting wires forming separate channels to the ADCs and DACs may have become capacitively coupled at high frequencies, causing most of these problems. There were also complications with some parts which limited the capability of the project to be extended, and these complications were dealt with as efficiently as possible. The next step would be verifying that the carrier modulation was implemented in a successful manner, as the communications testing focused on baseband modulation. However, the testing showed that there is still a lot that can be done to improve the test bed at baseband frequencies first.

For future versions of the test bed, a few of the components used would likely be replaced if better ones could be found. If, for example, the DAC were replaced with one which provided the correct outputs, OFDM could be implemented with fairly few modifications to the code discussed in Section 2.3.2. The pulse generator would also ideally be replaced with a more robust quadrature sine generator. There was also significant interference at high frequencies, likely due to the exposed wires in the prototyping setup. Using a breakout board directly from the GPIO pins to a board or using shielded wiring could help reduce this problem. A major challenge for the test bed was the fact that the Raspberry Pi, although powerful, is not a real time device. Using a dedicated real-time similarly priced microcontroller connected between each Raspberry Pi and its corresponding DAC and ADC would make the transmission more reliable and independent of the computation, also allowing for the

use of GNU Radio on the Raspberry Pi, which is a very powerful tool. This would not significantly increase the price, but would provide a more effective use of resources based on the strengths of the Raspberry Pi as compared to a fully real time device. Such a microcontroller would need to be used in conjunction with Raspberry Pis, however, as they are much more powerful.

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 any stray apostrophes

Bibliography

[1] J. Mitola. “Software radios-survey, critical evaluation and future directions”. In: *NTC-92: National Telesystems Conference*. IEEE, 1992. DOI: 10.1109/ntc.1992.267870.

[2] Mihir Patkar. *9 Things You Wanted to Know About Raspberry Pi*. [Online: Accessed 20-04-2018]. 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] National Instruments. *Software Defined Radio*. [Online: Accessed 27-04-2018]. 2017. URL: http://www.ni.com/en-gb/innovations/wireless/software-defined-radio.html.

[5] GNU Radio. *GNU Radio*. [Online: Accessed 12-01-2018]. URL: https://gnuradio.org/.

[6] Gareth Hayes. *GNU Radio and rtl sdr on Raspberry Pi*. [Online: Accessed 12-01-2018]. July 14, 2014. URL: http://garethhayes.net/gnu-radio-rtl_sdr-raspberry-pi/.

[7] M. Dickens, B. Dunn, and L.J. Nicholas. “Design and Implementation of a Portable Software Radio”. In: *IEEE Communications Magazine* 46.8 (2008), pp. 58–66. DOI: 10.1109/mcom.2008.4597105.

[8] Shin Guan Ku, A.W.C. Tan, and Heng Siong Lim. “A software defined radio testbed for simulation and real-world testing of RF subsampling receiver”. In: *International Conference on Frontiers of Communications, Networks and Applications (ICFCNA 2014 - Malaysia)*. Institution of Engineering and Technology, 2014. DOI: 10.1049/cp.2014.1408.

[9] 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.

[10] Johannes Schmitz et al. “Distributed software defined radio testbed for real-time emitter localization and tracking”. In: *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2017. DOI: 10.1109/iccw.2017.7962829.

[11] Yijun Qiao, Harald Haas, and Edward Knightly. “A Software-defined Visible Light Communications System with WARP”. In: *1st ACM Workshop on Visible Light Communication Systems*. 2014.

[12] Soumaya El Barrak et al. “Implementation of a low cost SDR-based Spectrum Sensing prototype using USRP and Raspberry Pi Board”. In: 4th International Conference on Automation, Control Engineering and Computer Science (ACECS-2017). 2017.

[13] 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.

[14] Gianni Pasolini, Alessandro Bazzi, and Flavio Zabini. “A Raspberry Pi-Based Platform for Signal Processing Education [SP Education]”. In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 151–158. DOI: 10.1109/msp.2017.2693500.

[15] MathWorks. *Simulink defined radio*. [Online: Accessed 03-02-2018]. 2015. URL: http://www.simulinkdefinedradio.com/.

[16] Raspberry Pi Foundation. *Raspbian*. [Online: Accessed 04-09-2017]. 2018. URL: https://www.raspberrypi.org/downloads/raspbian/.

[17] James Mackenzie. *Headless Raspberry Pi Setup*. [Online: Accessed 09-10-2017]. Jan. 2, 2017. URL: https://hackernoon.com/raspberry-pi-headless-install-462ccabd75d0.

[18] Simon Tatham. *PuTTY: a free SSH and Telnet client*. [Online: Accessed 11-09-2017]. June 8, 2017. URL: https://www.chiark.greenend.org.uk/~sgtatham/putty/.

[19] RealVNC. *Download VNC Viewer*. [Online: Accessed 16-09-2017]. 2018. URL: https://www.realvnc.com/en/connect/download/viewer/.

[20] Eduard Bloch and Christian Lackas. *System Administration Utilities. VPNC*. [Online: Accessed 30-10-2017]. Nov. 23, 2016. URL: https://manpages.debian.org/testing/vpnc/vpnc.8.en.html.

[21] Jeff Forcier. *Welcome to Paramiko!* [Online: Accessed 16-10-2017]. 2017. URL: http://www.paramiko.org/.

[22] NumPy Developers. *NumPy*. [Online: Accessed 27-01-2018]. 2017. URL: http://www.numpy.org/.

[23] Almar Klein. *imageio. Python library for reading and writing image data*. [Online: Accessed 24-03-2018]. 2014. URL: https://imageio.github.io/.

[24] Matplotlib Development Team. *Matplotlib*. [Online: Accessed 02-04-2018]. 2018. URL: https://matplotlib.org/.

[25] joan2937. *The pigpio library*. [Online: Accessed 04-12-2017]. Apr. 5, 2018. URL: http://abyz.me.uk/rpi/pigpio/index.html.

[26] Scott Chacon. *Git*. [Online: Accessed 04-02-2018]. URL: https://git-scm.com/.

[27] Python Software Foundation. *RPi.GPIO 0.6.3*. [Online: Accessed 05-09-2017]. Oct. 30, 2016. URL: https://pypi.org/project/RPi.GPIO/.

[28] Matt. *Simple Guide to the Raspberry Pi GPIO Header and Pins*. [Online: Accessed 09-02-2018]. June 9, 2012. URL: https://www.raspberrypi-spy.co.uk/2012/06/simple-guide-to-the-rpi-gpio-header-and-pins/.

[29] Sebastian Dahlgren. *Using paramiko to send SSH commands*. [Online: Accessed 02-10-2017]. Oct. 22, 2012. URL: http://sebastiandahlgren.se/2012/10/11/using-paramiko-to-send-ssh-commands/.

[30] Embedded Linux Wiki. *RPi Low-Level Peripherals*. [Online: Accessed 04-02-2018]. Jan. 19, 2017. URL: https://elinux.org/RPi_Low-level_peripherals.

[31] Raspberry Pi Stack Exchange. *What are the Electrical Specifications of GPIO pins?* [Online: Accessed 09-11-2017]. Jan. 11, 2017. URL: https://raspberrypi.stackexchange.com/questions/60218/what-are-the-electrical-specifications-of-gpio-pins/60219#60219.

[32] Mosaic Industries. *GPIO Electrical Specifications*. [Online: Accessed 04-04-2018]. URL: http://www.mosaic-industries.com/embedded-systems/microcontroller-projects/raspberry-pi/gpio-pin-electrical-specifications.

[33] Gert Van Loo. *GPIO Pads Control. Errata*. [Online: Accessed 27-11-2017]. Aug. 2, 2012. URL: https://www.scribd.com/doc/101830961/GPIO-Pads-Control2.

[34] Arduino. *ARDUINO UNO REV3*. [Online: Accessed 12-09-2017]. 2017. URL: https://store.arduino.cc/arduino-uno-rev3.

[35] petzval. *Accurate timing for real time control*. [Online: Accessed 18-09-2017]. Aug. 10, 2013. URL: https://www.raspberrypi.org/forums/viewtopic.php?t=52393.

[36] digibird1. *Raspberry Pi as an Oscilloscope @ 10 MSPS*. [Online: Accessed 02-10-2018]. 2014. URL: https://digibird1.wordpress.com/raspberry-pi-as-an-oscilloscope-10-msps/.

[37] 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.

[38] Joonas Pihlajamaa. *Benchmarking Raspberry Pi GPIO Speed*. [Online: Accessed 13-10-2017]. Feb. 15, 2015. URL: https://digibird1.wordpress.com/raspberry-pi-as-an-oscilloscope-10-msps/.

[39] joan2937. *pigpio C Interface. gpioHardwareClock*. [Online: Accessed 03-24-2018]. Mar. 20, 2018. URL: http://abyz.me.uk/rpi/pigpio/cif.html#gpioHardwareClock.

[40] Veeresh Taranalli. *CommPy: Digital Communication with Python*. Version 0.3.0. [Online: Accessed 14-11-2017]. 2015. URL: https://github.com/veeresht/CommPy.

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:	Date: 2·11·17

Assessing the Risk*

You can do this for each hazard as follows:

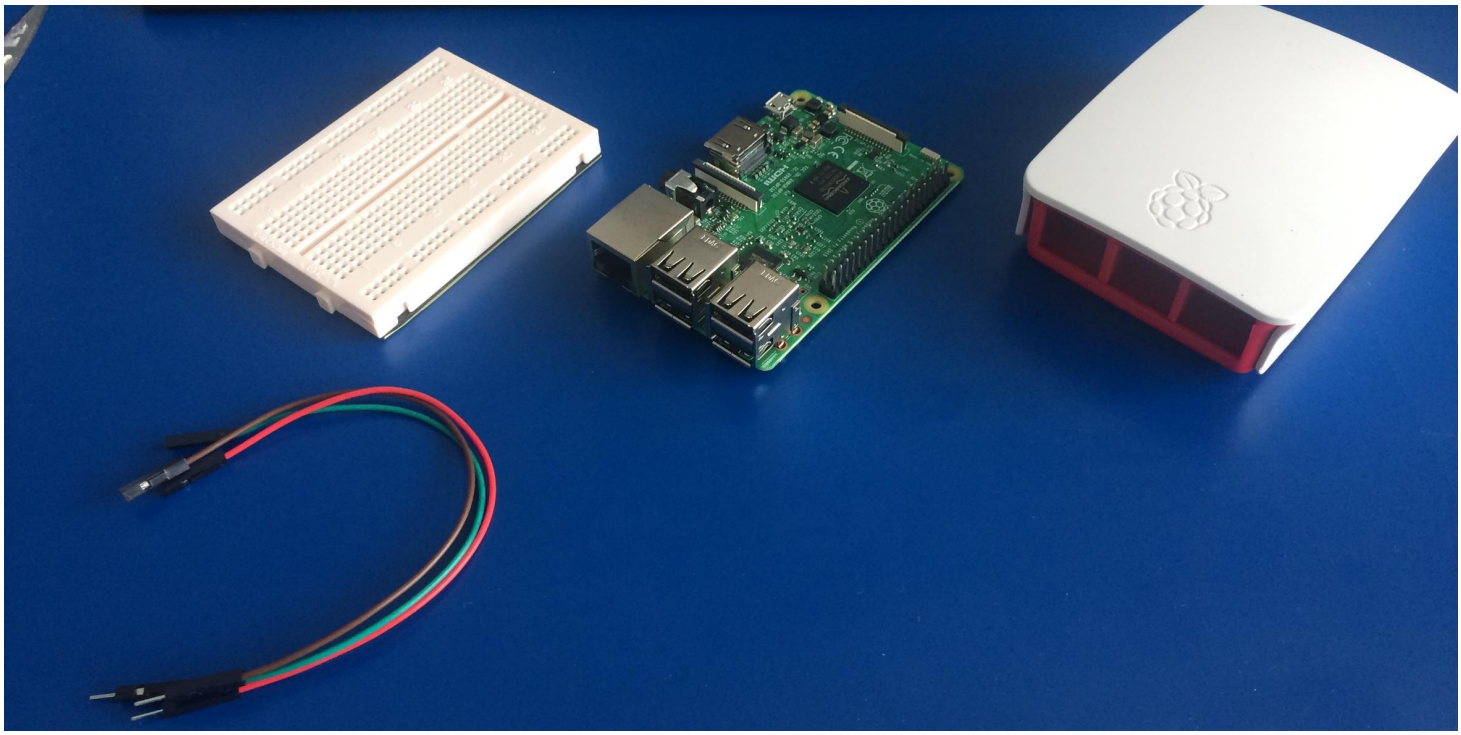
- Consequences: Decide how severe the outcome for each hazard would be if something went wrong (i.e. what are the Consequences?) Death would be "Severe", a minor cut to a finger could be regarded as "Insignificant".
- Likelihood: How likely are these Consequences to actually happen? Highly likely? Remotely 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

CONSEQUENCES	LIKELIHOOD (or probability)			
	High	**Medium**	**Low**	**Remote**
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

UNIVERSITY OF OXFORD

Supplementary Questions for 4th Year Project Students

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 department or at home		