University of Cape Town

EEE3096S

Embedded Systems II

Practicals

Keegan Crankshaw

22 April 2018

Introduction

Welcome to the practicals for EEE3096S. These instruction are applicable to all practicals so please take note.

It is critical to do the pre-practical work. Tutors will not help you with questions with answers that would have been known had you done the pre-practical work.

The practical handbook introduces the important tools, knowledge and skills you will need for these practicals. While references to more detailed information is given, students are encouraged to learn more about these topics using the internet and to take responsibility for their learning.

A special thanks to the tutors and other members who may have contributed to this practical manual by implementing practicals, writing up instructions and generating marking resources.

Practical Instructions

Failure to meet these instructions will result in negative marks.

- Please check Vula for your practical time. Many practicals require demonstrations, so you will need to attend.
- All practicals need to be uploaded to github
- All files submitted to Vula need to be in the format

```
pracnum_studnum1_studnum2.fileformat
```

- All text assignments must be submitted as pdf
- Each pdf should contain a link to your github repository
- All *reports* (that is, submissions for Prac 2, Project A and Project B) need to be written in Latex.
- Practicals 0 and 1 are to completed alone. Practical 2 onwards is to be completed in groups of two. You can choose your own groups as long as the person is in the same lab session as you. Ideally, your partner should stay the same for the duration of the course.

0 Prac 0 - Setting up and Connecting to Your Pi

The Raspberry Pi is a powerful SBC. While it can be used as a lightweight computer, its true strengths come from how versatile the platform is[1]. We will be using the Pi throughout the rest of the course in both practicals and the mini project. Before you attend the first tutorial session is is required that you setup your Pi so it is ready for operation. In addition, if you are unfamiliar with Linux, Git, ssh, and bash we strongly recommend you go through the additional exercises as outlined in the lab handbook.

0.1 Required before attending the first lab session: Setup your Pi

Before arriving at the first practical lab session of this course you are required to do the following. If you have trouble completing the following you should attend the hotseat on **Friday the 19th of July** to get help before the first lab session.

1. Setup your Pi and get SSH working, as per the lab handbook.
2. Create a GitHub account, and sign up for the GitHub Education Pack.
3. Enable Ethernet pass-through on the Pi to give it internet access. Instructions can be found in the lab handbook.
4. Fetch the practical repository
   (a) SSH in to the Pi
   (b) Ensure you have connectivity by trying to ping a website such as google.com. If not, debug your connection.
   (c) Fetch the repository

```
$ git clone --depth=1 https://github.com/kcranky/EEE3096S.git PracSource/
```

   (d) Create a copy of the original git folder

```
$ mkdir mypracs
$ cp -r PracSource mypracs
```

   (e) Remove the git settings

```
$ cd mypracs
$ rm -rf .git
```

   (f) Initialise a new repository, and put it on GitHub as per the instructions in "A Quick Git Get Go" in the lab handbook.

[1] Some examples include home automation, media centre or even as a robot

0.2 Submissions

There are no submissions for this practical.

1 Practical 1 - Git, Bash, GPIO

1.1 Overview

This practical sets out to familiarise you with the Pi and complete a simple programming task. If you have not yet done so, complete Prac 0. Ensure that you can SSH into the pi, as all the tasks for this prac require it.

To be completed individually.

Due date: See Vula

1.2 Pre-Prac Tasks

- Complete Prac 0 to have your Pi set up and configured
- Read the sections "Inputs" and "Outputs" on the RPi.GPIO documentation, available here
- Update your pracsource git repository, as there has been an update to the Python template

1.3 Pre-prac Requirements

This section covers what you will need to know before starting the practical.

- Have your Raspberry Pi Set up as per the requirements of Prac 0.
- Have an understanding of how you can edit text files on the Pi using an editor such as nano, or using VNC and a GUI-based editor.
- Have a basic understanding of Git

1.4 Hardware Required

- Raspberry Pi with configured SD Card
- RPi Power Supply
- Ethernet Cable
- A breadboard
- 2 x push buttons
- 3 x LEDs
- 3 x Resistors
- Dupont Wires

1.5 Outcomes of this Practical

You will learn about the following topics:

- Basic GPIO Usage
- Interrupts
- Debouncing
- Git
- Bash
- SSH

1.6 Deliverables

At the end of this practical, you must:

- Demonstrate your working implementation of the binary counter to a tutor
- Single PDF with your screenshots from the Terminal task to Vula in a pdf document. This PDF document should also contain a link to your GitHub repository.

1.7 Walkthrough

This practical consists of two parts. The terminal task can be completed at home and does not require demonstration to a tutor.

1.7.1 Terminal Task

While there are many Graphic User Interfaces (GUI) available for various distributions of Linux including Raspbian, it is often necessary to operate an embedded system without a GUI given the limited processing and memory resources of the device. Consequently it is important to learn to use Linux through the command line shell. A very common shell on many Linux systems is Bash. Learning about and being comfortable with the command line will help you greatly in working with embedded systems. The tasks below will introduce basics to you, but you should consult the manual for more information.

Start by SSH'ing into your Pi and create a folder called <your_student_number>.
Run the following commands, and take a screenshot after each command:

- $ ls (must show the folder you just created)
- $ ifconfig (eth0 must be visible)
- $ lscpu (number of cores must be visible)

Your result should look something like this:

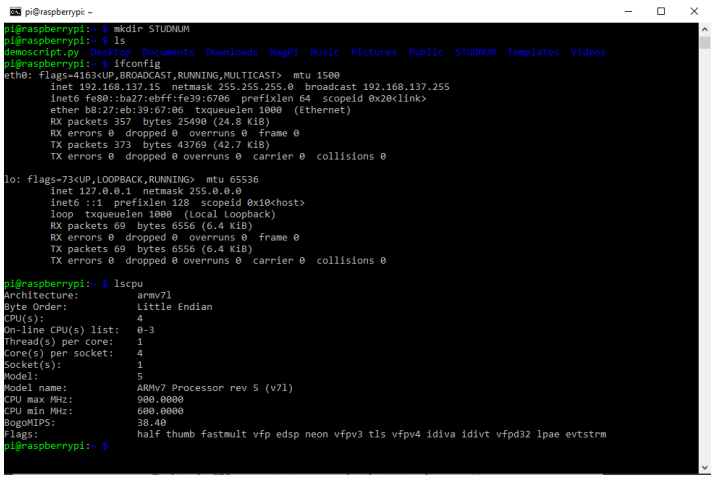

Figure 1: Example output after running the above commands

1.7.2 Programming Task

In this task you will develop a simple 3-bit binary counter, the values of which will be placed on LEDs connected to the Pi. The value on the counter should change depending on a button press. Be sure to use git to keep track of changes to your code. For instructions on using Git, refer to the lab handbook.

- Start by connecting to your Pi through VNC or SSH (see lab handbook)
- If not installed, install the Python GPIO libraries, as explained here. (They should be installed by default.)
- Connect 3 LEDs and 2 Push buttons to the GPIO pins of the Pi - taking care of which pins to use. Look at pinout.xyz and be sure to not use special purpose pins
- Read the Python Tips and tricks in the lab handbook to gain understanding of the commands and why they are used.
- Create a copy of the template in the mypracs folder.
- Write your code. It's suggested you work incrementally, committing your code using git each time you accomplish something.
  - Take a look at the RPI.GPIO examples, available here
  - Start by turning on and off a single LED in the main() method. Delay between toggling the GPIO by using something like `time.sleep()`
  - Now, use a button to toggle the state of the LED. Look at the lab handbook for help with debouncing and interrupts in Python.
  - Finally, implement a system that displays a binary value on 3 LEDs. Use one button to increase the value, and another button to decrease the value. The values should wrap around (i.e. increasing "111" should then display "000" and decreasing "000" should display "111"

- Hints
  - Look at *itertools.product* to generate a list of binary values.
  - Use a single integer counter as an index to the Python array that you've created. Make use of Python's `global` construct to do so
- Demonstrate this implementation to a tutor to get signed off, and push your code to GitHub.
- Submit a PDF as per the deliverables section

1.8 Mark Allocations

Marks will be given for:

- Correctly completing the terminal task
- Well structured and commented code
- Meaningful git commit messages
- A correct demonstration

Marks will be deducted for:

- Not using interrupts and debouncing on your button presses
- Not submitting files in the correct format
- Not linking to the practical on GitHub
- Copying another student's code
- Late submission

2 Prac 2 - Python vs. C

2.1 Overview

Pre-prac Requirements are ABSOLUTELY REQUIRED for this practical. The practical is fairly straightforward, but there is a lot of information to take in and you will likely not be able to complete it correctly if you do not understand the concepts.

This practical is designed to teach you core concepts which are fundamental to developing embedded systems in industry. It will show you the importance of C or C++ for embedded systems development. We'll start by running a program in Python, and comparing it to the exact same program, but written in C++. This shows us the importance of using a particular language. From there, we we'll try and improve the performance of the C code as much as possible, by using different bit widths, compiler flags, threading, and particular hardware available in the ARM Processor.

The quest for optimisation can lead one down a long, unending spiral. What is important to take away from this practical is awareness of concepts, the ability to use good practice when benchmarking.

There's a considerable flaw in this investigation in that there is no comparison of results to our golden measure. Meaning, by the end of the practical, you will (ideally) have considerably faster execution times - but your speed-up could be entirely useless if the result you're getting is not accurate. Comparison of accuracy is left as a task for bonus marks.

2.2 Pre-prac Requirements

2.2.1 Knowledge areas

This section covers what you will need to know before starting the practical. This content will be covered in the pre-prac videos.

- Introduction to benchmarking concepts
- Introduction to cache warming and good testing methodology (multiple runs, wall clock time, speed up)
- Instruction set architecture, bit widths and hardware optimisations
- Makefiles
- Report writing

2.2.2 Videos

Some videos were made to help you with this practical. Here they are:

- An Introduction to Benchmarking
- Compilers, Toolchains and Makefiles
- Report Writing

2.2.3 Submissions for Pre-Prac

Before the practical begins, you need to submit a PDF on Vula of the IEEE report, with the Title, Authors and Introduction edited to reflect the details of this practical. A due date will be available on Vula.

2.3 Outcomes

By the end of this practical you will have an appreciation for the importance of benchmarking, lower level languages, ISA and integrated hardware.

- Benchmarking
- Latex and Overleaf
- Bit widths
- Compiler Flags
- Instruction Sets
- Report Writing

2.4 Deliverables

At the end of this practical, you must:

- Submit a report no longer than 3 (three) pages detailing your investigation. You must use IEEE Conference style. You must cite relevant literature.

2.5 Hardware Required

- Raspberry Pi
- SD Card
- Ethernet Cable

2.6 Walkthrough

2.6.1 Overview

1. Establish a golden measure in Python
2. Compare Python implementation to C++ implementation, in terms of accuracy and speed
3. Optimise the C++ code through parallelization and compare speeds
4. Optimise the C++ code through compiler flags
5. Optimise the C++ code through different bit widths, ensuring that that changes in accuracy are accounted for
6. Optimise the C++ code using hardware level features available on the Raspberry Pi
7. Optimise the C++ code using a combination of parallelization, compiler flags, bit widths, and hardware level features

2.6.2 Detailed

1. Start by getting the resources off of GitHub

   ```
   $ cd ~/PracSource
   $ git pull origin master
   ```

   This updates the repository to ensure you have the Prac2 source files.

2. Read the README in the Prac2 directory. Here is a direct link.
3. Enter into the Prac 2 source files using the `cd` command. Run the Python code to establish a golden measure. Be sure to use proper testing methodology as explained in the pre-prac content.
4. Now, run the C code (don't forget to compile it first, and every time you make a change to the source code!). This code has no optimisations in it, and also uses floats - just like the Python Implementation [2].
5. How does the execution speed compare between Python and C when using floats of 64 bits? Record your results and comment on the differences.
6. Now let's optimise through using multi-threading.
   (a) You can compile the threaded version by running `make threaded`
   (b) The number of threads is defined in `Prac2_threaded.h`
   (c) Run the code for 2 threads, 4 threads, 8 threads, 16 threads and 32 threads.

[2] Floats in Python can get really weird - they don't stick at a given 32 bits. But for the sake of this practical, we're going to assume they do.

(d) Does the benchmark run faster every time? Record your results, and discuss the effects of threading in your report.

7. Record your results, taking note of the most performant one. What can you infer from the results?

8. Now let's optimise through some compiler flags

   (a) Open the makefile, and in the `$(CFLAGS)` section, experiment with the following options:

Table I: Compiler Flags for optimisation

Flag	**Effect**
-O0	No optimisations, makes debugging logic easier. The default
-O1	Basic optimisations for speed and size, compiles a little slower but not much
-O2	More optimisations focused on speed
-O3	Many optimisations for speed. Compiled code may be larger than lower levels
-Ofast	Breaks a few rules to go much faster. Code might not behave as expected
-Os	Optimise for smaller compiled code size. Useful if you dont have much storage space
-Og	Optimise for debugging, with slower code
-funroll-loops	Can be added to any of the above, unrolls loops into repeated assembly in some cases to improve speed at cost of size

9. Record your results, taking note of the most performant one. Which compiler flags offered the best speed up? Is it what you expected?

10. Now let's optimise using bit widths

    (a) The standard code runs using **float**

    (b) Start by finding out how many bits this is.

    (c) Run the code using 3 bit-widths: double, float, and __fp16. How do they compare in terms of speed and accuracy? Note: for __fp16, you need to specify the flag "-mfp16-format=ieee" under your $CC flags in the makefile

11. Now let's optimise using hardware level support on the Raspberry Pi

    (a) The Pi has various instruction sets. See what they are by running "cat /proc/cpuinfo" in a terminal

    (b) You can set a floating point hardware accelerator with the "-mfpu=" flag

    (c) The important ones for you to try are:

Table II: Compiler Flags for the Floating Point Unit

mfpu Flag	**Description**
none specified	Default implementation
vfpv3	Version 3 of the floating point unit
vfpv3-fp16	Equivalent to VFPv3 but adds hfp16 support
fpv4	Version 4 of the floating point unit
neon-fp-armv8	Advanced SIMD with Floating point
neon-fp16	Advanced SIMD with support for half-precision
vfpv3xd	Single Precision floating point
vfpv3xd-fp16	Single precision floating point, plus support for fp16

12. Find the best combination of bit-width and compiler flags to give you the best possible speed up over your golden measure implementation.

13. Is there anything else you can think of that may increase performance? Perform your experiments and record your results. Bonus marks will be awarded for additional experimentation. (Hint: How can you test that the results of the Python code - our golden measure - and our fully optimised code produce the same results?)

14. Finally, record your methodology, results and conclusion in IEEE Conference format using Latex (Overleaf is recommended as an editor - See instructions in handbook).

2.7 The Report

The report you need to complete needs to follow the IEEE Conference paper convention. This style you can find online. It is recommended you use Overleaf as an editor, as it allows collaboration and handles all packages and set up for you. In your report, you should cover the following:

- In your introduction, briefly discuss the objectives and core finding of your research
- In your methodology, discuss each experiment and how you plan to run it.
- In your results, record all your results in either tables or graphs. Be sure to include only what is relevant, but not miss out on anything that might be interesting. Briefly justify why you got the results you did, and what each experiment did to affect the run time of your results.
- In your conclusion, mention what you did, and what your final findings are (e.g. "The program ran fastest when..."). This should only be a few sentences long.

2.8 Marking

Your report will be marked according to the following:

- Following instructions
- Covering all aspects listed in this practical
- The quality of your report

Additional marks will be awarded for:

- Further experimentation
- Optimisation beyond what is described in this practical

Marks will be deducted for the following:

- Not using LaTeX
- IEEE Conference paper format not used

3 Prac 3 - I2C and PWM

3.1 Overview

Before connecting to the internet for the first time, you may have noticed that the time on your Pi is a little strange. This is because the Pi doesn't have what is called an RTC (real time clock). Instead, it relies NTPD (Network Time Protocol Daemon) to fetch, set and store the date and time. However, this might be problematic as you may not always have an internet connection, and if your Pi doesn't have the correct localisation options, you may end up with the wrong time due to timezone settings. It's possible to add an RTC to the Raspberry Pi to hold the system time correctly, but for this practical we're simply going to interface with the RTC using I2C, and set the time using buttons and interrupts.

3.1.1 Design overview

You will be creating a modified version of a binary clock. For your convenience, an image displaying the expected operation is shown below in Figure 2.

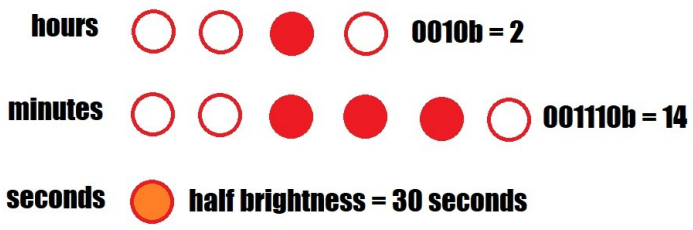

Figure 2: A Modified Binary Clock Showing 2:14 (AM or PM)

Two buttons should be connected, using interrupts and debouncing, which do the following:

1. Button 1 - Fetches the hours value from the RTC, increases it by 1, and writes it back to the RTC.
2. Button 2 - Fetches the minutes value from the RTC, increases it by 1, and writes it back to the RTC.

You cannot use any time libraries in your script, i.e. make sure you are using the I2C communication protocol between the RTC and Pi in your script.

3.2 Pre-prac requirements

This section covers what you will need to know before starting the practical.

3.2.1 Knowledge Areas

- I2C
- PWM
- Knowledge about BASH as learnt in previous pracs
- Read the data sheet for the MCP7940M [3]
- You should acquaint yourself with the Wiring Pi documentation, available here.

3.2.2 Pre-prac Submissions

A circuit diagram for the practical. You will need to upload a schematic detailing how you are going to set up the RTC and connect it to the Raspberry Pi. It should also show how the buttons and LEDs are connected to the Pi. Software used for drawing the circuit is up to you. [4]

3.3 Outcomes

You will learn about the following aspects:

- I2C
- Real Time Clocks
- Wiring Pi
- Starting a script on boot on the Raspberry Pi

3.4 Deliverables

At the end of this practical, you must:

- Demonstrate your working implementation to a tutor
- Submit your code on Vula alongside a short write-up (no more than three - 3 - pages) detailing how you completed the practical. It is strongly recommended you do use git, but a GitHub link is not required for practical submission. See the write up format below for further guidelines. Your code and write up should be contained in a compressed folder. Your write up should be a PDF. It is not required to complete the write up using LaTeX.

[3] The Pi has onboard resistors for I2C, so you don't need to use those.

[4] You should be aware that Wiring Pi has 3 different potential modes for pin usage. Be aware of that during config and your wiring.

3.5 Hardware Required

- Configured Raspberry Pi
- RPi Power Source
- Ethernet Cable
- A breadboard
- 2 x push buttons
- 11 x LEDs (total)
- 11 x Resistors (total)
- Dupont Wires
- RTC
- Capacitors (kit)
- Crystal

3.6 Walkthrough

1. If you didn't in Prac 0, start by enabling I2C in raspi-config
2. Do a git pull in the prac source folder to fetch the Prac 3 content
3. Build the circuit you designed in the pre-prac
4. Run `$ gpio i2cdetect` to see if you can see the RTC (0x6F)
5. Open `BinClock.c` and write the code required. Some function templates are made available to give you a guide, but you may be required to write more functions as you see fit. Function definitions are placed in `BinClock.h`
    - Be sure to take care with regards to which pin numbering is used. You can run `$ gpio readall` to check pinouts and current assignments
6. As a fun task, you can get the clock to run on startup using `rc.local`. You can read through this guide.[5]

3.7 Some Hints

1. Read the Docs. This includes the datasheet for the RTC (which you will absolutely have to do), as well as the documentation of Wiring Pi.
2. You will need to debounce your button presses. You can use hardware debouncing, but this circuit is a bit complex, so to save space we recommend software debouncing.
3. The source code provided to you has a lot of implementation in it already. Ensure you read and understand it before embarking out on the practical.
4. Wiring Pi doesn't have a cleanup function like RPi.GPIO (Python). It's suggest you write your own that's caught and executed upon a keyboard interrupt.
5. You may notice a function called `toggleTime`. This function pulls the time from the Raspberry Pi, and writes it to the LEDs. This is a nice option for "reloading" the system time to RTC.

[5] Admittedly, this isn't a very efficient or effective way to keep track of time. There are services that will keep track of time for us in the Pi OS, rather than us polling the RTC every second. We could even write our own code better - letting the time on the Raspberry Pi "freewheel" and having an interrupt fetch the time from the RTC every few minutes (or hours, or days or just on boot). At this point though, it becomes more sensible to simply use the hwclock service.

3.8 Write up Format

Your write up should consist of the following headings:

1. Introduction
2. A UML use-case diagram of the system
3. I2C communication using Wiring Pi
   (a) Initialisation
   (b) Send data
   (c) Receive Data

   Include example timing diagrams in your descriptions
4. A short paragraph on interrupts and debouncing, and why they're important in Embedded Systems (one advantage of each is enough)
5. The circuit diagram from the pre-prac, with any changes you've made updated in the circuit diagram

4 Prac 4 - SPI and Threading

4.1 Overview

While the Pi 3B+ does have an audio jack, it uses a purely PWM (pulse width modulation) based implementation for audio. Audiophiles among you may know that this is not a great way of playing audio. You can read more about the Raspberry Pi and it's audio jack here.

To improve audio quality, we can use a DAC (digital to analog converter).

4.2 A Very Short Introduction to Digital Audio

If you'd like to better understand the basics of audio quality, I suggest reading this article. If you find any of this interesting even in the slightest), or you're interested in signal processing, I strongly suggest the DSP course in fourth year.

Sound waves are continuous waves in time. Unfortunately, we can't represent continuous waves in digital systems. Instead, we take discrete samples of the continuous waves at a specific interval (with help from Nyquist) to be able to reconstruct the wave to the best of our ability. This rate is called the *sample rate*. You may recognise common audio sample rates, such as 44.1kHz, which is standard CD quality.

Figure 3 below[6] shows the difference between the continuous and discrete representations. The sampling interval is represented by the value T. From basic physics, we know that $f = \frac{1}{T}$. So if we're using a CD for playback, we can determine the sampling period, T, to be $T = \frac{1}{f} \approx 22.7\mu S$.

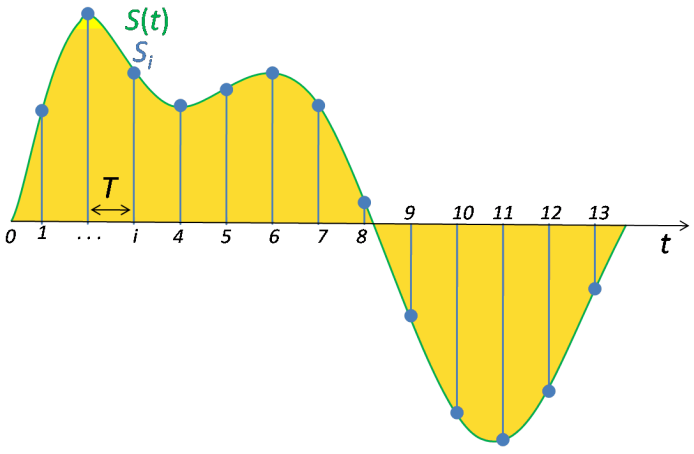

Figure 3: Signal sampling representation. The continuous signal is represented with a green coloured line while the discrete samples are indicated by the blue vertical lines.

[6]By Email4mobile (talk) http://en.wikipedia.org/wiki/File:Signal_Sampling.png, Public Domain, https://commons.wikimedia.org/w/index.php?curid=8693098

Recall from Prac 2, where we spoke about about bit widths. The greater the bit width, the more accurate our number. Now consider the DAC we're using the MCP4911. It is a 10 bit DAC, meaning we have 10 bits to work with. If you recall from your lectures, it means we can split the voltage outputs into 10 distinct levels. CD quality uses 16 bits.

Our practical uses 8 bit 16kHz audio - the lowest possible quality we could make! This is purely because we're using a cheap DAC. So unfortunately the audio you will get at the output will sound worse than what you might expect from a song - but it won't sound much different from if you had to play the same file through a high end audio system. What's important to take away from this practical is an introduction to SPI, and some knowledge about audio, hard time constraints, and an appreciation that the Raspberry Pi may not always be the best choice for your application.

Audio processing has a hard time requirement. This means that the audio events need to happen at a very specific time with no variations. Because the Raspberry Pi uses a Linux based operating system, there's no ability to for it to perform hard time operations. Instead it performs the operations to the best of it's ability.

Because audio is a hard time constraint, we need to make sure that samples are ready to play as soon as they are requested. To do this, we use a technique called "circular buffering". Correct and complete implementation of a circular buffer can get tricky, so in this practical we're mimicking it by using a single array split into two parts. As the one half of the buffer (array) loads, the other half plays. We need to put measures in place to ensure that we don't write over data we are yet to play, or try and play from the buffer before any audio samples have been written to it. Because our Pi operates faster than we need the audio to be played out over SPI, we're mostly okay, but to be safe we're going to implement these checks anyway. The buffer size that is set has consequences, too. Too small a buffer size, and the audio playback will experience "popping". Too large, and we use too much memory. For this practical, a total buffer size of 1000 samples should be sufficient.

4.3 Pre-prac requirements

This section covers what you will need to know before starting the practical.

- Read the MCP4911 Datasheet and understand the registers and what each bit represents (see page 24).
- Revise bit shifting in C/C++ from ES I
- Read the WiringPi documentation on SPI

4.4 Outcomes

You will learn about the following aspects:

- Reading from an external file
- SPI
- Threading

4.4.1 Pre-prac Submissions

A circuit diagram for the practical. You will need to upload a schematic detailing how you are going to set up the DAC and connect it to the Raspberry Pi. It should also show how the buttons are connected to the Pi. Software used for drawing the circuit is up to you. [7]

4.5 Deliverables

At the end of this practical, you must:

- Demonstrate your working implementation to a tutor
- Submit your report on Vula

4.6 Hardware Required

- Configured Raspberry Pi
- RPi Power Source (power brick)
- Ethernet Cable
- A breadboard
- 2 x push buttons
- MCP4911 DAC
- Dupont Wires
- A set of earphones

4.7 Further Instructions

- If you didn't in Prac 0, enable SPI in raspi-config
- Fetch the updated Practical code off GitHub by running git pull in the prac source, and move it in to your own folder to work with. This is a tricky practical and use of git (not necessarily GitHub) is strongly recommended for this practical
- Read through the code that is given to you, and try and understand what you are still required to do. be aware that you need to make changes to Prac4.h
- Start by writing your initialisation. You need to determine the rate at which the audio plays by setting the SPI Clock. You should also be aware that the Raspberry Pi isn't that great at holding an SPI clock. There is occasionally a scaling factor of 8/5 in the

[7] You should be aware that Wiring Pi has 3 different potential modes for pin usage. Be aware of that during config and your wiring.

SPI clock due to the fact that the SPI clock comes from the CPU clock, which changes depending on load.[8,9]. So if you want a 16kHz SPI clock, you need to multiply that by 8/5 and tell the Wiring Pi library to use a clock of 25.6kHz. You can use the formula $BITRATE = SAMPLERATE * WIDTH * CHAN * 8/5$ to determine the SPI clock you need. Recall width here refers to the total number of bits to be transferred for a single sample (16 bits).

- Write code to read in the file to the buffers. The file that you have been provided with is a 16kHz, 8 bit .raw file. This makes it simple for the sound to be processed as all that needs to be done is to read in each 8 bit character and send it to the DAC. Make sure you understand how the buffering is working.
- Create the code that plays audio over SPI to the DAC. It's a good idea to give this thread a high priority. You can read about thread priorities here.
- Create the two interrupts you require to pause playback, and stop playback. Stop playback could be considered as an "exit" as well. Pause should stop playback and reading from the buffer when first pressed. When pressed again, playback should continue.
- You can listen to the output of the DAC on earphones (given that you're passing the correct control bits). Otherwise, you need to pass it through an amplifier.[10]
- You need to demo playback, pause/play and stop to a tutor in a demo. They will examine your code to ensure you've used threading and interrupts.

4.7.1 Report Instructions

Your three page write up should consist of the following headings:

1. Introduction
   Describe the prac (what you're doing and how you're doing it) [0.5-1 pages]
2. SPI communication using Wiring Pi [0.5 pages]
   (a) Initialisation, including the clock speed calculation
   (b) Send data
3. A few sentences highlighting the importance of real time constraints, and why the Raspberry Pi (under Raspbian) is unable to implement these. [0.5 pages]
4. The circuit diagram from the pre-prac, with any changes you've made updated in the circuit diagram [0.5-1 pages]

[8]Note that this factor isn't constant - sometimes the Pi might actually run at the clock rate you give it. To resolve this issue, you can force a stable clock rate on the CPU, but that isn't necessary for this practical. Testing tells me that the Pi is more likely to downclock, requiring us to need that factor. If you want to be certain, hook the SPI clock pin out to an oscilloscope

[9]Understand why here

[10]You probably don't want to do this as the audio quality is so low in this prac.