UNIVERSITY OF BIRMINGHAM

FOURTH YEAR PROJECT - FINAL REPORT

Highly Stable Compact Digital Temperature Controller for use with Ultrastable Atomic Systems.

Author: Aaron Jones

4th Year MSci Physics Student,

School of Physics and Astronomy,

University of Birmingham.

Email: axj120@bham.ac.uk

Number: 1148420

Word Count: 8435

Supervisor: Dr. Yeshpal Singh

Research Fellow,

University of Birmingham.

Email: y.singh.1@bham.ac.uk

Keywords: Digital, Temperature Control, Arduino, Raspberry Pi, Laser, Cavity, Atomic Chamber, Vacuum Chamber

March 27, 2015

Abstract

This report discusses the design and implementation of a highly stable digital temperature controller. The controller is designed to be implemented for stabilisation of: Ultra High Finesse Cavities, Diode Lasers, Optical Equipment and for the reduction of Blackbody Radiation Gradients in Atomic Chambers. Hobbyist microcontrollers, Block Systems and High Level Languages were used to increase generality and reduce the technical barrier for development. A stability of $(3 \pm 1)mK$ was obtained for an open system, with breadboard design. A prediction of $(0.47 \pm 0.16)mK$ stability was obtained for a closed breadboard system. Significant reductions on this are predicted to be available by printing the circuit onto a PCB.

Acknowledgements

For their invaluable assistance I would like to thank: Dr Ole Kock, for his patience when explaining the clock and his support when producing written material; Wei He and Lyndsie Smith, for their invaluable support in the lab, I wish you all the best of luck in your continued research. I would also like to thank my supervisor, Dr Yeshpal Singh for proposing such an enjoyable project and for his support throughout. Finally, I would like to thank everyone in Clock lab, without you the project would not have been a success.

Contents

List of Figures

List of Circuit Diagrams

List of Tables

Chapter 1

Introduction

Quantum Technology has a large number of potential applications[37]. Many regard the first quantum technology to be the development of the transistor[37], for which Shockley, Bardeen and Brattain were awarded the 1956 Nobel Prize in Physics[36], since then the field of electronics has advanced rapidly, however, this field solely concerns itself with the quantum nature of electrons. Many more quantum technologies can be developed when entire atoms can be manipulated, such as: the Gravity Gradiometer[41], Optical Lattice Clock[34] and Quantum Magnetic Sensors[16]. These were initially developed between 2001[41] and 2005[34], however, such experiments often consist of a vast arrays of fragile electronics and optics[18]. This constrains the experiment to a lab and hence limits the usefulness of the application[31].

Throughout this report I will use a Portable Optical Lattice Clock for motivation and examples of usage, however, the techniques developed in this report are very general and may be applied to any atomic physics experiment. Specifically, optical lattice clocks are the next generation of clock technology[42]. The traditional atomic clock is a caesium clock and these have been gaining an order of magnitude in accuracy every decade[31]. However, they are now reaching a lower accuracy limit of $0.02ns$ per day[33], (a precision of $\approx 2 \times 10^{-16}$) due to their Quantum Projection Noise[33][24]. As a result of this accuracy there have been advances in the fields of Broadband Communication[34], GPS[26] [34] and Quantum Mechanics[26]. In addition the second was redefined as,

> "9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom."[25]

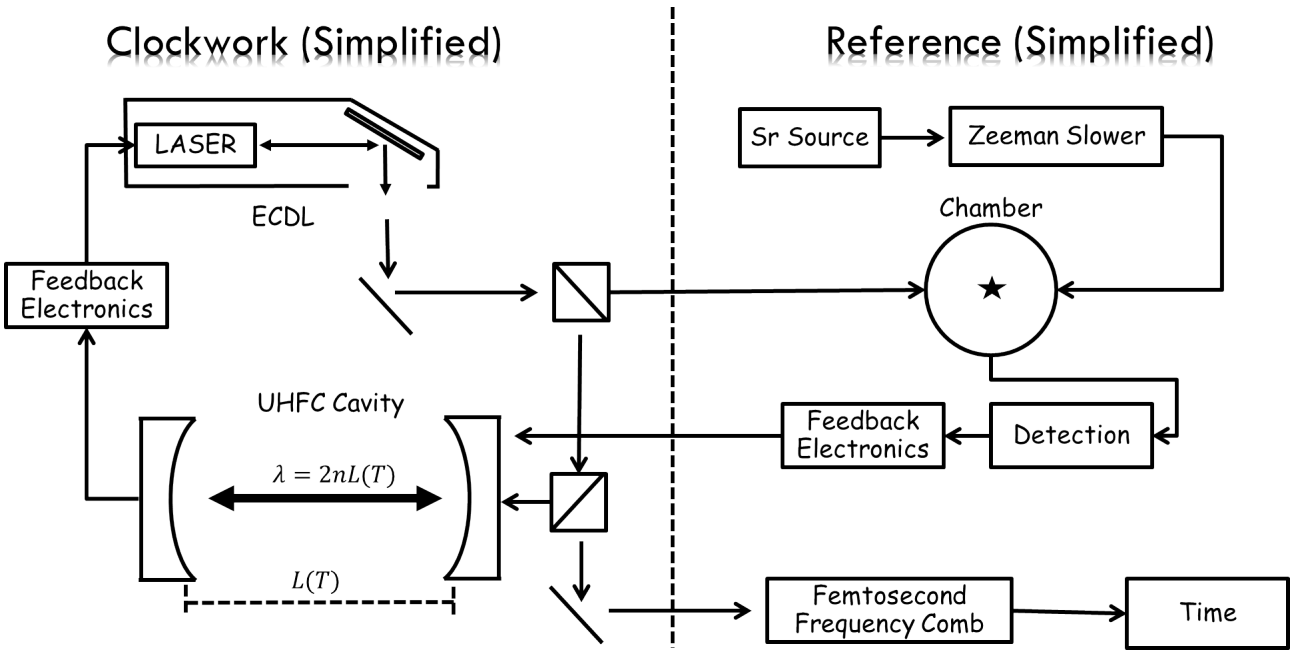


Figure 1.1: A highly simplified Optical Clock Schematic. Laser light is initially generated using a tuneable diode laser, the laser is in an External Cavity Diode Laser configuration[26]. Then this is locked to an Ultra High Finesse Fabry-Pérot Cavity and electronic feedback lowers the ECDL linewidth to the Hz range. This cavity is a local oscillator, or clockwork for the system. This laser is then used to interrogate an electronic transition in Sr. To achieve this atoms are cooled using a permanent magnet Zeeman slower, then they are loaded into a MOT using additional optics (not shown). This then forms a fundamental reference frequency.

Since time is a base unit[25], this has lead to reductions in the uncertainty of all derived units, such as the meter[25]. In order to push the boundaries of science, development of the optical clock is essential. However, due to a host of processes such as tidal movement[31], glacial melting[31] and atmospheric pressure fluctuations[31] the gravitational fluctuations around an optical lattice clock would perturb the system too much to reach its full potential[31]. Putting the system in space allows new tests of fundamental physics such as tests of Einsteins Theory of Relativity[45], Geophysics[45] and the Standard Model[51]. However, this intrinsically requires a portable system.

1.1 Optical Clocks

Optical Clocks, like all clocks typically have a clock mechanism and a reference[2]. For a Personal Computer, the clockwork is provided by the internal Quartz oscillator and the operating system will periodically check a time server for an up to date time, thus providing a reference. In an optical clock the clockwork is also a local oscillator, this time in the form of a very stable laser and the reference is an atomic transition[24]. Such clocks have a lower Quantum Projection Noise than atomic clocks: firstly, because the frequency of operation is much higher[33], but also because the atoms are in a lattice[8] and so the error is immediately averaged across the sample, reducing the measurement time compared to a single Ion clock[8]. A schematic diagram of how an optical clock works is available in Figure 1.1.

1.1.1 The Atomic Reference

The atomic reference is the section of the clock that ensures that the measurements of time are locked to a fundamental frequency. In this case the reference is obtained by manipulating the outer electron of Strontium through optical transitions. Firstly, SrO is heated and turned into a hot vapour using resistance heating or other methods[45]. Then the atoms are cooled using a permanent magnet Zeeman slower[45]. Zeeman slowers work by splitting the energy levels, progressively bringing the atoms on resonance with a counter propagating laser beam, thus by scattering photons, the Strontium is slowed[15]. Then the atoms can be loaded into a Magneto Optical Trap (MOT) which can cool and trap the atoms[22]. The magneto section uses an Anti-Helmholtz configuration coils to trap the atoms. For optical part in strontium, two stages of cooling are used, using progressively narrow linewidth transitions[45] to realise a cloud at micro kelvin temperatures[31]. The atoms can then be trapped in a 1D lattice[45] using additional optics. The atoms are then excited into the ${}^3P_0$ state using the clock laser. Assessing the florescence between ${}^1S_0$ and ${}^1P_1$ is then a measure of how many atoms were not excited during the clock interrogation[42]. The laser is offset, the process is repeated and comparison of the two florescence's generates an error for the clock laser[42]. The energy levels for Sr are shown in Figure 1.2.

1.1.2 The Optical Oscillator

The oscillator is formed using a diode laser. The diode laser is chosen because of its compact size, reduced additional equipment and tune-able properties[26]. In order to keep the oscillations stable this is placed in a External Cavity set-up. A diode laser has a wavelength dependant on a 2D parameter space consisting of temperature and current. By feeding light

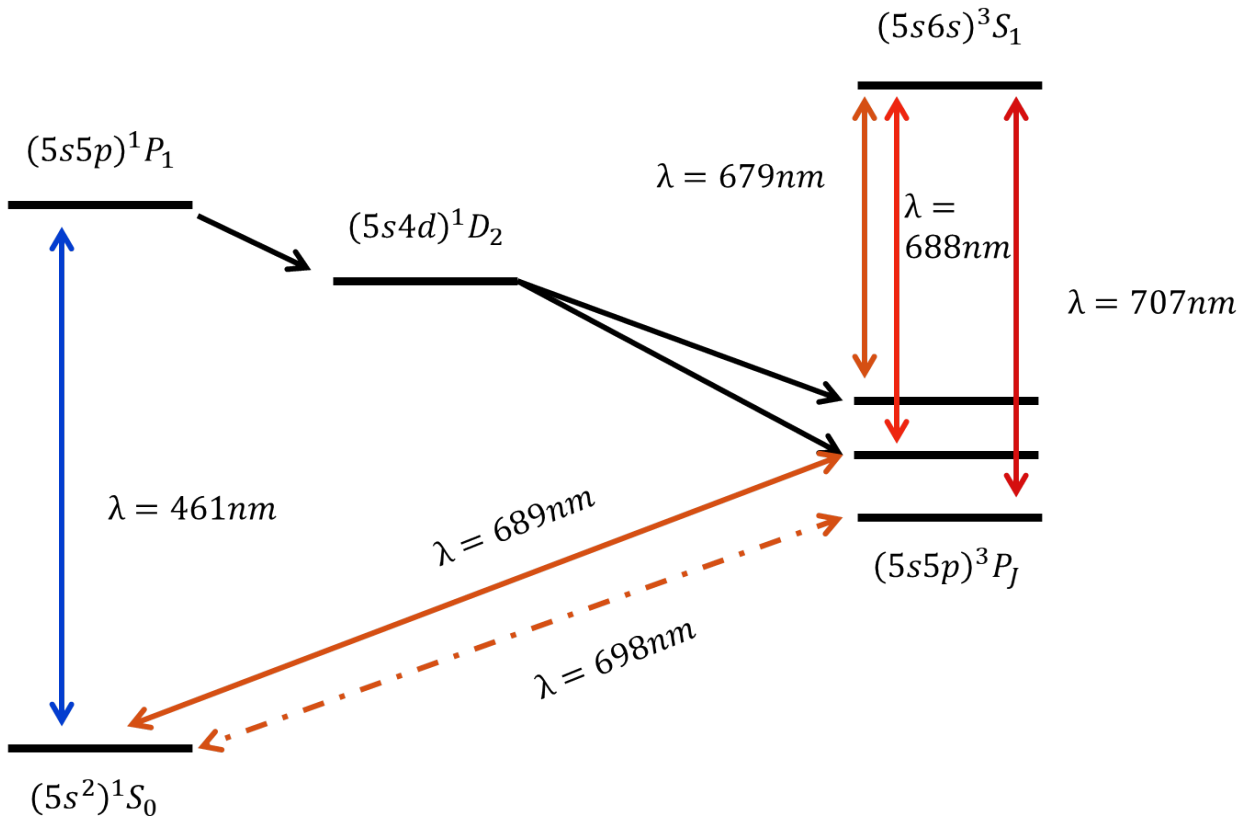


Figure 1.2: Some of the energy levels used in Strontium Optical Clocks[42]. The colour of the transitions roughly matches the visible colour, with some exaggeration of small differences. The full ground state is $1S^1\ 2S^2\ 2P^6\ 3S^2\ 3P^6\ 3d^{10}\ 4S^2\ 4P^6\ 5S^2$. The dotted line shows the $698nm$ clock transition. In ^{88}Sr a single photon transition is totally forbidden, however, for ^{87}Sr this is weakly allowed due to hyperfine mixing; both leading to an ultra-narrow transition, suitable for a time reference. The $461nm$ blue transition has a broad natural linewidth and is used for first stage cooling[42]. The $689nm$ transition is used for second stage cooling. After the clock measurement any remaining atoms in the $^3P_0$ state are pumped to the $^3S_1$ state, where they decay rapidly to the ground state[45].

back into the cavity, the angle of the ECDL grating forms an additional parameter[26]. A region of space free of mode hops is then located and the wavelength is held stable with linewidth of the order of $100kHz$. This laser is then split into 3 parts, one is passed to the atomic reference to calculate an offset against a fundamental constant, one is passed through an optical frequency comb to generate a highly stable microwave signal that can be read by analogue electronics and finally one is passed into a Ultra High Finesse Optical cavity. The laser is then locked to to this cavity using a Pound-Drever Hall technique[26] and by feeding an electronic signal back to the laser, the linewidth of the ECDL can be narrowed to $\sim 1Hz$[45][26].

1.2 Temperature Stabilisation

Many of the components required in high precision atomic physics require temperature stabilisation. As discussed in the first section of this chapter, often these experiments need to be portable, or have other constraints placed upon them. This means that if a solution is to be accepted, it will need to be small, modular and power efficient, with a wide range of applications. This project focused on the UHFC but for competeness a list of further applications is shown in Appendix C

1.2.1 Ultra High Finesse Cavity

Temperature stabilisation for the Ultra High Finesse Cavity used for the Sr clock laser in Birmingham was the main motivation for my project. The cavity requires a high degree of thermal stability to achieve its design specification of sub-hertz linewidth. Considering the that the resonant frequency for a Optical Cavity is given by[26],

$$f_{\text{Res}} = \frac{nc}{2L}, \tag{1.1}$$

and the thermal expansion of a material is given by,

$$\Delta L = L\alpha\Delta T, \tag{1.2}$$

where α is the thermal expansion coefficient. It is then possible to calculate the change in resonant frequency based on the thermal expansion of the cavity. Firstly, differentiating Equation 1.1 and substituting for f_0 gives,

$$\delta f = \left.\frac{\partial f}{\partial L}\right|_{f=f_0} \delta L \tag{1.3}$$

$$= -\frac{nc}{2L_0^2} \tag{1.4}$$

$$= -\frac{f_0}{L}\delta L, \tag{1.5}$$

where the 0 subscript indicates the design point. Taking a linear approximation and substituting for ΔL gives,

$$\Delta f \simeq -\frac{f_0}{L}\Delta L, \tag{1.6}$$

$$\simeq -\frac{f_0}{L}L\alpha\Delta T, \tag{1.7}$$

$$\simeq -\frac{c}{\lambda_0}\alpha\Delta T, \tag{1.8}$$

and is therefore independent of the length of the cavity. Letting $\Delta T = 1mK$, $\alpha = 2 \times 10^{-6}$ and $\lambda = 689nm$, which are typical values for this problem, gives an change of $8.7MHz$. This is considerable given such a small thermal change. Therefore, the cavity is made out of a mixture of *ULE*[1] glass and fused silica as a compromise between this effect and other sources of thermal noise[26]. Furthermore, the system is thermally and vibrationally isolated by placing it in a vacuum system with multiple layers of heat shields[26]. In addition, active compensation is provided by means of Peltier Elements and Thorlabs AD590 temperature sensors. This active compensation holds the system at the turning point for the expansion co-efficient of the *ULE* glass, this is around 12^oC[26]. This leaves a residual thermal coefficient of $0 \pm 30 \times 10^{-9}K^{-1}$ in the range $(5 \rightarrow 35)^oC$, with a minima at 12^0C[26]. Recalculating the Equation 1.8 with $\alpha = (0,\, 1,\, 30) \times 10^{-9}$ yields $\Delta f = (0,\, 0.435,\, 1.31)kHz$. It has previously been estimated that if the cavity is within $10mK$ of the minima, the frequency dependence $50Hz/mK$. Therefore, the active system must be able to hold the cavity at this minima, with no long term drift and also be able to detect sub $1mK$ fluctuations in temperature.

1.3 Detailed Aims

There are many commercial temperature controllers available for atomic physics and so it is important to determine why a new system is needed. These range from the temperature controllers provided with the diode lasers, such as the ThorLabs

[1] Ultra Low Expansion

Specification	TED200C	WTC3243	Iguana	Requirement	Desired
Multi-Level Control	✗	✗	✗	✓	✓
Accessible Algorithm	✗	✗	✗	✗	✓
Maximum Output Current (A)	2	2.2	1	2	3
Output Current Step Size	$10mA$	-	$119nA$	$2mA$	$40\mu A$
Physical Size (litres)	3.60	1.32[See a]	10.5 + 36.0[See b]	10.5	1.32
Non-PC Output	✓	✓	✗	✓	✓
Numeric Output	✓	✓	✗	✗	✓
Digital Output	✗	✗	✓	✓	✓
Data Logging	✗	✗	✓	✓	✓
Live Graphing	✗	✗	✗	✓	✓
Suitable for non-UHFC applications	✓	✓	✗	✗	✓
Theoretical Temperature Stability	1mK	0.9mK	1mK	1mK	0.1mK

Table 1.1: A table showing a comparison of the requirements for this system against currently available systems. As can be seen no system on the market is will meet the requirements for temperature stabilisation of the UHFC.

Table Remarks

[a] Once placed in a control box

[b] Physical Size + Size of interface PC, excluding screen, keyboard and mouse

TED200C system[50], to the commercial generic WTC3243[48], to the custom built Iguana Temperature Controller[26]. The primary motivation for my project was stabilisation of the UHFC. Due to the amount of heat shielding this system requires a very long integration time. As noted in Dr Johnson's PhD thesis, this is has been achieved with analogue systems, but it requires a very large bank of capacitors[26]. Since most of the ultra stable PI controllers available on the market are analogue these simply cannot achieve the integration times required. The Iguana system is a digital PID controller based on a micro-controller, it is currently used to stabilise the temperature of the UHFC, however, the system is large and written in assembly language, making editing the system difficult and inaccessible. Furthermore, the system is single level only and to avoid positive feedback across the UHFC's multiple heat shields a multi-level algorithm should be used. Lastly, the system requires a Linux Desktop for interface and programming.

It would be possible to re-program and adapt the current system, possibly reduce the size and raise the current limit, however, due to recent developments, it would be far easier to create a: new, modular, well documented system, written in an higher level language. If the code was made modular then any algorithm can be input without modifying the hardware. This leads to the list of requirements shown in comparison Table 1.1.

Chapter 2

Design of a Digital Temperature Controller

A digital temperature controller consists of five distinct sections:

1. the controller algorithm,
2. the processing unit,
3. the temperature detection system,
4. the temperature correction system and
5. the interface.

These are listed in the order in which they must be chosen. From requirements on a system it is possible to generate a list of requirements for the algorithm, which then places requirements on the processing unit, which then places requirements on the final three systems. In this chapter I will discuss the key parts of the design. Full circuit diagrams and detailed discussions about component selection are available in the Appendices. At several points I was able to make use of standard practises and protocols from control engineering and hobbyist electronics, everything else was developed from scratch.

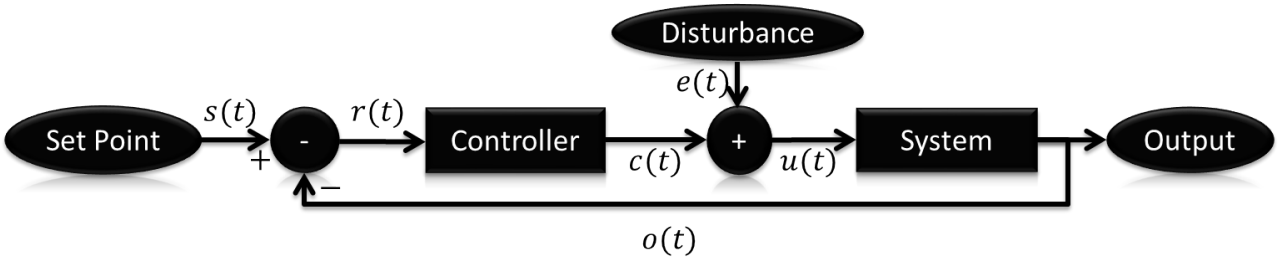


Figure 2.1: The block diagram for a generic feedback control system. Conventionally, block diagrams start with the set point, placed in the top left corner, then flow across the page towards the output[12]. Generic feedback systems such as this are often modelled in two blocks: the controller and the system[20]. Disturbances may occur at any point, however, it is easiest to model them as occurring just before the system[52].

2.1 Proportional, Integral and Differential Control

Many systems in laboratory environments require some form of active feedback in order to ensure that conditions in which the experiment are performed remain stable across successive iterations of the experiment. If the conditions differ or are not measured then reproducibility of the experiment is called into question. Furthermore, if they are not stable, then it is not possible to attribute an observed change in the system, to a measured quantity. These feedback systems generally fall within the domain of control engineering and some basic knowledge of control theory is often required in practical laboratory work.

A basic feedback control system is shown in Figure 2.1. In this report I will use t as a time variable, $s(t)$ to denote the set point, $r(t)$ to denote the difference between the set point and the response from the system, $c(t)$ to denote the output by the controller, $e(t)$ to denote any disturbances in the system, $u(t)$ to denote the response to the system and $o(t)$ to denote the system output.

The Proportional, Integral, Differential controller was first derived by Nicolas Minorsky as a proposed solution to the automatic steering of ships based on: the current position, the inertia of the craft and constant terms such as the windage[10]. Such controllers often offer the simplest solution to any control problem[3] and over 97% of regulatory controllers use PID feedback[28]. Due to these reasons and for improved generality, I have chosen to implement a PID controller algorithm for this temperature control system, however, I have implemented the algorithm entirely in code and making extensive use of the classes provided in the c++ language; thus the algorithm can easily be replaced if detailed knowledge of the

underlying processes are available.

2.1.1 PID Controllers in General

Deterministically the three terms are often described as representing: the current error, the past errors and a predication of future errors[30]. The past error or integral term accounts for constant losses, therefore bringing the value onto the set point exactly[12]; while the derivative, or future error, term is implemented for fast systems compared to their controller[30]. Although, as shown during war time efforts, control loops are best analysed in frequency space and not deterministically[11], therefore the following calculations will be in reciprocal space[1].

2.1.2 Proportional Control

In any controller, a higher ratio of controller gain, $A_c(f)$, to system gain, $A_s(f)$, leads to a better stability[2]. Thus it is important to set the proportional constant as high as possible without causing positive feedback.

Calculation of the Frequency Response of the System

The complete system will have phase and amplitude contributions from 4 elements, the measurement time, τ_m, leading to a phase shift ϕ_m, the calculation time, τ_c, with phase shift ϕ_c, a phase shift though any analogue components, (Op-Amps, Current Sources, etc.) ϕ_o and finally a phase shift in the though the controlled element, ϕ_s. The total phase shift ϕ_t is given by the sum of these. By taking ratios, the phase shift for a fixed delay, with no frequency dependence is,

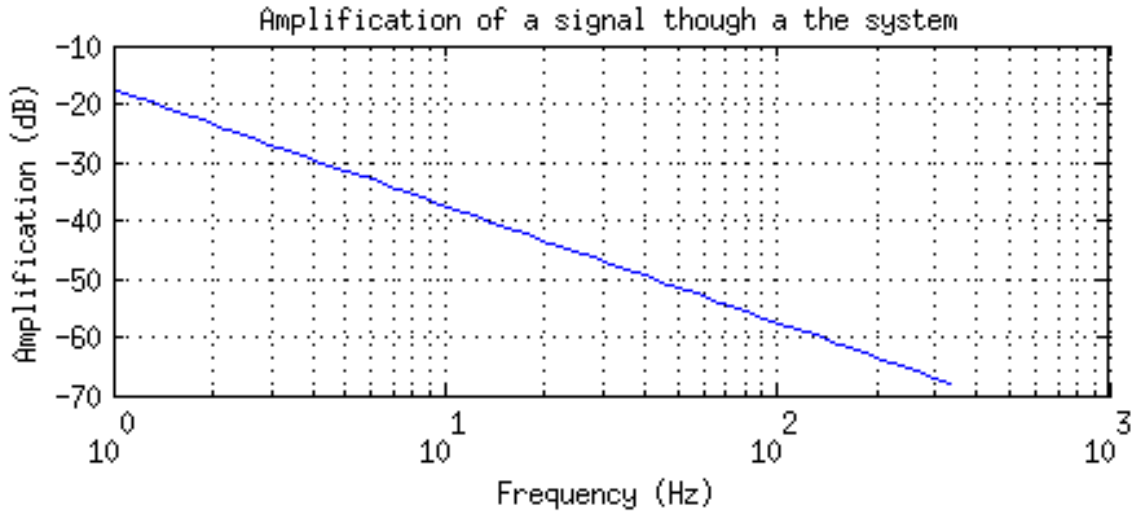
$$\phi(\tau, f) = \frac{2\pi\tau}{T} = 2\pi\tau f. \tag{2.1}$$

The calculation and measurement time can easily be estimated as $1ms$ for each. The phase shift through the remaining electronics can be estimated as a $10\mu s$ delay. For a proportional controller, the controller gain is simply the proportional constant, therefor the phase and amplitude contributions for the controller are known.

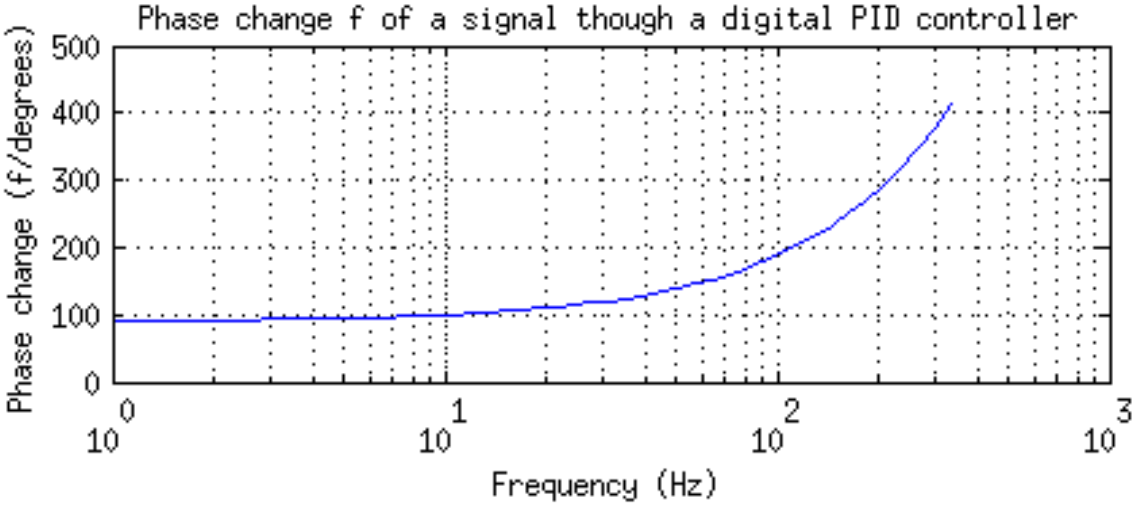
By assuming the controlled element thermalizes instantly and applying the first law of thermodynamics it is possible to

[1] See Modern Control Systems[20] and Instrumentation and Control Systems[12] for calculations in other spaces.

[2] A derivation for this is shown in section F.1



(a) An example of the amplification of a signal though the controlled element. This particular plot is assuming that Equation 2.2, holds true with $m = 40.65g$, $c = 0.897J/g$, $P(t) = 4.5e^{i\omega t}W$.



(b) An example of the phase change through the system. This particular plot is assumes Equation 2.2, with an additional $2.7ms$ delay for electronics.

Figure 2.2: Examples of phase changes in a controller system with active feedback.

show that the temperature of the system, T, at time, t, for a periodic input power $P(t)$, is given by[3],

$$T(t) = \underbrace{-i}_{\phi_s} \times \overbrace{\frac{1}{\omega mc}}^{A_s} \times P(t) + T_1, \tag{2.2}$$

where T_1 is the initial temperature of the system. Thus the phase response, $\phi_s = -i = -90^o$ and the amplitude response, $A_s = \frac{1}{\omega mc}$, can be deduced. If the system does not thermalize instantly but does so over some time then there will be an additional phase response. This additional response was correctly estimated to be $\sim 10s$ at the start of the project[4]. Graphs showing the predicted amplitude and phase response for this system are shown in Figure 2.2

Limitations on the Proportional Gain

The controller gain, A_c, should be as high as possible for best controller performance, but, the total gain, $A_c + A_s$, must also be less than or equal to unity when the total phase shift in the system is equal to π. Otherwise, unstable oscillations will occur due to positive feedback[52]. This places an upper limit on the proportional gain and so additional control is required[5].

The intention is to design a system for which the dominant phase response is the controlled element rather than the controller. Thus for the remainder of the design, the constraint that the controller shift should be less than 0.1% of the total shift is applied, this allows a total processing time of $\sim ms$.

2.1.3 Integral Control

It is likely that a thermal system there will be constant heat dissipation between the element and its surroundings, leading to steady state losses and a constant offset from the set point[12]. Since these are low frequency (LF), the gain may be improved, provided the high frequency (HF) response remains unchanged. Therefore, an amplification method must be chosen such that it has a high LF response but low HF response. In the case of integral control, assuming that the controller

[3] As shown in section F.2

[4] It is shown in subsection G.2.2 to be around $\sim 30s$.

[5] The optimum response for a system is given when the total phase shift is $\frac{2\pi}{3}$ at the unit loop gain point[52]. From this it is possible to determine a analytical value for the proportional constant, however, this requires a good analytical model of the system and if you have that, then there are better algorithms than PID. As such analytical PID constants are rarely used in control systems.

can output sufficient power, the gain limits are[52],

$$\lim_{f\to 0} g = \infty, \tag{2.3}$$

$$\lim_{f\to \infty} g = 0, \tag{2.4}$$

making integral control an excellent choice and allowing constant offsets can be eliminated without affecting system stability.

Limitations of Integral Control

The gain of an intergral controller is given by[52],

$$A_{ci} = \frac{\text{Signal Out}}{\text{Signal In}} = \frac{k_i \int_0^{t'=t} e^{i\omega t'} \mathrm{d}t'}{e^{i\omega t}} = \frac{k_i}{i\omega}. \tag{2.5}$$

It is clear that setting k_i as high as possible will decrease the convergence time, however, by setting $k_i = \frac{1}{\tau_i}$, the integrator cut off frequency can be determined as $2\pi k_i$. If this is too high then the phase change will increase and the system will become unstable[6].

2.1.4 Derivative Control

Correctly tuned derivative control will increase the transient performance of the system by introduce a phase lead of $\pi/2$ as can be shown by re-applying Equation 2.5[20]. The amplification is therefore, $k_d\omega$, leading to a differential cut of frequency, $f_d = \frac{1}{k_d}$. It is usual to select f_d equal to the frequency which results in a total phase lag of $2\pi/3$, $f_d = f_c$[52], extending the region of stability and compensating for integral control, making derivative control very useful for fast systems[12]

However, the transient response is often of little interest in thermal systems where the phase lag is dominated by the system and the set points are infrequently changed. Furthermore, in systems dominated by noise the derivative controller often increases the sensitivity of the system to noise and is generally considered to make the system less stable. This system was dominated by noise and so derivative control was not implemented.

[6]Setting $k_i = 0.1 f_c(\phi_t)$, where $f_c(\phi_t)$ is the frequency with total phase shift π, reduces the phase margin by less than 6^o and is a common choice[52]

2.2 System Design

To carry out the PID calculation a two part digital system was implemented. An Arduino was used to interface with Analogue To Digital and Digital to Analogue Converters and carry out the calculation. This was programmed in a modular fashion so the controller algorithm could be changed. The Arduino was chosen because it will have a roughly constant loop time of the order of a few ms, which is very suitable for temperature control. The Raspberry Pi was then chosen as an interface unit. This duel approach allows the controller to have the benefit of a full Operating System, while still retaining constant loop execution time[7].

2.3 The Temperature Detection System

The performance of a temperature control system is often characterised by the ability to accurately measure the temperature at a sufficiently quick rate. The base unit of temperature is defined as,

> "The kelvin, unit of thermodynamic temperature, is the fraction 1/273.16 of the thermodynamic temperature of the triple point of water".[25]

It is of course very difficult to use this as a method of measuring temperature and it makes far more sense to use a calibrated instrument[12]. It is very common practise to use either a $10k\Omega$ NTC[8] Thermistor or a Transistor based temperature transducer[52]. Both devices rely on the strong temperature dependence of semi-conductors[12]. I will only discuss the Thermistor in this report, but circuitry for the AD590 is shown in section D.4.

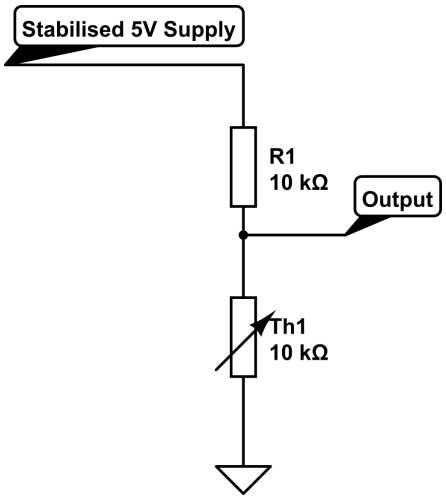
2.3.1 The $10k\Omega$ Thermistor

It is well known that the resistance of most metals have a weak temperature dependence, $\sim 3 \times 10^{-3}$[52]. Since the shape of the Fermi function is strongly temperature dependant[38] it is possible to develop non-linear temperature sensors using semi-conductors[12] where the resistance can be approximated by a cubic equation. The Thorlabs 10k thermistor is a common choice due to its low cost and $15s$ time constant when in air[49]. There are two possible circuits for determining a voltage from a resistance.

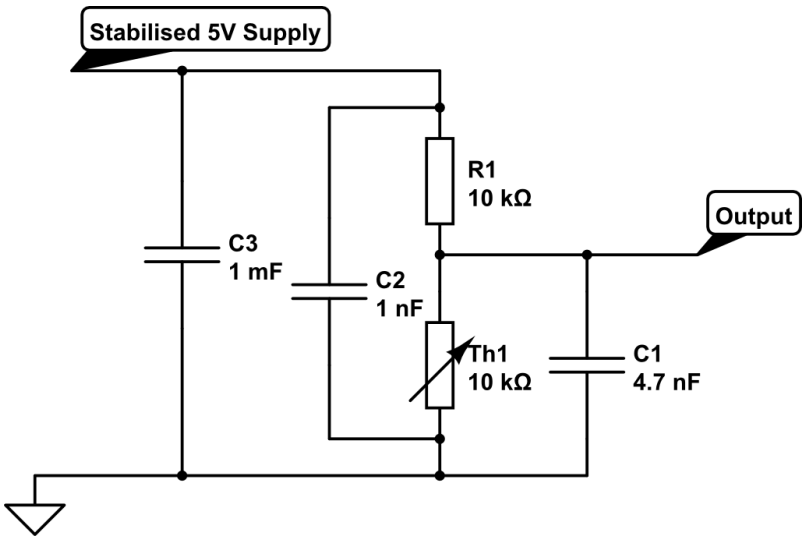
[7] The details of this analysis and selection are shown in Appendix D

[8] Negative Temprature Coefficiant

Potential Divider



(a) A basic potential divider used to generate a voltage from a known resistance. The voltage at the output depends on the resistance of Th_1. This is easy to implement and no calculations need to be performed. However, this circuit is very susceptible to noise on the 5V line.



(b) A potential divider with noise rejection. C_3 and C_2 reject noise from the power supply, C_3 can be placed anywhere and rejects noise caused by high current components, C_2 should be placed as close to the divider as possible and rejects residual power supply noise. C_1, R_1 and Th_1 all form a low pass filter. C_1 should be placed as close to the measuring device as possible as this rejects transmission noise. The cut off frequency is temperature dependant and for a Thorlabs 10K: $f_c^{50^oC} = 4.4KHz$ and $f_c^{0^oC} = 13kHz$ as derived in section F.5.

Circuit Diagram 2.1: Two methods of generating a voltage from a resistance using a potential divider. The first is simple and easy to implement, while susceptible to power supply noise. The second is more complex, but is immune to most noise.

The simplest and most common way to determine a voltage from a resistance is by using a Potential Divider as shown in Circuit Diagram 2.1a. The voltage at the output of such a circuit is given by[9],

$$V_o(V_i, Th_1) = V_i \frac{Th_1}{R_1 + Th_1}. \tag{2.6}$$

Solving this equation for Th_1 gives,

$$Th_1(V_o)) = \frac{R_1}{\frac{V_i}{V_o(T)} - 1}. \tag{2.7}$$

[9] As shown in section F.4

The resistance and hence temperature, can then be determined by floating point operations on the Arduino, provided $Th_1(T)$ is known. $Th_1(T)$ is normally given as a cubic equation on the data sheet[49].

Advantages The advantage of this system is its simplicity. As long as the $5V$ voltage is stable then there will be no noise in the system. In addition a reasonably wide range of voltages, $(1.1690 \rightarrow 2.5000 \rightarrow 3.6767)V$ is output for the temperature range $(0 \rightarrow 25 \rightarrow 50)^oC$.

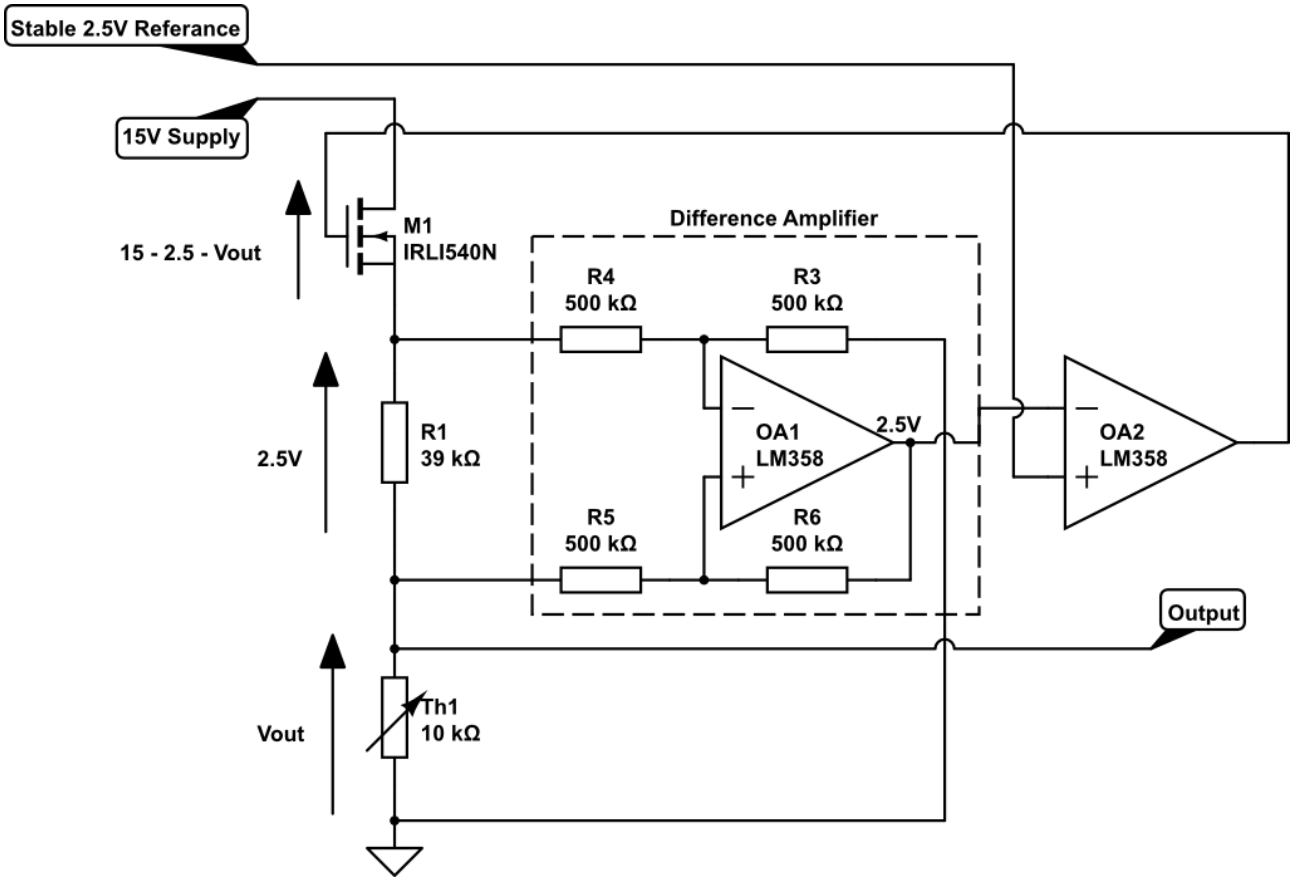
Disadvantages Unfortunately any noise on the 5V line will directly translate onto the output with the appropriate scaling factor, this can be avoided using Circuit 2.1b. Furthermore, any slow long term drifts in the 5V voltage will directly translate onto slow drifts in the apparent set point for the system. To overcome this either: the voltage of the $5V$ lines is measured, or a stable voltage supply used. Finally, while the $(1.1 \rightarrow 3.6)V$ range is good, it wasting 50% of the spectrum, or 1 bit.

2.3.2 The Current Controller Method

An alternative method of generating a $(0 \rightarrow 5)V$ voltage from a resistance is to pass a constant current through it. This can be achieved by measuring the voltage across a known resistance in series with the thermistor and using a feedback loop to control the current such that this voltage does not change. One method of doing this is to use a difference amplifier in series with a comparator as shown in Circuit 2.2.

Advantages This system can be configured so that the minimum temperature of the system leads to an output of 5V, therefore the entire voltage range is used. In addition, this circuit rejects noise both 15V supply noise and drift (the 2.5V supply may not drift).

Disadvantages Unfortunately, in practise implementation is difficult. Firstly, the resistance of a NTC thermistor will rise to $32k\Omega$ at 0^oC, allowing a maximum current of $150\mu A$, to avoid damaging the ADC. This is a very small current and fluctuations in this must be held constant. Secondly, any current drawn by Op Amps and the output must be negligible, therefore, the input impedance must be at least $0.5M\Omega$. For op-amps to function the current flowing in the feedback loop must be high compared to the input leakage current. Since typical input impedance's are $\sim M\Omega$, the leakage current may be comparable to the feedback current. Thirdly, a high frequency cut filter cannot be added at the output as it will affect



Circuit Diagram 2.2: The constant current resistance measurement. The difference amplifier determines the voltage across R1, this is then compared against a steady 2.5V signal. If the voltage across R1 is less than 2.5V, OA2 ramps up, lowing the resistance of the MOSFET and allowing more current to flow and vica versa. The voltage on the output is simply given by $V = ITh_1$. In this circuit $I = 64\mu A$.

Temprature (^oC)	Change in T (mK)	Visible with			Change in Voltage (V)
		15 Bits	16 Bits	24 Bits	
25	100	✓	✓	✓	5.40E-03
25	10	✓	✓	✓	5.40E-04
25	1	✗	✗	✓	5.40E-05
25	0.1	✗	✗	✓	5.4E-06
50	100	✓	✓	✓	4.60E-03
50	10	✓	✓	✓	4.60E-04
50	1	✗	✗	✓	4.60E-05
0	100	✓	✓	✓	4.60E-03
0	10	✓	✓	✓	4.60E-04
0	1	✗	✗	✓	4.60E-05

Table 2.1: A simplified table showing the visibility of small temperature changes across the range of operation of the device. The table shows that 15 Bit conversion is suitable across the whole temperature range for stabilities of up to $10mK$. After this point, there is little to be gained from 16 Bit and the move to 24 is therefore necessary.

the current flow. Finally, when the system was build unstable oscillations occurred due phase shifts in the system. All of these issues can be fixed, however, for the sake of one bit, it is far simpler to implement a better ADC. For very stable systems, it may be worthwhile if the 24 Bit ADC is too noisy.

2.3.3 Conversion to a Digital Signal

Once a voltage has been created, it is necessary measure this and produce a digital result. These devices are called Analogue To Digital Converters (ADC's). Assuming that the temperature detection method is a thermistor in a potential divider configuration, it is possible to determine the accuracy of 15, 16 and 24 bit converters. The actual relationship is very non-linear and depends on the absolute temperature as well as the change as shown in Figure 2.3. The resolution

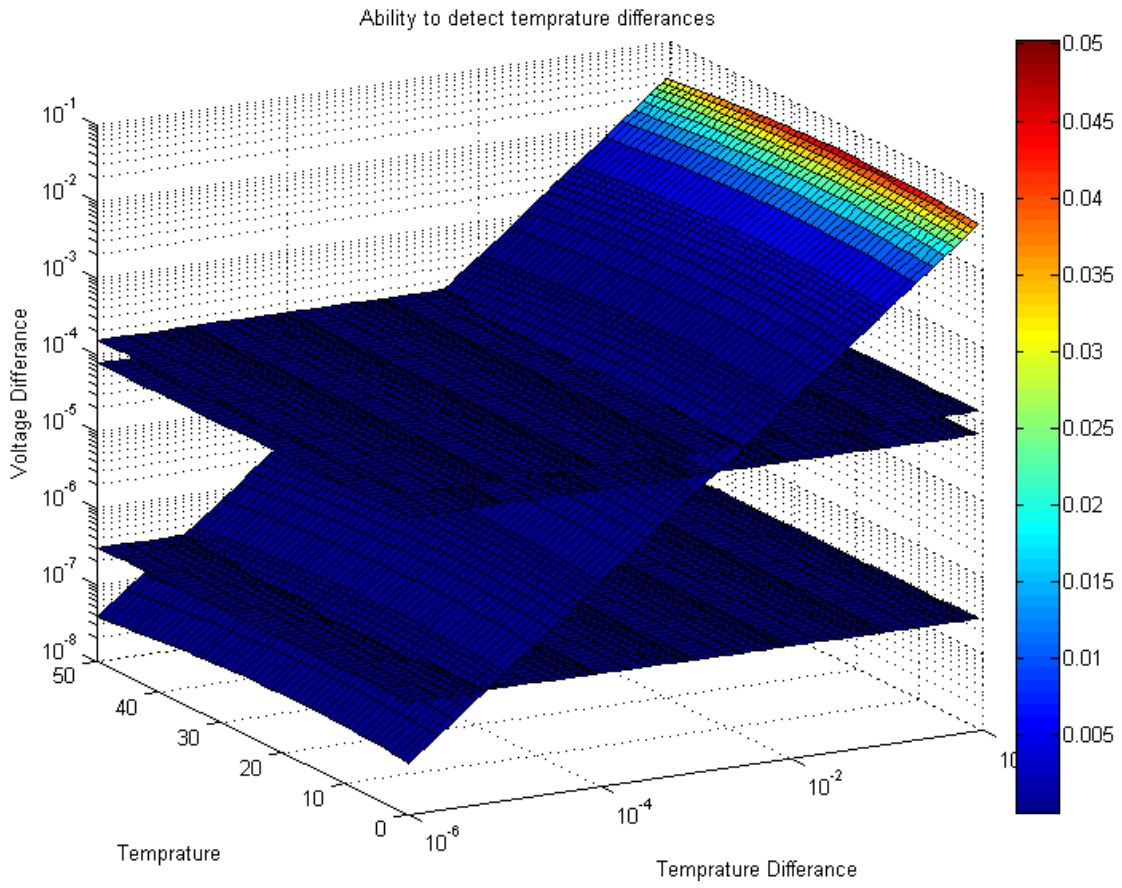


Figure 2.3: A semi logarithmic plotting the voltage difference between two temperatures for a range of temperatures for a thermistor in a potential divider configuration. Three horizontal planes are shown, going from top to bottom these are the resolution limit of a: 15 Bit, 16 Bit and 24 Bit ADC. The colour scale is linear and highlights the difference across the temperature range $(0 \rightarrow 50)^oC$, with the highest sensitivity when $R_1 = Th_1$. This graph highlights the general shape of the dependence, however, for numeric values please see Table 2.1

of the 15 Bit Converter is $150\mu V \approx 4mK$ resolution, the 16 Bit $76\mu V \approx 2mK$ resolution and 24 Bit $300nV \approx 5\mu K$ resolution[10]. To achieve an accuracy of $1mK$ it is clearly required to use a 24 Bit Analogue to Digital Converter. A full circuit diagram for this is available in Circuit Diagram D.1.

2.4 The Output

Any temperature controller must be able to both heat and cool the controlled element. Such devices are broadly split into two categories, refrigeration and thermoelectric. Refrigeration devices are often substantially more powerful and able to maintain higher temperature differences, however, these rely on adiabatic cooling[53] and the equipment required for this is often large and lacks fine control. The alternative is Thermoelectric cooling. This relies manipulating the fermi surface within a metal in order to transport heat[47]. The rate of heat flow from one side of the device to the other is then a linear function of the current flowing through the device[47][11].

2.4.1 Thermoelectric Cooling

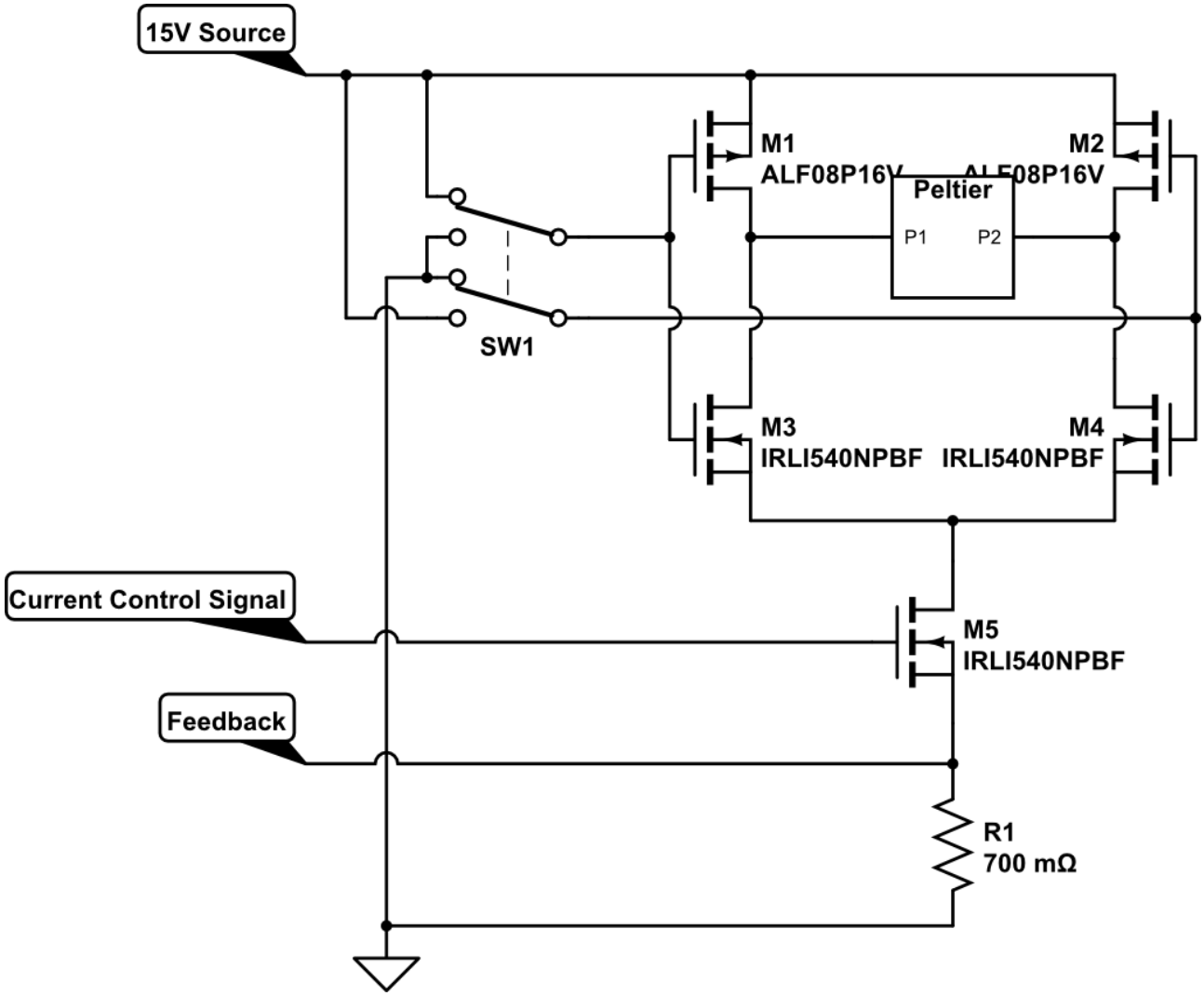
Thermoelectric cooling devices come in the form of small Peltier Elements, these thin devices transport heat from one side of the device to the other and are able to sustain temperature differences of up to 70^oC[40]. If the device is not properly heatsinked then heat energy (or a lack of) will build up on one side of the device and the temperature limit will be exceed. As a result these devices require a large heat sink, for longer periods of operation the heat sink must have suitable conductivity to a large heat bath, such as the air in the room. By manipulating the current it will is possible to control the heat flow on and off the controlled element such that the temperature is held constant.

2.4.2 Current Control For Thermoelectric Cooling

Controlling the heat flow requires an accurate current controller. This presented a technical challenge since the only example of such a controller was the WTC3243 and the work was closed source. In addition the voltage output by the DAC ranged between $(0 \rightarrow 5V)$, making the negative current direction difficult to implement. It was possible to centre the a new ground about 2.5V for the current controller, however, Peltier Elements can use up to $3A$[40] and this would

[10] Resolutions are quoted for $T = 25^oC$ for other resolutions please see Table 2.1.

[11] The derivation for this is not relevant to the temperature controller and isn't shown, however, a good description of the physics involved is provided in The Oxford Solid State Basics by Steven Simon[47]



Circuit Diagram 2.3: The Current Controller Circuit for the Peltier Junction. M1 to M4 form a direction changer for the flow of current through the Peltier Junction. When switch SW1 is its current position, the gate voltage on M1 and M3 is $15V$, hence the P-Type, M1, is not conducting, while, the N-Type, M3, is conducting, the gate voltage on M2 and M4 is 0V, hence, by the same logic M2 is conducting and M4 is not conducting. Thus, the current flows through M2, then the Peltier then M3. When the switch position is changed, the reverse argument holds, therefore the current flows through the Peltier Junction in the other direction. M5 and R1 are the current limiter, the current flowing though the circuit is determined by the voltage across R1, this is fed back and compared to the desired value, the current control signal is then adjusted based on this value. If this signal goes up, the resistance of M5 goes down, allowing more current to flow and vica versa.

cause an unavoidable waste of $P = IV = (2.5 \times 3)W$ of power. Instead it is possible to flip the polarity of the Peltier as shown in Circuit Diagram 2.3. The required feedback loop is simply a non-inverting amplifier and a comparator[12]. This comparator can compare the voltage between a $0 \rightarrow 5V$ signal and the voltage produced passing current though the resistor, hence allowing the current to be controlled by a DAC.

2.4.3 Digital to Analogue Converter

Common digital to analogue converters come in 8 and 16 bit variants. The Arduino Nano can imitate a 8 Bit Converter, using Pulse Width Modulation[13]. Unfortunately, the register required for PWM was in use and so a separate DAC was used. The minimum step size on the 8 bit converter would take 9.1 minuets to change the temperature of the block by $1mk$ assuming no losses, while on the 16 bit converter would take 38 hours. Therefore a 16 Bit DAC is required for very stable systems such as the UHFC, where the thermal constant is many hours. The AD5667R is a 16 Bit analogue to digital converter with internal 2.5V reference and a full scale output of 5V. Since a internal reference is required for the 24 Bit ADC and the cost of the chip similar to 8 bit versions, this can be used on all systems regardless of stability.

[12] Shown in Circuit Diagram D.2

[13] This is where the value changes between 0 and 5V very quickly so that on average the signal is the right value. A low pass filter can clean up the voltage

Chapter 3

Performance, Data and Analysis

In this chapter I will look at the performance of the temperature controller. Temperature controllers are limited by a number of factors, including their transient response, their response to noise, any sources of drift or shift and the accuracy with which they can modify the temperature. I will open the chapter with a discussion of real temperature noise and how this can be differentiated from other sources of noise. I will then look at the sources of drift and shift. Finally, I will demonstrate how temperature fluctuations cause unstable laser emission and how a temperature controller can fix this[1].

3.1 Sources of Noise to the Signal

Any real system suffers from noise and the temperature controller discussed in this report is no different. There are four types of noise in this controller: there is electrical noise introduced in the analogue electronics, there are digital rounding errors due to conducting mathematics in binary at fixed precision, there are real temperature fluctuations and then there are temperature fluctuations introduced by the system.

3.1.1 Real Temperature Fluctuations

The real temperature fluctuations must be calculated for each system. Of course due to the nature of the statistical physics, it is possible for very small systems[2] to have real noise fluctuations, however, it is impractical to connect this temperature

[1] The transient response may also be of interest and so is included in section G.2

[2] In this case small refers to systems where the number of atoms is much less than Avagdro's Number.

controller to such a system. In this section we will consider temperature fluctuations that are introduced by an unwanted system. The worst case scenario is a step change in the outside temperature. This could be achieved by a Air Conditioning unit turning on, or a piece of lab equipment starting up and blowing hot air over the controlled element.

Room temperature air can fluctuate by 1^oCs^{-1}, [See. 3] however, it will take a reasonable amount of energy to change the temperature of the block and the specific heat capacity of air is low. Thus the real temperature fluctuations on the block will be many orders of magnitude below this. By turning the controller off and graphing the temperature, fluctuations of around $(30-40)mK$ where observed with period of the order of on a timescale of $\sim 40s$ for an aluminium block of size $\left((3.0\times 5.0\times 1.0)\pm(0.1)^3\right) cm^3$. Of course, by insulating the system this can be reduced.

3.1.2 Electrical Noise

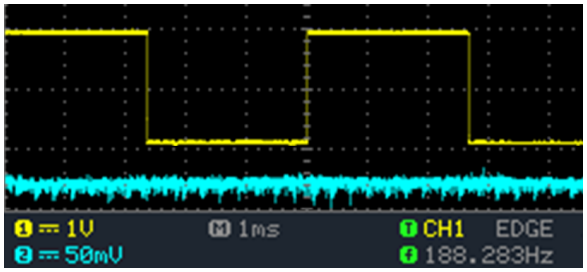
Often the controller and the controlled element cannot be collocated for practical reasons. This means that the wire connecting the two devices may act like an antenna and pick up interference. The AD590 temperature sensor overcomes this by outputting a current and Analog Devices have released an application note (AN-273) detailing how to overcome any residual noise[29].

Low Pass Filter

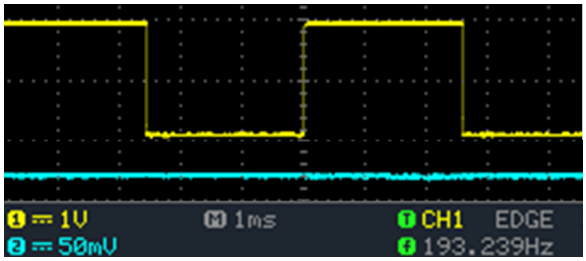
In the case of the thermistor the addition of a $4.7nF$ capacitor will reject high frequency noise, as discussed in section 2.3. $4.7nF$ is chosen because depending on the temperature this will reject noise above $\sim 4.4kHz$, since a loop of the control program takes $\sim 1ms$ this is well above the sample rate and so there will be no loss of information. In this case of the 16 Bit converter this leads to a 3.0 ± 0.9 reduction in noise amplitude on a breadboard as shown in Figure 3.1. Using Table 2.1 equates to $\sim 50mK$.

The 24 Bit converter does not feature an internal oscillator and so it must be driven by a $2MHz$ clock. This high frequency signal can couple between connections, in particular the input to the ADC as shown in Figure 3.1c. Furthermore, when the ADC register is read, SCLK can also couple, this leads to noise upwards of $580mV$ peak to peak as shown in Figure 3.2a. Fortunately the ADC reads the voltage at the quietest part of the signal (shown in Figure 3.2). Adding the low pass filter lead to a (6.3 ± 1.6) reduction in noise during the measurement period, T_C, as shown in Figure 3.2b. Using Table 2.1 this equates to $\sim 200mK$.

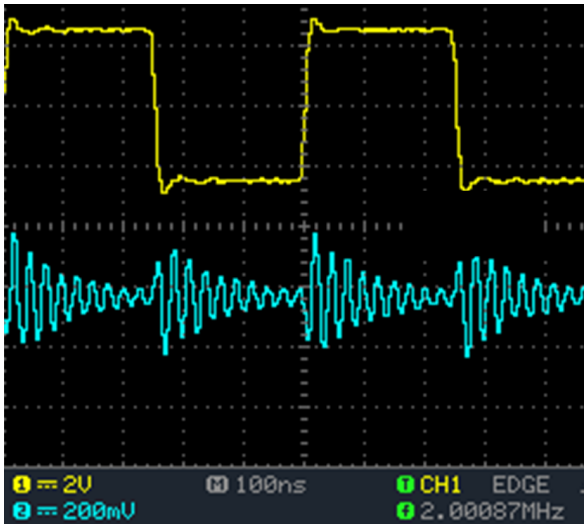
[3] Derivation shown in section F.3



(a) The noise on the input to the 16 Bit ADC without a low pass filter capacitor. Fluctuations of $(10 \pm 3)mV$ are observed, these are due to the cables picking up random interference at the mV level. Unfortunately due to technical limitations on the oscilloscope a better magnification was not available.

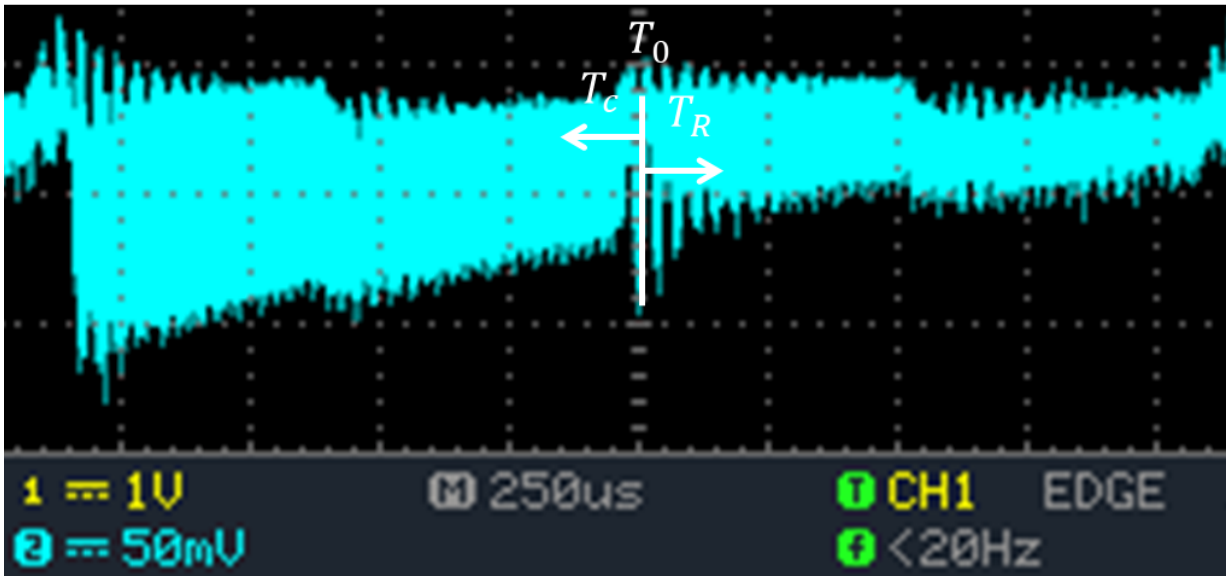


(b) The noise on the input to the 16 Bit ADC with a low pass filter capacitor. Fluctuations of $(3 \pm 3)mV$ are observed. This is a significant reduction from the Figure 3.1a and shows that the $4.7nF$ capacitor works as expected.

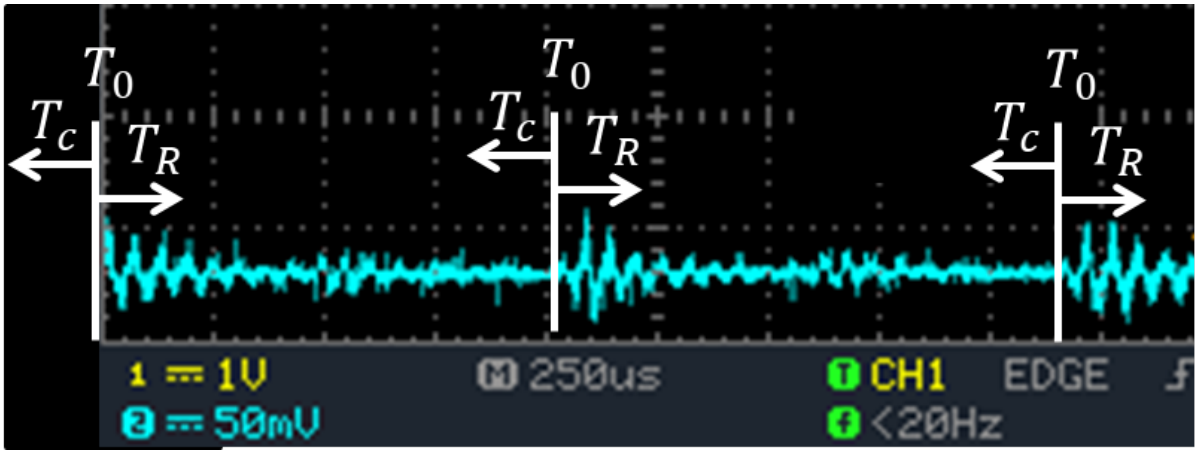


(c) To try and get a better magnification in Figures a) and b) the oscilloscope was capacitively coupled to the circuit. This trace is for the 24 Bit ADC, but as can be shown by contrasting with Figure 3.2b (which is in all other ways identical) there is a clear amplification of the noise. As such the scope had to be directly coupled to the circuit and the magnification was limited. Also visible in this trace is that there is some interference between the $2MHz$ clock and the input on the 24 bit ADC.

Figure 3.1: Scope traces showing the noise on the input to the analogue to digital converters. In Figure 3.1b and 3.1b the yellow trace is an output from the Arduino, changing state just before a measurement is taken, the blue trace is the input to the ADC. In Figure 3.1c, the yellow trace is the $2MHz$ clock to the 24 Bit ADC and the blue trace is the input to the 24 Bit ADC.



(a) The noise on the input to the 24 Bit ADC without a low pass filter. The fluctuations are a superposition of: random interference; interference from the $2MHz$ clock; and interference from the $SCLK$ and data out. T_R indicates the time in which the ADC is being read and there is a clear increase in noise due to this, T_C indicates the time in which the ADC is measuring the voltage. T_C is carried out in the quietest part of the signal, this would not be the case if the ADC was continuously read.



(b) The noise on the input to the 24 Bit ADC with a low pass filter, there is a clear reduction in noise amplitude from Figure 3.2a. T_C and T_R are defined as before. As before the voltage is measured in the quietest part of the signal, where $(8 \pm 2)mV$ fluctuations occur, elsewhere fluctuations up to $(50 \pm 5)mV$ occur.

Figure 3.2: Oscilloscope traces showing noise on the 24 bit converter.

Additional Noise Reduction Techniques

To further reduce the noise the temperature can be averaged over a N point moving average. N must be chosen so that it doesn't hide any real temperature fluctuations. The temperature fluctuations can be suppressed by insulating the system and limiting current through the Peltier element. Since the system knows when it is stable, it is possible to then limit the Peltier from changing by more than a specified amount, although this isn't implemented. The noise will be further reduced if the system is placed on a PCB with a ground plane. Averaging before the controller algorithm will decrease the response to noise at the cost of increasing the phase margin. Noise on the voltage lines is discussed in section G.1.

Other Sources of Noise

The output must be reasonably stable, however, in a thermal system, high frequency signals on the output will be immediately be averaged by the Peltier element since the response is slow. One such example is $100kHz$ noise through the Peltier element as shown in Figure 3.3. This signal is caused by positive feedback in the current controller circuit (Circuit Diagram D.2). Lastly, it is important to determine the noise due to rounding errors and the noise to to floating point arithmetic. Floating point is accurate to 8 decimal places[21] and the code is designed so that this shouldn't be exceeded. This gives a realistic temperature control limit of $1\mu K$. Finally, there is quantisation noise, but since the noise in larger than this, it should average out.

Electrical Noise Conclusions

It can be concluded that the 16-Bit system can be used without considering noise in too much detail, without averaging, provided the $4.7nF$ is installed. If 24 Bit precision is required, then the system should be insulated and an appropriate average time calculated.

3.1.3 Possible Sources of Long Term Drift and Shift

The set point on the temperature controller is digital. This means it simply cannot drift, however, if the analogue set-point is enabled then there may be some drift, this is why it is important to disable the analogue set point when it is no longer required. In addition the analogue section of the controller may experience drift and shift. The analogue output section

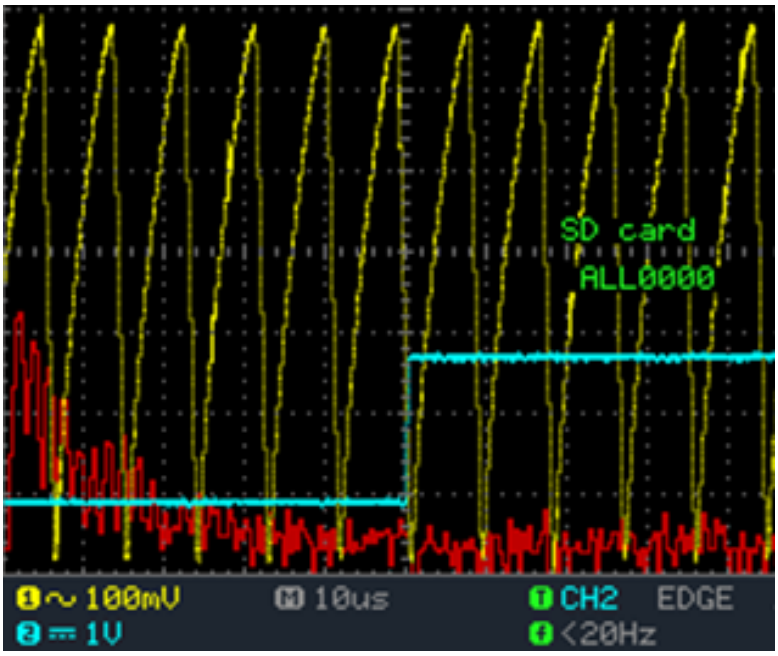


Figure 3.3: Noise through the Peltier Junction. In yellow is the signal passing though the Peltier junction, with any DC components removed, there are $700mV$ oscillations at $100kHz$; in blue is the loop signal from the Arduino, showing that this signal is independent of any other high frequency oscillations and in red is a Fourier transform of the yellow signal.

may drift or shift, however, this is of little importance since the controller the PID algorithm will account for this by design. All that remains if the shift on the temperature sensor.

Thermistor based Temperature Sensor

Connector As the temperature in the room varies, so will the resistance of the cables connecting the thermistor. These cables are generally around 50Ω, with temperature co-efficients of around $\sim \times 10^{-3}$[52], therefore the change will be 0.05 in $10^3 \rightarrow \sim 0.01mK$.

Resistor The resistor used in the potential divider will also experience a temperature shift, using specialist components this can be reduced to $230\mu K$ per degree ambient change. In an analogue PID controller, every component would shift like this and so the effect would become compounded, causing macroscopic fluctuations and set point drifting. In this digital system, this is the only component that experiences a shift. To avoid this the controller can either, regulate its own temperature to within a few mK, or calculate the shift and correct for it.

Voltage Fluctuations Finally, long term voltage fluctuations on the $5V$ line will appear directly on the temperature measurement. By connecting the 5V line to an ADC it is possible to monitor these voltage fluctuations and account for them. The 16 Bit ADC is used for this, therefore, the temperature cannot drift by any more than $2mK$. The same applies to the 24 Bit 2.5V line. If a better stability is required, then a more stable power supply must be used.

AD590 Temperature Sensor

In the AD590, only the resistance of the resistor will shift, as before this can be calculated and accounted for as before.

3.1.4 Stability given a perfect system

Given a well insulated system, self monitoring, stable voltage supply and correct PID constants, then the system will be limited by the length of the average. The observed stability is $(3 \pm 1)mK$, with a $250ms$ average time, extrapolating to a $1s$ average time gives a theoretical stability of $3mk \times \sqrt{\frac{.250}{1}} = 1.5mK$ and for a 10 second average $(3 \pm 1)mk \times \sqrt{\frac{.250}{1}} = (0.47 \pm 0.16)mK$. Hence sub micro kelvin stability is possible as long as the time constant of the system is well above $10s$.

3.1.5 Observed Stability

A perfect system to asses the stability does not exist and so a lower bound on the performance was calculated using a worst case system. Such a stability is achieved when the controlled element is small, with a large degree of airflow. An aluminium laser diode mount, of dimensions $\left((3.0 \times 5.0 \times 1.0) \pm (0.1)^3\right) cm^3$, with a fan blowing air across it, was used. The temperature was set to 25^oC, which was $(4 \pm 2)^oC$ above the measured room temperature. The PID constants were optimised as shown by the critical damping of the system in Figure 3.4. I then achieved a stability of $25^oC \pm 3mK$ as shown in Figure 3.5.

3.2 Application Example - A Laser Diode System

As discussed previously in section C.2, in most atomic physics experiments it is of crucial importance to keep the linewidth of the laser systems as low as possible. There of few experiments where it is of more importance than the Optical Clock, where linewidths of $1.37Hz$ or less are often used[26]. In this section, for the ECDL system in use for the $689nm$

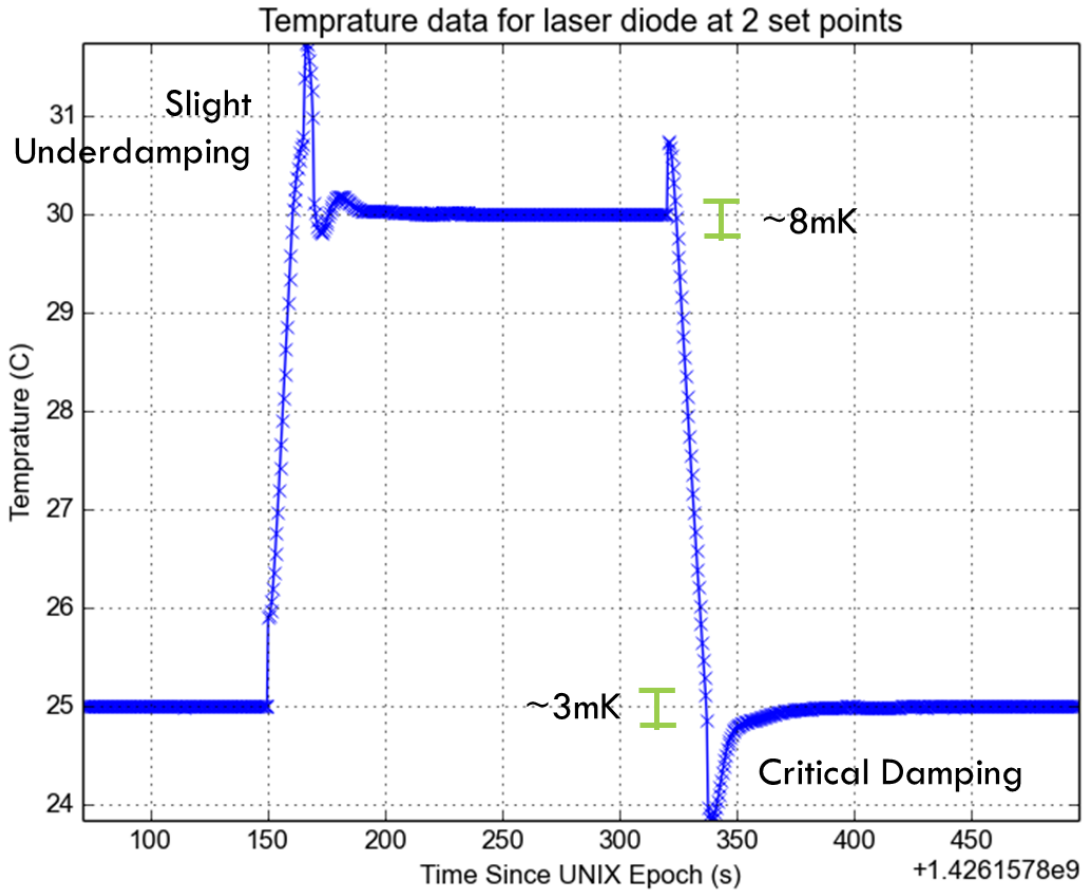


Figure 3.4: Temperature data for an open laser diode system when set at 2 different set-points. This graph shows the non-linearity of the temperature control system. The two pieces of evidence are the under damping and larger oscillations when at the higher set-point. Because the system is non-linear, different PI parameters are required at each temperature for the best accuracy. This deviation is only small and so for most cases the same parameters can be used, provided that the temperature is not changed to drastically.

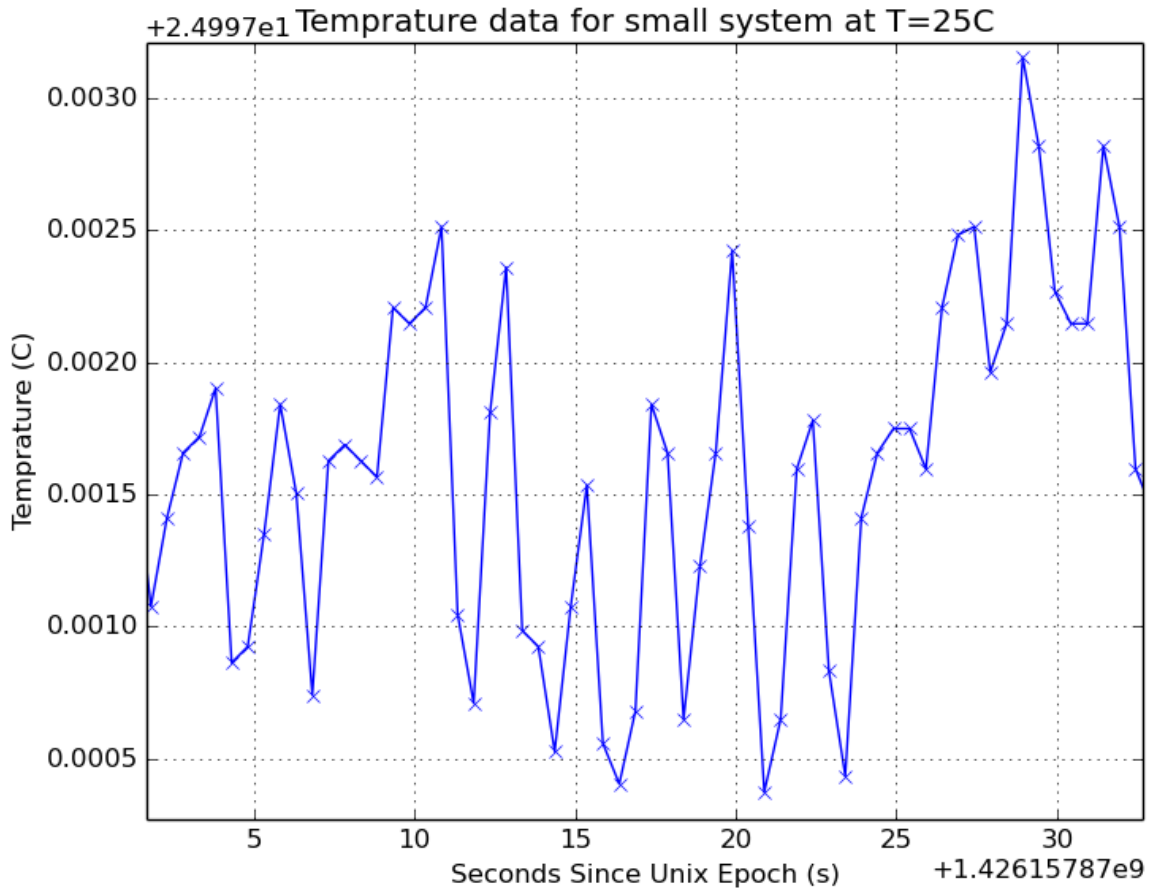


Figure 3.5: Temperature data for an open laser diode system when set at once the system is stable. This graph shows the stability of the system even for a small element with no insulation and a reasonably short averaging time of $250ms$. The oscillations that remain are a mixture of real thermal fluctuations caused by changes in the air temperature passing the device and electrical noise. For a larger, more insulated system, the thermal response time for 1mK change would be longer and therefore a longer average could be used, making the system less responsive to electrical noise and hence more stable.

re-pumper lasers in a Neutral Strontium Optical Clock, I will show how the frequency stability of this mode can be dramatically increased with appropriate temperature stabilisation[4].

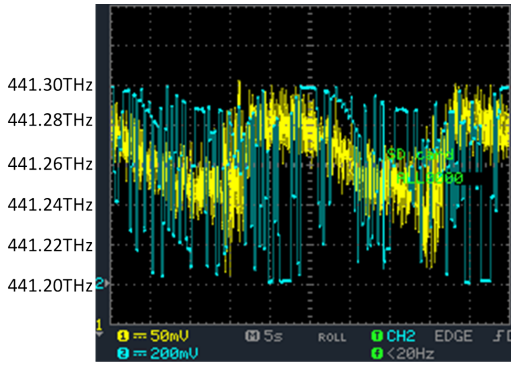
3.2.1 Frequency Stability

Temperature normally changes very slowly and so the effect is a slow drift in the laser frequency occasionally passing though regions of mode hops. In an hour, ambient room temperature changes of 2^oC are not uncommon. By modulating the set-point on the controller, the ECDL temperature was modulated and a temperature dependence found. Care was taken to use sinusoidal variation and modulate through a small amount, as step changes in temperature can kill a laser diode. The effect of drifting temperature is shown in Figure 3.6a. When an analogue temperature controller is connected, the frequency is then held constant regardless of room temperature changes as shown in Figure 3.6b. However, with the analogue controller, some instability remains and this was not present when the system was connected to the temperature controller presented in this report as shown in Figure 3.6c, although it must be stressed that the effect may only be slight since current noise is of a similar magnitude to the temperature noise when the WTC3243 is connected.

3.3 Review of the System

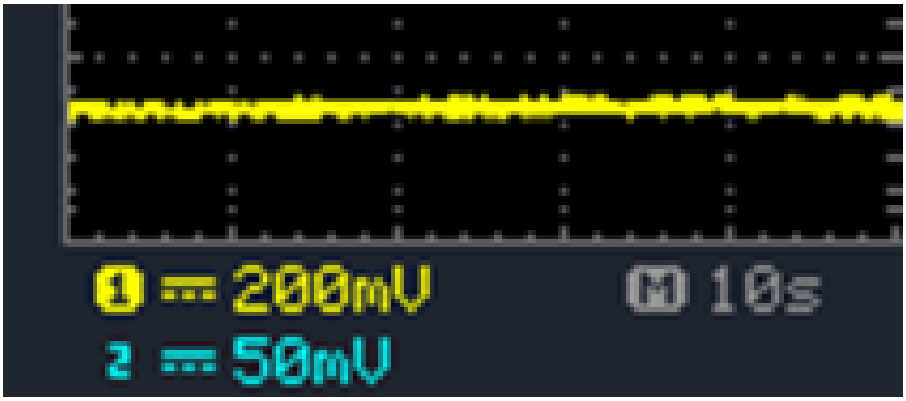
In section 1.3 I discussed the key requirements for a new temperature controller, including: multi-level control, outputs of several amps, stability at $1mK$ and output both without and without a PC. While not implemented, multi-level control is available due to the modular nature of both the code and the hardware. The use of high level languages such as C++ make the code easy to edit and the use of hobbyist micro-controllers make the system easy to use. The system met all of the design goals, as shown in Table 3.1. The project is complete in the sense that a generic temperature controller has been designed that performs as required. What remains is to implement the system onto the UHFC and any other projects that require it, unfortunately, designing the current regulator and getting below $10mK$ stability took longer than anticipated and so this implementation was not completed in the required time-frame. Overall, the project was successful and the temperature controller met its aims.

[4] A theoretical explanation for this behaviour is covered in section F.6



(a) Set-point modulation at $t = (38 \pm 1)s$, $(2 \pm 0.2)^{o}C$ peak to temperature change. The yellow line shows the shape of the temperature change, the blue line shows the frequency of the laser diode. If the laser frequency exceeds $441.29THz$ it loops though from the bottom. It can clearly be seen that temperature fluctuations affect the laser emission as predicted.

(b) Here the ECDL is held at a constant temperature using a WTC3243 chip, as can be seen the frequency is very stable. There are high frequency changes due to current instability, but these are superimposed upon very small, longer term oscillations. These oscillations are due to the instability in the analogue WTC3243 chip.



(c) Here the ECDL is held at a constant temperature using a the controller presented in this report. There are still high frequency changes due to current instability, but the small, longer term oscillations are no longer visible, even over the longer measurement period.

Figure 3.6: Oscilloscope traces showing the effect of temperature control on a sophisticated ECDL

Specification	Iguana	Requirement	Desired	Achieved
Multi-Level Control	✗	✓	✓	✓[See a]
Accessible Algorithm	✗	✗	✓	✓
Maximum Output Current (A)	1	2	3	2 (8)[See b]
Output Current Step Size	$119nA$	$2mA$	$40\mu A$	$40\mu A$[See c]
Physical Size (litres)	10.5 + 36.0[See d]	10.5	1.32	1.32[See e]
Non-PC Output	✗	✓	✓	✓
Numeric Output	✗	✗	✓	✗
Digital Output	✓	✓	✓	✓
Data Logging	✓	✓	✓	✓
Live Graphing	✗	✓	✓	✓
Suitable for non-UHFC applications	✗	✗	✓	✓
Theoretical Temperature Stability	1mK	1mK	0.1mK	0.47mK[See f]

Table 3.1: A table comparing actual performance against the aims. It clearly shows that the system meets all the requirements and matches most of the desired goals.

Table Remarks

[a] While not implemented, it has been demonstrated that this is possible.

[b] $2A$ was demonstrated, but up to $8A$ is theoretically possible. Careful heat-sinking and higher supply voltages will be require if this is desired. It may also be advisable to drive the P-Type MOSFETs with a gate voltage higher than the supply for the current regulator.

[c] Max Current / $2^{\text{No Bits on DAC}}$

[d] Physical Size + Size of interface PC, excluding screen, keyboard and mouse

[e] Predicted value once placed onto a printed circuit board

[f] Theoretical, $3mK$ demonstrated for an open system

Chapter 4

Conclusions

I started this report by highlighting the need for development in the area of Atomic Physics, I then took the specific case of the Optical Clock and showed the need for its development. I then demonstrated the need for accurate temperature control for the Ultra High Finesse cavity and explained the motivation for the project. Finally, I closed Chapter 1 by laying out the requirements for a temperature controller and comparing against both commercial and non-commercial systems.

I introduced Chapter 2, with a discussion of the generic PID controller paying particular attention to its suitability for temperature control. I estimated the required response times for my system and used this to determine requirements for the electronics I used. I then showed and discussed the key circuitry that was used.

In Chapter 3, I discussed some of the limitations of the system. I discussed real temperature noise and how a differentiation was made between that and electrical noise. I showed that the system had a stability of $(3 \pm 1)mK$ in an open environment and predicted a stability of $(0.47 \pm 0.16)mK$ for a closed system. It was then demonstrated that the laser emission frequency drifts over time due to temperature fluctuations. I showed how a digital temperature controller can be implemented to overcomes these fluctuations. I closed Chapter 3 with a comparison of the system against the project aims and discussed the potential for further work. Finally, in this chapter I have reviewed the report and highlighted my key result of $(3 \pm 1)mK$ stability for an open system an $(0.47 \pm 0.16)mK$ stability for a closed system.

Appendix A

Bibliography

[1] I. MacIver C. Bradshaw L. Roberts J. Lacy L. Maciw B. Adams E. Galkowska A. Morgan A. Jones, S. Cooper. Investigation into magnetic shielding using metglas®, group studies - atom interferometry, final report. School of Physics and Astronomy, University of Birmingham, mar 2014.

[2] Jun Ye E. Peik P.O. Schmidt Andrew D. Ludlow, Martin M. Boyd. Optical atomic clocks. *Cornell University*, jul 2014.

[3] Kiam Heong Ang, G. Chong, and Yun Li. Pid control system analysis, design, and technology. *Control Systems Technology, IEEE Transactions on*, 13(4):559–576, July 2005.

[4] Y. Arakawa and H. Sakaki. Multidimensional quantum well laser and temperature dependence of its threshold current. *Applied Physics Letters*, 40(11), 1982.

[5] Arduino, `http://arduino.cc/en/Main/arduinoBoardNano`. *Arduino Nano*, oct 2014. Date Retrieved: 10th March 2015.

[6] Arduino. Introduction. Online, `http://www.arduino.cc/en/Guide/Introduction`, mar 2015. Retrieved 10/03/2015.

[7] Atmel, 1600 Technology Drive, San Jose, California 95110, United States. *ATMEL 8-Bit Microcontroller with 4/8/16/32KBytes In-System Programmable Flash Datasheet Summary*, oct 2014.

[8] J. R. Williams S. L. Campbell M. Bishof X. Zhang W. Zhang S.L. Bromley J. Ye B. J. Bloom, T. L. Nicholson. An optical lattice clock with accuracy and stability at the 10^{-18} level. *Nature*, 506(7486):0028–0836, feb 2014.

[9] Brett Beauregard. *Arduino PID Library*. `mailto:br3ttb@gmail.com`, `http://playground.arduino.cc/Code/PIDLibrary`, dec 2012.

[10] S. Bennett. Nicholas minorsky and the automatic steering of ships. *Control Systems Magazine, IEEE*, 4(4):10–15, November 1984.

[11] S. Bennett. A brief history of automatic control. *Control Systems, IEEE*, 16(3):17–25, Jun 1996.

[12] W Bolton. *Instrumentation and control systems*. Newnes, Oxford Burlington, MA, 2004.

[13] M. W. Brimicombe. *OCR electronics for A2*. Hodder Arnold, London, 2009.

[14] James John Brophy. *Basic electronics for scientists*. McGraw-Hill, 1983.

[15] P. Cheiney, O. Carraz, D. Bartoszek-Bober, S. Faure, F. Vermersch, C. M. Fabre, G. L. Gattobigio, T. Lahaye, D. Gury-Odelin, and R. Mathevet. A zeeman slower design with permanent magnets in a halbach configuration. *Review of Scientific Instruments*, 82(6):–, 2011.

[16] Andrew J Daley. Condensed-matter physics: Rotating molecules as quantum magnets. *Nature*, 501(7468):497–498, 2013.

[17] davekw7x. *Arduino Forum, Using Arduino, Programming Questions, Clock*. `http://forum.arduino.cc/index.php?topic=62964.0-`, jun 2011.

[18] M. de Angelis, M.C. Angonin, Q. Beaufils, Ch. Becker, A. Bertoldi, K. Bongs, T. Bourdel, P. Bouyer, V. Boyer, S. Drscher, H. Duncker, W. Ertmer, T. Fernholz, T.M. Fromhold, W. Herr, P. Krger, Ch. Krbis, C.J. Mellor, F. Pereira Dos Santos, A. Peters, N. Poli, M. Popp, M. Prevedelli, E.M. Rasel, J. Rudolph, F. Schreck, K. Sengstock, F. Sorrentino, S. Stellmer, G.M. Tino, T. Valenzuela, T.J. Wendrich, A. Wicht, P. Windpassinger, and P. Wolf. isense: A portable ultracold-atom-based gravimeter. *Procedia Computer Science*, 7(0):334 – 336, 2011. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

[19] Analogue Devices. Ad590 2-terminal ic temperature transducer. Datasheet Rev G., One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A, http://www.analog.com/media/en/technical-documentation/data-sheets/AD590.pdf, oct 2013.

[20] Richard Dorf. *Modern control systems*. Addison-Wesley, Menlo Park, Calif, 1998.

[21] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[22] J. Goldwin, S. B. Papp, B. DeMarco, and D. S. Jin. Two-species magneto-optical trap with ^{40}K and ^{87}Rb. *Phys. Rev. A*, 65:021402, Jan 2002.

[23] Eugene Hecht. *Optics*. Addison Wesly, fourth edition, 2001.

[24] N. Hinkley, J. A. Sherman, N. B. Phillips, M. Schioppo, N. D. Lemke, K. Beloy, M. Pizzocaro, C. W. Oates, and A. D. Ludlow. An atomic clock with 10^{18} instability. *Science*, 341(6151):1215–1218, 2013.

[25] Bureau international des poids et mesures. *The International System of Units*. Bureau international des poids et mesures, eighth edition, 2006.

[26] Steven Johnson. *Narrow linewidth lasers for use with neutral strontium as a frequency standard*. Doctor of philosophy, School of Physics and Astronomy, University of Birmingham, December 2013.

[27] Aaron Jones. Project report : Improving laser linewidth using a grating stabilised external cavity diode laser. 3rd Year Photonics Lab Report, Aaron Jones, University of Birmingham, jan 2014.

[28] Richard M. Murray Karl Johan Astrm. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2010.

[29] Paul Klonowski. Use of the ad590 temperature transducer in a remote sensing application. Application Note AN-273, Analog Devices, One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, USA, 2015. http://www.analog.com/media/en/technical-documentation/application-notes/44190261583605578615358315 6an273.pdf.

[30] C. Knospe. Pid control. *Control Systems, IEEE*, 26(1):30–31, Feb 2006.

[31] Bjorn Ole Kock. *Magneto-Optical Trapping of Strontium for use as a Mobile Frequency Reference*. Doctor of philosophy, School of Physics and Astronomy, University of Birmingham, apr 2013.

[32] P.-L. Liu, L. E. Fencil, J.-S. Ko, I. P. Kaminow, T. P. Lee, and C. A. Burrus. Amplitude fluctuations and photon statistics of InGaAsP injection lasers. *IEEE Journal of Quantum Electronics*, 19:1348–1351, September 1983.

[33] J. Lodewyck. An even better atomic clock. *Spectrum, IEEE*, 51(10):42–64, October 2014.

[34] Ryoichi Higashi Hidetoshi Katori Masao Takamoto, Feng-Lei Hong. An optical lattice clock. *Nature*, 435(7040):321 – 324, may 2005.

[35] Takuya Ohkubo Hidetoshi Katori Masao Takamoto, Manoj Das. Cryogenic optical lattice clocks. *Nature Photonics*, 9(3):185–189, feb 2015. `http://dx.doi.org/10.1038/nphoton.2015.5`.

[36] Nobelprize.org: Nobel Media. The nobel prize in physics 1956. `http://www.nobelprize.org/nobel_prizes/physics/laureates/1956/`, Date Accessed: 21st Feb 2015.

[37] Gerard Milburn. *Quantum Technology*. Allen & Unwin Science, 9 Atchison Stret, St Leonards, NSW 1590, Australia, first edition, 1996.

[38] Andre Moliton. *Solid-State Physics for Electronics (ISTE)*. Wiley-ISTE, 2009.

[39] E.A. Parr. 1 - computers and industrial control. In E.A. Parr, editor, *Programmable Controllers (Third Edition)*, pages 1 – 32. Newnes, Oxford, third edition edition, 2003.

[40] Peltier-Technik. *Produkte - Peltier-Elemente*. Peltron GmbH, `http://www.peltron.de/peltierelemente_standard.htm`. [Date Retrieved 25/03/2015].

[41] A Peters, K Y Chung, and S Chu. High-precision gravity measurements using atom interferometry. *Metrologia*, 38(1):25, 2001.

[42] S. Bize U. Sterr A. Grlitz Ch. Lisdat M. Schioppo N. Poli A. Nevsky C. Salomon S. Schiller, G. M. Tino. The space optical clocks (soc) project. Final report, Heinrich-Heine-Universitt Dsseldorf, Dsseldorf, Germany, jan 2012.

[43] M. S. Safronova, S. G. Porsev, U. I. Safronova, M. G. Kozlov, and Charles W. Clark. Blackbody-radiation shift in the sr optical atomic clock. *Phys. Rev. A*, 87:012509, Jan 2013.

[44] D Sands. *Diode Lasers*. Institute of Physics, first edition, 2004.

[45] S. Schiller, A. Gorlitz, A. Nevsky, S. Alighanbari, S. Vasilyev, C. Abou-Jaoudeh, G. Mura, T. Franzen, U. Sterr, S. Falke, C. Lisdat, E. Rasel, A. Kulosa, S. Bize, J. Lodewyck, G.M. Tino, N. Poli, M. Schioppo, K. Bongs, Y. Singh, P. Gill, G. Barwood, Y. Ovchinnikov, J. Stuhler, W. Kaenders, C. Braxmaier, R. Holzwarth, A. Donati, S. Lecomte, D. Calonico, and F. Levi. The space optical clocks project: Development of high-performance transportable and breadboard optical clocks and advanced subsystems. In *European Frequency and Time Forum (EFTF), 2012*, pages 412–418, April 2012.

[46] Siegman. *Lasers*. University Science Books, 1983.

[47] Steven Simon. *The Oxford Solid State Basics*. Oxford University Press, Oxford, 2013.

[48] Team Wavelenght, 51 Evergreen Drive, Bozeman, Montana, 59771, USA. *Datasheet and Operating Guide WTC3243 & WTC3293 Ultrastable TEC Controller &Evaluation Board*, n edition, aug 2014. `http://www.teamwavelength.com/downloads/datasheets/wtc3243.pdf`.

[49] Thorlabs. Th10k thermistor. Datasheet, `http://www.thorlabs.de/thorproduct.cfm?partnumber=TH10K`, oct 2012.

[50] ThorLabs. *Thermoelectric Temperature Controller - TED200C Operation Manual*, 3.2 edition, jan 2014. `http://www.thorlabs.de/thorcat/15900/TED200C-Manual.pdf`.

[51] G.M. Tino, L. Cacciapuoti, K. Bongs, Ch.J. Bord, P. Bouyer, H. Dittus, W. Ertmer, A. Grlitz, M. Inguscio, A. Landragin, P. Lemonde, C. Lammerzahl, A. Peters, E. Rasel, J. Reichel, C. Salomon, S. Schiller, W. Schleich, K. Sengstock, U. Sterr, and M. Wilkens. Atom interferometers and optical atomic clocks: New quantum sensors for fundamental physics experiments in space. *Nuclear Physics B - Proceedings Supplements*, 166(0):159 – 165, 2007. Proceedings of the Third International Conference on Particle and Fundamental Physics in Space Proceedings of the Third International Conference on Particle and Fundamental Physics in Space.

[52] Ch. Schenk U. Tietze. *Electronic Circuits - Handbook for Design and Applications*. Springer, twelfth edition, 2002.

[53] Hugh Young. *Sears and Zemansky's university physics*. Addison-Wesley, Boston, 2012.

Appendix B

Definitions

Please see below for the definitions of items used in this text.

B.1 Commonly Used Variables

A_c = Total Controller Gain

A_s = Total System Gain

c = Heat Capacity

c_l = Speed of Light

f = Frequency

g = Total Gain

i = Imaginary Unit Vector

k_p = Proportional Gain

k_i = Intergral Gain

k_d = Differntial Gain

λ = Wavelenght

L = Lenght

m = mass

T = Temprature

t = Time

ω = Angular Frequency

U = Internal Energy

u = Unknown function

v = Velocity

V_i = Voltage In

V_o = Voltage Out

B.2 Acronyms

- AC : Alternating Current
- ADC : Analogue to Digital Converter
- BBR : Black Body Radiation
- DC : Direct Current

- ECDL : External Cavity Diode Laser
- FFT : Fast Fourier Transform
- GPS : Global Positioning System
- HF : High Frequency
- Laser : Light Amplification by Stimulated Emmision of Radiation
- LF : Low Frequency
- OS : Operating System
- PC : Personal Computer
- PCB : Printed Circuit Board
- PID : Proportional, Integral, Differnetial [Control(ler)]
- UHFC : Ultra High Fineese Cavity
- ULE : Ultra Low Expansion Material
- UoB : University of Birmingham
- USB : Universal Serial Bus

Appendix C

Further Applications

C.1 Atomic Chamber

The strontium atoms used for the reference section of this experiment are held at sub millikelvin temperatures in a vacuum system, however, the chamber that houses this vacuum system is several orders of magnitude hotter. This means that it emits black-body radiation (BBR) onto the cooled Strontium atoms[34]. The BBR shift was reported in 2013 to be one of the largest irreducible shifts facing optical clock development[2]. The effect can be substantially reduced by cryogenic cooling of the clock and has been successfully attempted in Japan[35], although, since characteristic BBR has wavelengths 100x longer than the optical transitions involved, the shift can be approximated by the DC Stark Shift with $\sim 1\%$ accuracy[43]. This can be determined analytically and subtracted from the measured value[8]. However, this requires knowing the temperature exactly. In addition if there are any temperature gradients across the chamber then atoms on one side will be disproportionately affected in comparison to the other atoms. For small chambers chambers in compact systems, there is heat dissipation from magnetic coils, lasers, resistance heating of SrO, etc. Since it will take time for the system to thermalise, active compensation must be used to hold the system at a steady temperature.

C.2 Laser Systems

It is very well known that diode lasers have a strong temperature dependence[4]. It is normal lab practise to hold these diode lasers in the region $(15 \rightarrow 60)^oC$ with a stability of $10mK$. In any atomic physics experiment it is common to

have many such lasers which all have in-dependant analogue temperature control. Centralised digital control would offer a much cleaner environment for laboratory work. Such a system should have graphing and data logging facilities.

C.3 Wider application of Temperature Stabilisation

The two systems discussed above and the UHFC are just three examples from the work on the Optical Clock in Birmingham where very accurate temperature control is required. If a system can be built that meets the requirements of all of these systems it can generically be applied to many other experiments. From work with cold atoms, Portable Gravimeters and Gradiometers require the laser and cavities to trap the atomic sample[18] and from the field of astronomy temperature sensors are used to lock their optics to cavities[26]. A comprehensive list of examples could be longer than this report and so isn't included.

Appendix D

Component Selection & System Design

Here I will discuss why the digital components were chosen and how they were implemented.

D.1 Selection of a Digital Controller

Digital systems require careful section of a processor unit. These units fall into two distinct categories, logic systems and computer architecture[39]. Logic systems are designed for a specific application using gates such as NOT , AND and OR [39] and therefore they cannot be modified easily and so computer architecture is preferred. Since this system may be implemented on portable projects with size and power constraints, the processor is limited either a Mini-PC or Micro-Controller.

As discussed in section 2.1.2, ms loop times are required[1].Assuming 1000 instructions per loop, the processor clock speed should be MHz. Furthermore, for optimum PID operation, each loop of the control algorithm must take roughly the same amount of time since the PID parameters depend on the phase shift through the system. This limits the choice to either: a) A Mini-PC with a GHz processor and carefully programmed scheduling daemon or b) A dedicated MHz microprocessor with no operating system, running a single threaded program. (a) is theoretically possible, but (b) is far easier to implement.

[1]Furthermore, thermal fluctuations can be up to $1Ks^{-1}$ (See section F.3 for details.) so for mK stability ms loop times are certainly required

D.1.1 The Arduino

The Amtel ATmega range of microprocessors feature an AVR architecture processing unit which can run at up to $20MHz$[7]. The AVR architecture is specifically designed to be compatible with high level programming languages such as C++ and bespoke compilers have been designed, such as the avr-gcc. Furthermore, the Amtel ATmega328 is available on prepackaged with a clock and USB interface on the Arduino Nano Revision 3[5]. The Arduino Nano is one of a family of open source microprocessor boards that have been developed to lower the barrier of entry for microprocessor use[6]. These come complete with an Integrated Development Environment[5] and cross platform compiler. The low barrier for entry makes the Arduino a superb choice for Physics applications where the user is likley to be technically minded but not necessarily familiar with micro-controllers. Due to its small size, breadboard mounted capability the Arduino Nano was chosen over other ATmega328 powered boards[2].

D.2 Design

As shown in Table 1.1 it was desirable to keep the total size of the system small, while retaining both a non PC output, and a PC output. Processing data to produce statistics, logs and graphs is often very CPU intensive and so to keep the loop time as low as possible, it is desirable to move this onto a separate processing unit. These operations are likely to only be conducted on one temperature controller at a time and so one unit can be used for controlling several Arduino's. A computer can be used for this, but a Raspberry Pi can make a much more permanent solution. The two devices can then communicate over USB[3]. The leads to a 2 block system. Firstly, there is a controller that is able to accept set-point modulation and informs the user if system is stable, this consists of a Arduino Nano and some additional electronics. Secondly, there is an interface unit, that allows dynamic control, such as reprogramming the controller, changing the set points, *etc.* A block diagram for the entire system is available in Figure D.1.

D.2.1 Interface

Two levels of interface have been programmed on the temperature controller. As a standalone controller the system has been programmed to output the following data to the user, through a series of LEDs.

[2]Since the project began Arduino have released the Arduino Yun and Zero which may be of interest.

[3]There is a limit of 127 USB devices per PC, in practise no more than 50 controllers should be connected to a Pi

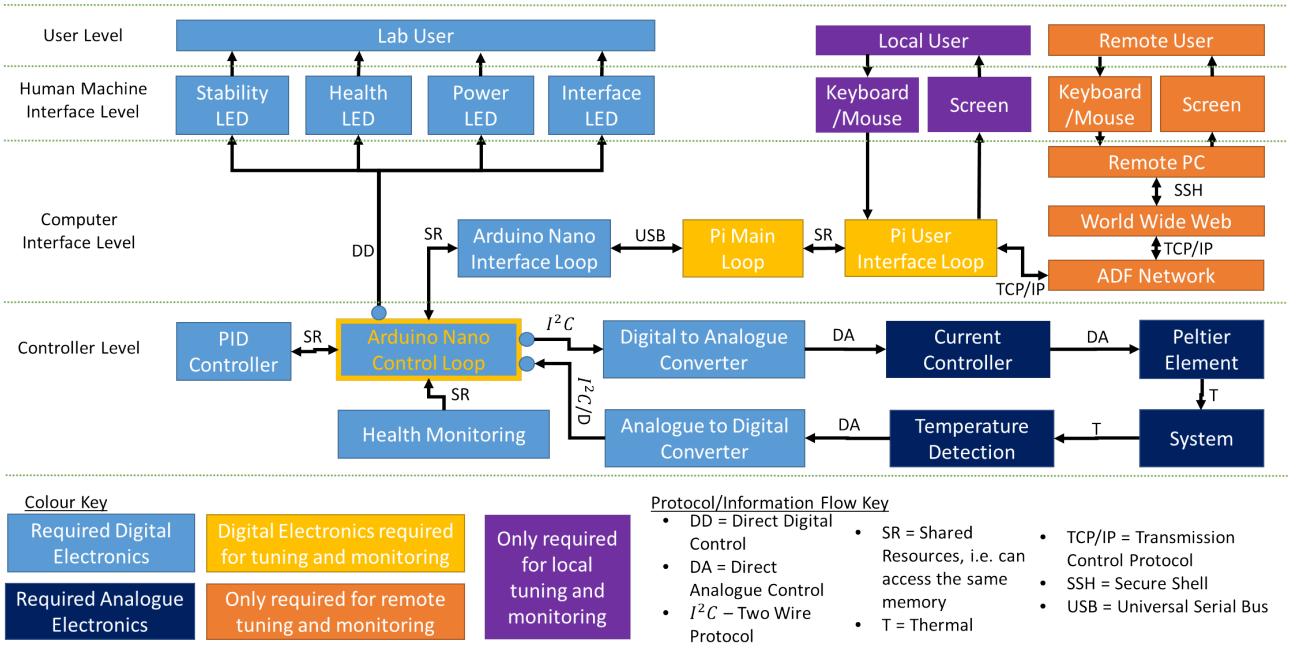


Figure D.1: Information flow through the control system. Blue blocks represent those belonging to the temperature controller. Coloured blocks are generally run on an optional computer or Raspberry Pi. These blocks that allow additional features, such as tuning, monitoring, reprogramming etc. There are two options, remote and local, these correspond to orange and purple blocks respectively. The yellow boxes are required for any form of monitoring. In this system information flows from all directions towards the 'Arduino Nano Control Loop' placed centrally in the chart where it is processed. The 'Controller Level' shows the feedback system employed for the temperature control: the new set point is calculated as a voltage, converted to a current and passed through a Peltier Junction; this affects the temperature of the system, which is fed back to the Nano and passed back to the PID controller.

Powered This LED lights if the system is powered, it is inbuilt on the Arduino board.

Communicating These are also inbuilt LED's and flash when the Arduino communicates over USB.

Loop active This LED changes state on every loop of the controller. If the led is not flashing then the controller has frozen. This LED was added to assist debugging.

Connected to PC This LED indicates whether the Arduino is communicating the the User Interface program, developed specifically for this controller.

Manual Setpoint Adjustment Enabled This LED indicates whether the user has enabled adjustment of the set point.

Stable This led lights when the system has remained within a specified range of its set point for a set period of time (default is $\pm 10mK$ for $500ms$).
In addition, the use has the ability to enable a switch, which lights and causes the device to adjust the set-point based on the setting of a potentiometer. Since this introduces noise, the switch disables this feature.

Dynamic Mode

The requirements for the controller include scope for a PC interface. To that end, I have developed a Graphical User Interface (GUI) for the temperature controller. As discussed previously, because this is processor intensive it runs on a Raspberry Pi[4] separate from the temperature controller and connected via USB. The Pi can then allow direct access to this GUI via a keyboard, mouse and screen (local mode); or the Pi can act as a Secure SHell server (SSH), allowing the user to control the parameters from any networked computer (remote mode). The block diagram for this is available in Figure D.1. The code used for the GUI is available in the appendix, it is well commented. Table D.1 shows a list of features and which ones require the GUI available in dynamic mode.

D.2.2 The program flow

The Nano runs a single threaded program using classes to separate the control elements, hence the 'Arduino Nano Control Loop' can be thought of as the main loop, a flow chart is shown in Figure D.2. This loop can be thought of as a three step

[4]Or any Linux/MAC/Windows installation provided python2.7 and the Arduino IDE are installed.

Function	Standalone Controller	With GUI	After Re-Programming	After Re-Coding	After Re-Soldering
Maintain the temperature	✓	✓	✓	✓	✓
Inform the user on exceeding this range	✓	✓	✓	✓	✓
Control Peltier up to 1.8A	✓	✓	✓	✓	✓
Initial set up	✗	✓	✓	✓	✓
Adjust the set temperature temporarily	✗	✓	✓	✓	✓
Change the PID constants temporarily	✗	✓	✓	✓	✓
Change the required accuracy temporarily	✗	✓	✓	✓	✓
View live temperature statistics	✗	✓	✓	✓	✓
Graph output	✗	✓	✓	✓	✓
Change default parameters	✗	✗	✓	✓	✓
Change averaging time	✗	✗	✓	✓	✓
Change between 16 and 24 bit precision	✗	✗	✓	✓	✓
Control Peltier to 2.5A	✗	✗	✗	✓	✓
Change the control algorithm	✗	✗	✗	✓	✓
Control Peltier to 8A	✗	✗	✗	✗	✓

Table D.1: A table showing the requirements to implement different features of the temperature controller. On the left is a list of properties that it is possible to change, on the right is the mode required to do this. Re-Programing refers to changing constants in the constants section of the Arduino code and then uploading this to the controller. This is required to change default behaviours. Re-Coding means changing constants located elsewhere in the code, or perhaps implementing a new class. Re-Soldering refers to changing passive component values.

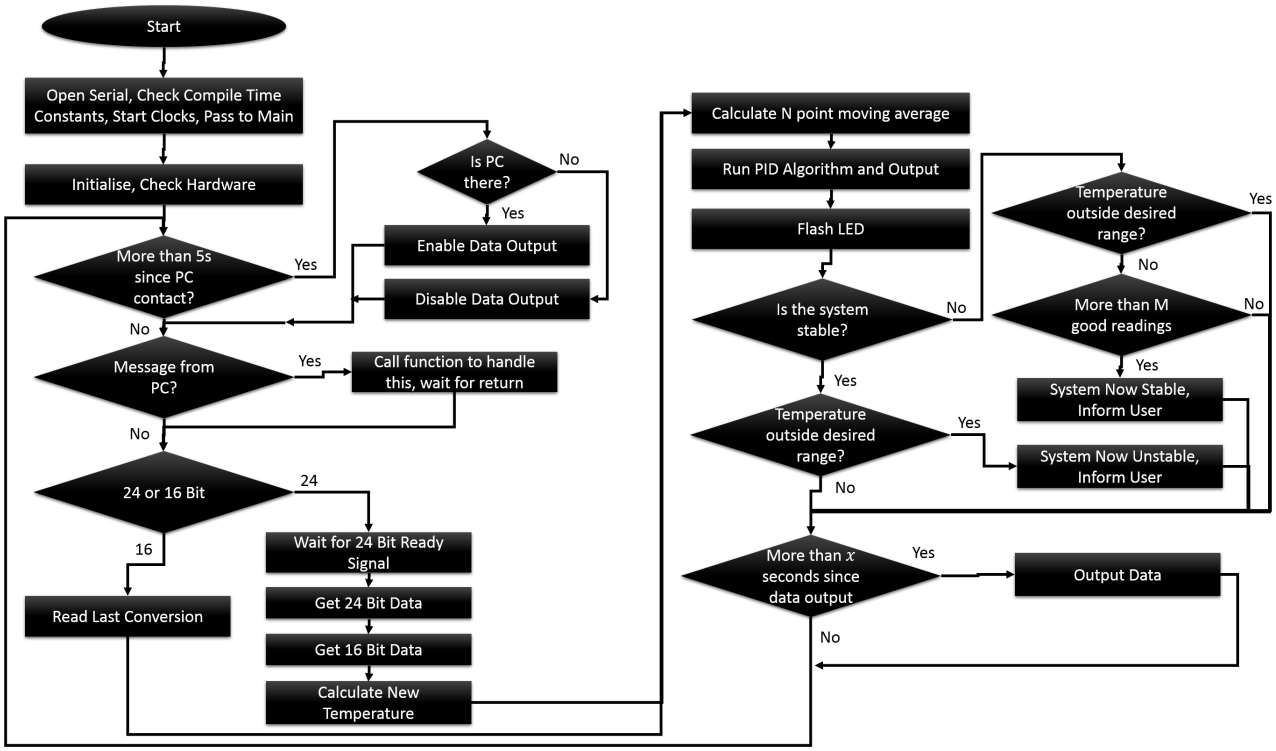


Figure D.2: A simplified structure of the main execution loop on the Arduino. The model is asynchronous and partially interrupt based. On every loop the program checks to see an interrupt is waiting, if this check will take more than a few microseconds then the program simply checks to see if the wait condition has been exceeded and runs the check once every few seconds. This means that on most loops the loops the Arduino simply: evaluates a few *if* statements, reads in data, calculates the new PID settings and finally, outputs the new correction value. This allows complex tasks to take place, while keeping loop time low.

process:

1. Collect information from the Analogue to Digital Converter,
2. Call the PID algorithm,
3. Outputting to the Digital to Analogue Converter.

After a specified number of loops it will run the 'Arduino Nano Interface Loop'. This isn't a real loop but a set of statements that are executed only after a certain number of cycles. These statements will check whether the PC is still connected and exchange information over USB if requested. If the PC is no longer connected then the data output will be disabled to save CPU cycles. The full code is available in the appendices. I developed the code structure and each of the major classes. Because the system was designed in a modular way I was able to build on several pieces of code that had already been developed: the PID algorithm is fairly generic and is slightly modified from a Arduino Playground Example released under the GPLv3 and written by Brett Beauregard[9]; the code to generate a $2MHz$ clock pulse for the 24 Bit ADC was written by *davekw7x* as an answer to an Arduino Community Forum Question[17]; finally, the code used to communicate with the 24 Bit ADC was written and tested by Iain MacIver as part of a investigation into MetGlas Magnetic Shielding[1].

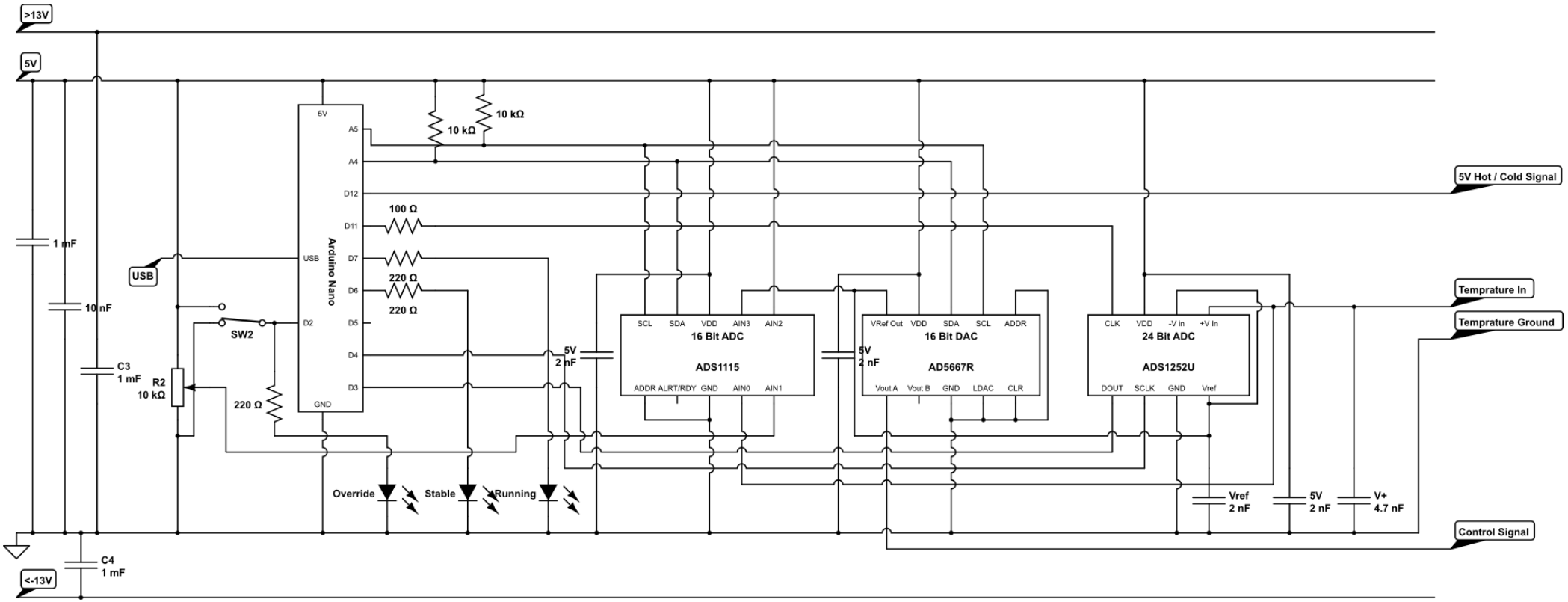
D.3 The Final Circuit

The final circuit is split into three parts to make it easier to understand. These are: the digital processing circuit, the current controller and the current regulator. These blocks are chosen as they have three sets of requirements.

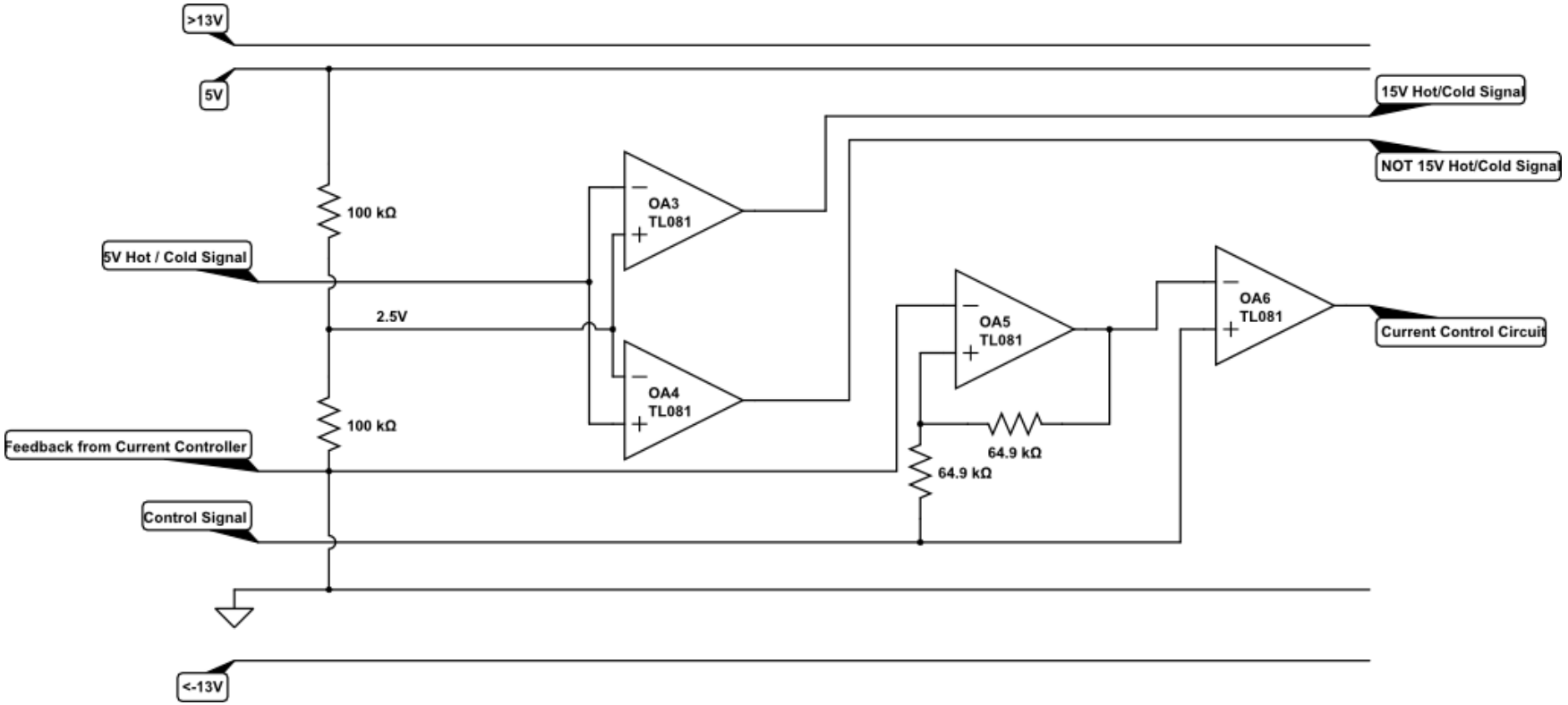
The digital circuit is a low voltage circuit ($5V$) with some high frequency channels and a low current. This means it only requires standard size PCB tracks and a ground plane, in addition the 5V and ground must be very stable.The current controller has a mix of higher voltage components, $\pm 15V$, and lower ones, $+5V$. The current is low and there are no high frequency components. This reduces the number of contraints on the PCB design.

The current regulator has $+15V$ supply and enough current to drive the pelitier, this can be up to $8A$ if the correct components and voltages are chosen. The PCB must be designed using thicker tracks capable of carrying such high currents.

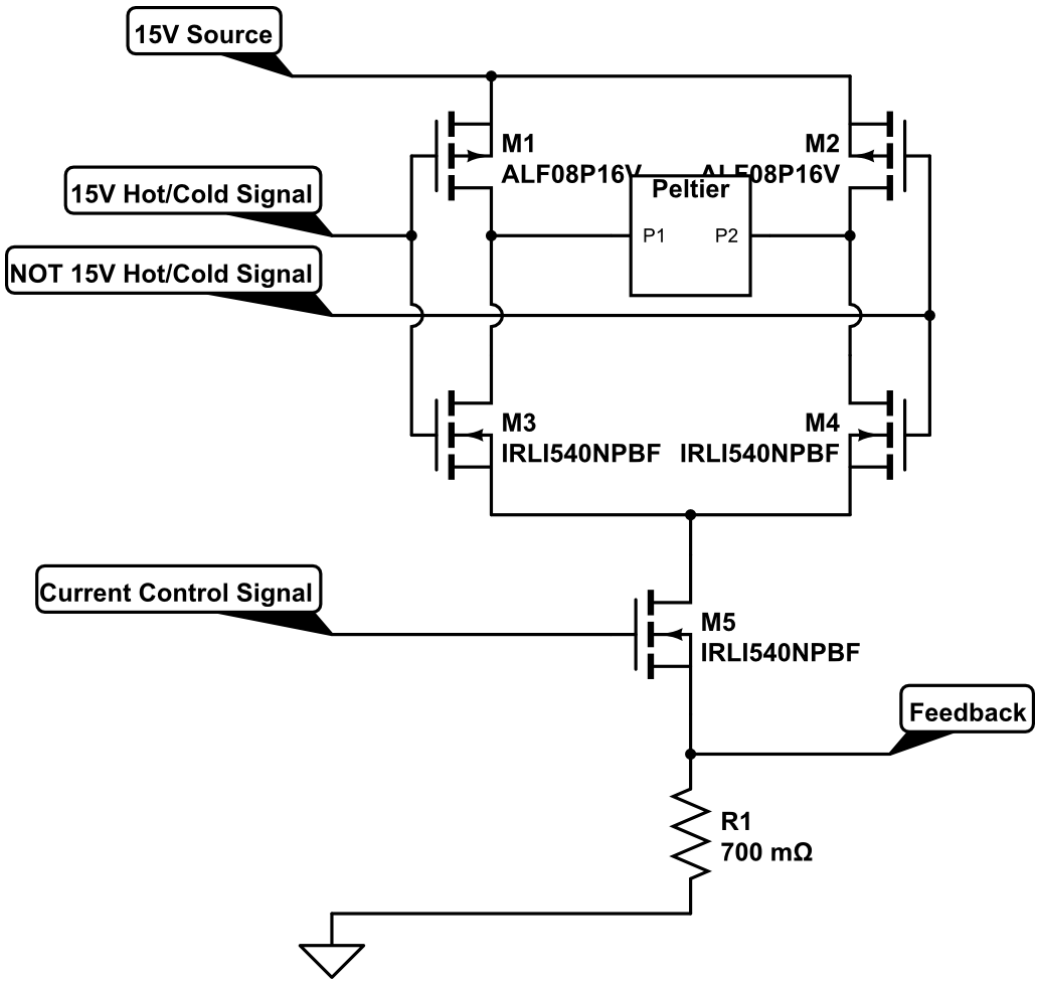
With the addition of a temperature sensor, as discussed in section 2.3 this forms the complete circuit.



Circuit Diagram D.1: The digital processing circuit. Inputs to the Nano are shown on the left, outputs on the right. The outputs from this circuit go to Circuit Diagram D.2. Further discussion is available in subsection D.3.1.



Circuit Diagram D.2: The current controller circuit. The inputs come from Circuit Diagram D.1 and the outputs go to Circuit Diagram D.3. Further discussion is available in subsection D.3.2.



Circuit Diagram D.3: The current regulator circuit. This circuit has regulates the flow of positive and negative current through the Peltier junction based on the inputs from Circuit Diagram D.2. Further discussion is available in subsection D.3.3.

D.3.1 The Digital Processing Circuit

The digital processing circuit is shown in Circuit Diagram D.1. Leftmost are three capacitors, since the Peltier element and this circuit share a common ground, this stabilise each of the voltage lines. The capacitors are large to account for the high currents involved. There is an input to allow the circuit to be connect to the PC via USB. $SW2$ and $R2$ allow the user to enable analogue set point modulation and control it respectively. The override LED is hardwired to this input. Outputs $D6$ and $D7$ are connected to the Stable and Running LED's respectively, though a 220Ω resistor for protection. $D4$ and $D5$ are used for communication with the 24-Bit ADC. $D11$ is used as a $2MHz$ clock for the 24-Bit ADC, when 24 Bit mode is disabled this is turned off to reduce noise in the circuit. The 100Ω resistor after $D11$ reduces the effect of small oscillations following the change of state of the clock. Pin $D12$ tells the current regulator circuit in which direction the current should flow through the Peltier element. $A4$ and $A5$ are used for the I^2C communication protocol, the $10k\Omega$ pull-up resistors ensure that the default state of the line is high, as per the protocol. To further reduce noise, each digital component is fitted with a $2nF$ capacitor between the $5V$ input and the local ground. This is placed as close to the input as possible. Finally, $V+$ on the 24 Bit ADC is fitted with a $4.7nF$ capacitor, this is designed to cut out any high frequency noise as discussed in section 2.3.1.If it is necessary for the circuit to monitor its own temperature then the connection between $V+$ on the 24 Bit ADC and $AIN0$ should be broken, then $V+$ can be used for monitoring the element temperature and $AIN0$ for monitoring the circuit temperature. If the circuit actively controls its own temperature then $VoutB$ on the 16 Bit DAC can be used for this.

If the controller is required the control many temperature elements, then multiple 24 Bit ADC's can be added. The same CLK, SCLK, V-, and Vref connections can be used, however, DOUT will need to be connected to a new pin on the Arduino, these can be read simultaneously. If additional DAC's are required, then these can be added, although they will require additional addresses, please consult the data-sheet for how to achieve this.

D.3.2 Current Controller

The current controller circuit is shown in Circuit Diagram D.2. This circuit consists of two parts, OA3 and OA4 compare the input signal against a 2.5V reference. If this signal is greater than 2.5V, OA4 goes to +15V and OA3 goes to 0V and vica versa. The second part is the feedback for the current regulator, the input is doubled to increase stability by the non-inverting Op-Amp OA5; OA6 compares this against the desired amount of current and the signal ramps down,

decreasing the current through the circuit if it is to high and vica versa.

D.3.3 The Current Regulator

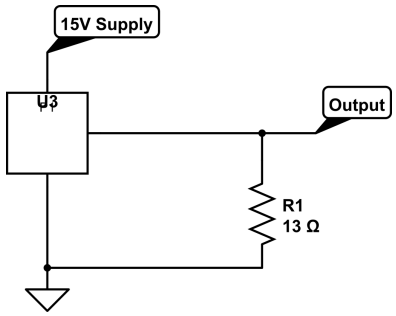
The current regulator circuit is shown in Circuit Diagram D.3. Its behaviour is covered in Circuit Diagram 2.3, with the switch replaced by the control as shown. If more Peltier elements are required then an additional one of these circuits will be required for each one.

D.4 The Thorlabs AD590

It has been known for many years that Diodes have a strong temperature dependence[14], Thermodiodes and Transistor based Temperature Transducers exploit this effect in order to generate a stable temperature reading[12] by outputting an exact amount of current per degree of absolute temperature[52]. The AD590 is such a sensor and will output exactly $1mA$ per degree kelvin[19]. Because the device outputs a current, implementation is easy as shown in Circuit D.4

Comparison to the Thermistor

The $10k$ Thermistor gives a good indication of absolute temperature and of change in temperature, however, over time the Resistance-Temperature relationship may drift[49], furthermore the Resistance of all of the connections thermistor will affect the reading. Because the AD590 outputs a current, the nature of the connections is not relevant to the transducer and does not affect the reading[19]. The AD590 is designed with very stable systems in mind, whereas the $10k$ is an accurate quicker and cheaper solution.



Circuit Diagram D.4: Suggested circuit for measuring the temperature with an AD590. Since the device outputs a current a single resistor is all that is required. The power and ground should be local to the measuring device, not the AD590 to avoid errors[19]. It may be required to connect a non inverting active low pass filter if the signal is noisy.

Appendix E

Quick Start Guide

In this Appendix I will briefly explain the steps required to implement the temperature controller. The advice here is based on a mixture of guesswork, instinct, practise and a small amount of science. I do not attempt to explain why the methods work, or reference how I came to deduce them, especially since some of the fields, such as 24 Bit sensing, are generally considered to be a *dark art*. Hopefully it will help, but please do not feel limited by the content. If you are looking to implement this system and are after advice then please feel free to contact me on `mailto:axj336@gmail.com`. I am happy to respond to any questions even long after graduation. Table E.1 shows a list of components.

E.1 Printing the circuit

First I suggest that anyone wanting to implement this circuit gets the test system working, as a proof of concept. Then before printing the circuit you must asses your requirements. For a general system with one Peltier Junction, $10mK$ stability and a thermistor then the 24 Bit ADC is not required and circuits, D.1, D.2, D.3 and 2.1b can be printed. Note that Circuit Diagram D.3 will require thicker traces and a ground plane should be implemented on all PCB's to reduce noise. With this design, electrical noise on the analogue inputs is the limiting factor, so pay attention to this when designing the PCB.

For additional Peltier Junctions, additional DAC's and copies of Circuits D.3 and D.2 will be required. These will require additional code. Lastly, for each sensor an analogue input will be required. You do not need to monitor the 2.5V voltage if you are not using the 24 bit sensors, in addition the set point modulation can be moved onto one of the 10 Bit ADC's on

Component	**Suggested Component**	**Quantity Required**
Very Stable 0.75 Ohm Resistor	Any as long as low temperature coefficient	1
P-Type MOSFET	ALF08P16V	2
N-Type MOSFET	IRLI540NPBF	3
Peltier Element		1
LED	L-53GD (Green);	1
LM358 (Op Amp)	LM358APE4	3
Approx 220 Ohm Resistor	Any / MF50 220R	1
Arduino Nano	Nano 3.0 Atmel Atmega 328 MCU board	1
Approx 100k Resistor	Any / MF25 10K	2
Approx 100 Ohm Resistor	Any	3
Very Stable 10KOhm Resistor	PTF5610K000BYEB	1
1000uF Capacitor	Any	1
10nF Capacitor	Any	1
4.7nF Capacitor	Any	1
1nF Capacitor	Any	4
24 Bit ADC (ADS1252U)	ADS1252U	1
16 Bit ADC (ADS1115)	ADS1115	1
16 Bit DAC (AD5667R)	AD5667R	1
Pi V2	Raspberry Pi 2 Model B	1

Table E.1: Components List

the Nano. This leaves two sensors on the 16 Bit ADC for temperature control. Additional 16 Bit ADC's and DAC's can be connected on the same I^2C line as long as the correct address is chosen, the datasheets explain this fairly well.

E.2 Adjusting the code

The code is available in a Bit-bucket repository:

`https://bitbucket.org/surfmanjones/arduinotempraturecontroller`

and is also shown in Appendix You must choose whether 16 or 24 bit mode is to be used, paying attention to the noise constraints discussed in subsection 3.1.2. Editing the code should be obvious, it is well commented and structured into high level blocks. There is a slight subtly in reading several 24 Bit DAC's. The can be read simultaneously provided that the same $SCLK$ is used. Read in the whole register and then mask the bits into the relevant variables. This must be done using port level operations. A good guide is available here: `http://www.arduino.cc/en/Reference/PortManipulation`.

E.3 Setting Up

It is recommended that each PCB is tested before use. Then test your system using test Peltier Junctions and sensors, small blocks of aluminium will do, if it fails, it should do disastrously, this makes it easy to debug, but may damage a sensitive system. The code must be uploaded to the Arduino using a computer. If a Pi is to be used, then this must be set up, default Raspbian, will do, then install python2.7 and the Arduino environment, both are available in the standard repositories[1]. Then load the scripts onto the Pi and program.

E.4 Suggested Tuning Procedure

First you must calculate a reasonable average time for your system, if you estimate the mass and the specific heat capacity and the power output by your Peltier element, then the time to change the temperature by $1mK$ is a reasonable estimate. To tune the PID, use guesswork, there are various guides advice available on the Internet, but guesswork is by far the

[1] If you open terminal and connect to a network, then the command *sudo apt-get install python && sudo apt-get install arduino*, should do this for you. Installing software on Linux is normally done via apt-get, its much easier than on Windows.

easiest. Start by setting the integral and differential constants to zero, and find a proportional constant that is as high as possible without causing oscillations. Then reduce that by about 5% and slowly increase the integral constant until you see oscillations. Differential simply amplifies noise, so I wouldn't implement it.

E.5 Circuit Validation

It is important to test that the measured temperature is the actual temperature. To do this I used an accurate voltmeter to ensure that the measured voltage and the actual voltage were the same on the inputs to the ADC's. I then calculated the systematic errors due to resistance in the connecting wire. Then I measured the temperature of a body with two thermistors to guard against and manufacturing defects. If these checks are all satisfactory, then there are no errors in the system.

Appendix F

Derivations

F.1 A Mathematical Model for a Controlled System

Consider the controller shown in Figure 2.1, if the controller has gain, A_c, and the system has gain, A_s, then it is possible to deduce to following expressions,

$$r(t) = s(t) - o(t), \tag{F.1}$$

$$u(t) = e(t) + c(t), \tag{F.2}$$

which simply reduce the number of variables. Now, adding in the amplification factors,

$$c(t) = A_c r(t), \tag{F.3}$$

$$= A_c \left(s(t) - o(t)\right), \tag{F.4}$$

$$o(t) = A_s u(t), \tag{F.5}$$

$$= A_s \left(e(t) + c(t)\right), \tag{F.6}$$

therefore by re-arranging[52],

$$o(t) = A_s \left(e(t) + A_c \left(s(t) - o(t)\right)\right), \tag{F.7}$$

$$o(t)(1 + A_s A_c) = A_s A_c s(t) + A_s e(t), \tag{F.8}$$

$$o(t) = \frac{A_s A_c}{1 + A_s A_c} s(t) + \frac{A_s}{1 + A_s A_c}, \tag{F.9}$$

we can produce a mathematical model of the system[1]. Now it is possible to determine the response of the output, $o(t)$, to some input error signal, $e(t)$, by calculating $\frac{\partial o}{\partial e}$[52]

$$\frac{\partial o}{\partial e} = \frac{A_s}{1 + A_s A_c}. \tag{F.10}$$

Thus we can deduce that for any controller, it is essential to make the total controller gain as high as possible in order to keep the response to any unwanted disturbances, $\frac{\partial o}{\partial e}$, as low as possible.

F.2 Response of a System to Temperature Changes

By approximating the geometry of the controlled system to a cubic aluminium block of dimensions $(3 \times 5 \times 1)cm$ and assuming that the aluminium thermalizes instantly, it is possible to do some representative calculations for a laser diode mount. Using the first law of thermodynamics,

$$\Delta U = Q_{\text{in}} + W_{\text{on}}, \tag{F.11}$$

and asserting that an internal energy change will result in a temperature change given by,

$$\Delta T = \frac{\Delta U}{mc}, \tag{F.12}$$

it is possible to determine that,

$$\Delta T = \frac{P(t)\delta t}{mc}, \tag{F.13}$$

where P is the power of the controlling system. Thus taking differentials,

$$\frac{\partial T}{\partial t} = \frac{P(t)}{mc}. \tag{F.14}$$

Now, Fourier Theory states that any periodic signal can be expressed as the sum of sine waves. So calculating $T(t)$ for the case $P(t) = Ae^{i\omega t}$ gives,

$$T(t) = \frac{A}{mc} \int_0^{t'=t} e^{i\omega t'} \mathrm{d}t'. \tag{F.15}$$

[1]This derivation is shown here for completeness, it is also shown in Electronic Circuits[52], which was used extensively while modelling the PID controller.

Solving and applying the initial condition $T(0) = T_1$ gives,

$$T(t) = \frac{-Ai}{\omega mc} e^{i\omega t} + T_1 \tag{F.16}$$

$$= \underbrace{-i}_{\phi_s} \times \overbrace{\frac{1}{\omega mc}}^{A_s} \times P(t) + T_1. \tag{F.17}$$

Thus the phase response, $\phi_s = -i = -90^o$ and the amplitude response, $A_s = \frac{1}{\omega mc}$, of the system to a periodic input signal $P(t)$ can be deduced. If the system does not thermalize instantly but does so over some time then there will be an additional phase and amplitude response.

F.3 Estimation of Thermal Fluctuations

A very rough estimation of the thermal fluctuations in an air conditioned laboratory can be obtained using the following estimations:

1. Room size $\sim (2 \times 10 \times 10)m = 200m^3$
2. Air inlet vent size $\sim 1m \times 1m$
3. Air conditioning flow rate $\sim 20ms^{-1}$
4. Air inlet temprature is room temprature $\pm 10^0 k$
5. Air pressure $100kPa$, Room Air Temperature $273^o k$, Gas Constant $286JKg^{-1o}k^{-1}$

Then by applying the ideal gas law,

$$\rho \approx \frac{P}{RT}, \tag{F.18}$$

the density of air can be approximated to be $1.3Kgm^{-3}$. The mass of air flowing into the room is therefore $1.3 \times 1 \times 1 \times 20 = 26$ and the mass of air in the room is $1.3 \times 2 \times 10 \times 10 = 260$. Assuming that the mass of air remains unchanged, the average energy change of the air can be determined,

$$\delta E_{av} = m_{in} c(\pm 10)\delta t. \tag{F.19}$$

Dividing through by the total mass of air and the heat capacity gives the average temperature change in to room,

$$\delta T_{av} = \frac{m_i n}{m_{\text{room}}}(\pm 10)\delta t = \pm \frac{26 \times 10}{260}. \tag{F.20}$$

Letting $\delta t \to 0$ gives,

$$\frac{\mathrm{d}T}{\mathrm{d}t} = 1^{o}ks^{-1}. \tag{F.21}$$

F.4 Potential Divider

Assuming that the current drawn by the output is negligible, then the total current in the Circuit 2.1a can be calculated from Ohms law,

$$V_i = I_t R_t, \tag{F.22}$$

where V_i is the voltage in, I_t is the total current and R_t is the total resistance. Re-arranging and subtituing gives,

$$\frac{V_i}{R_t} = I_t, \tag{F.23}$$

$$\frac{V_i}{R_1 + Th_1} = I_t. \tag{F.24}$$

The re-applying Ohms law to find the voltage across Th_1 for current I_t,

$$V_o = I_t Th_1 \tag{F.25}$$

$$V_o = \frac{V_i}{R_1 + Th_1} Th_1 = V_i \frac{Th_1}{R_1 + Th_1}, \tag{F.26}$$

where V_o is the voltage at the output with respect to the circuit ground.

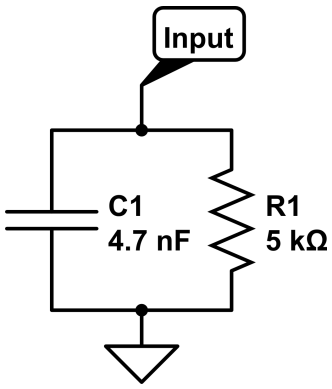
F.5 High Frequency Cut with parallel resistors

It is well known that a resistor and a capacitor can be used to cut frequencies higher than f_0 when configured as shown in Circuit Diagram F.1a. The actual response is more complex[53], however, it is an exceptionally common approximation to assume that any frequencies above f_0 are nulled and those below are left intact, where[13],

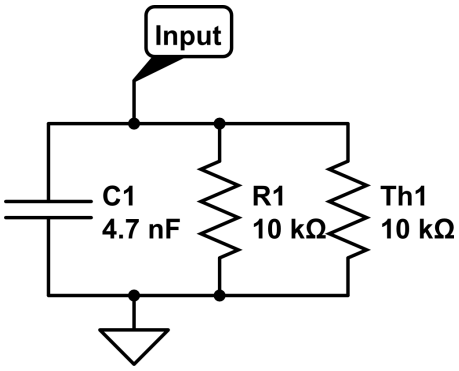
$$f_0 = \frac{1}{2\pi RC}, \tag{F.27}$$

where R and C are the resistance, $R1$, and capacitance, $C1$, respectively as defined in Circuit Diagram F.1a. Then when circuit F.1c is used,

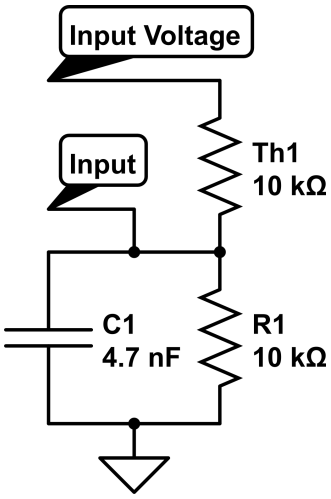
$$\frac{1}{R} = \frac{1}{R1} + \frac{1}{Th1} \tag{F.28}$$



(a) A standard low pass filter circuit. Frequencies higher than f_0 are passed to ground via the capacitor. The input is also the output for this circuit.



(b) This is the same low pass filter, since the total resistance is still $5k\Omega$.



(c) This is the same low pass filter, provided the voltage is constant, it doesn't matter where the resistor is connected.

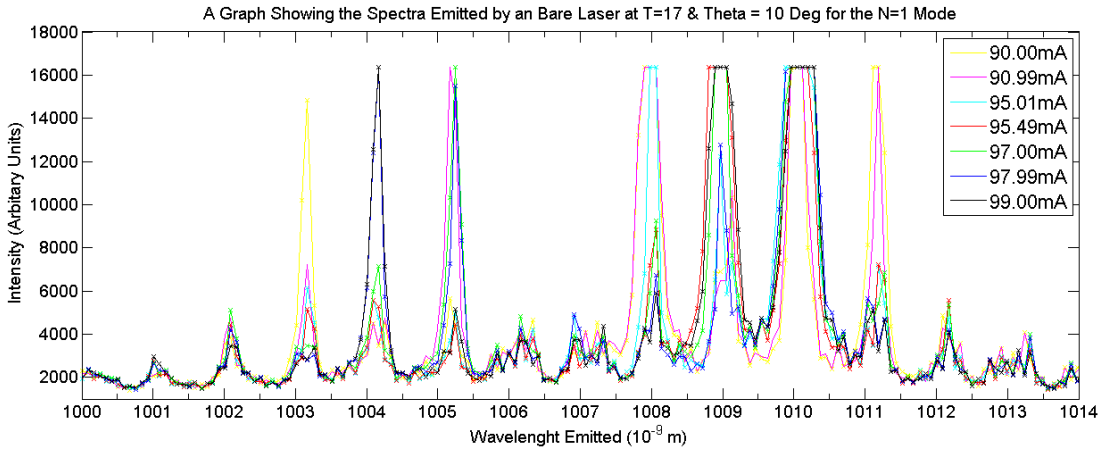
Circuit Diagram F.1: Three identical low pass circuits. It doesn't matter how the resistors are arranged provided the total resistance is the same.

gives the new resistance[13] and f_0 can be recalculated. If the resistance of $Th1$ varies with temperature then so will f_0. Calculating this for the values shown and a Thor Labs 10K Thermistor gives $f_0^{25^oC} = 6.77kHz$, $f_0^{0^oC} = 13kHz$ and $f_0^{50^oC} = 4.4kHz$. These value are chosen to be above the sample frequency of the ADC as discussed in subsection 3.1.2.

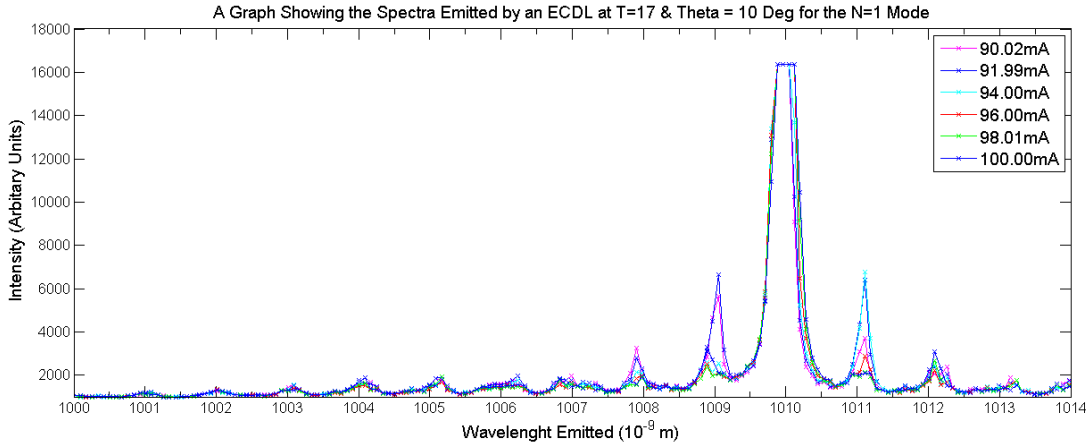
F.6 Laser Diode Temperature Dependence

Laser systems in general produce monochromatic, coherent and well collimated light[23], however, the laser cavity often supports many modes[46] as shown in Figure F.1a. This effect is more dominant in laser diode systems than other laser systems due to the small area in which the light is emitted from[26]. This effect shows itself as both amplitude and phase fluctuations[46][2]. In addition the collimation is often poor with diode laser systems due to diffraction in exiting the lasing cavity. The linewidth and collimation of the diode laser can be reduced using an External Cavity Diode Laser (ECDL) and

[2]These fluctuations can actually be accurately modelled as a coherent wave with Gaussian noise[32]



(a) An example emission spectra of a bare diode laser. The peaks show that many modes of laser operation are supported. Each of these modes competes in the laser system to become the dominant mode.



(b) An example emission spectra of a External Cavity Diode Laser (ECDL). The optical feedback from the ECDL back into the cavity causes one mode to dominate and suppresses all the others, as can be seen by contrasting with Figure F.1a.

Figure F.1: Data showing the modes supported in a Diode laser both with and without additional optics. This unpublished data was collected as part of work during previous studies[27]. Copies are available on request.

a lens[26]. An ECDL will decrease the number of modes emitted, as shown by contrasting Figure F.1a and Figure F.1b. However, the dominant mode is a still function of a 3D parameter space including the angle of the ECDL grating[27], the current flowing through the diode[44] and the temperature of the diode[44]. Furthermore, the angle of the grating is normally controlled using a piezo, which is temperature dependant, as is the length of the External Cavity. This makes an ECDL more temperature dependant than an ordinary laser diode.

In addition to the temperature dependence of the ECDL, there is also a dependence in the diode. There are many reasons this, however, one of the easiest to explain is the thermal expansion of the lasing cavity. A laser cavity in general has two mirrors at either end[23], these mirrors serve two purposes, a) to cause more passes of the laser beam through the gain medium, increasing amplification and b) to act as a high finesse cavity selecting just one mode out of the several supported modes.

Such a high finesse cavity works by setting up standing waves within the cavity. When the wave equation,

$$\frac{\partial^2 u}{\partial t^2} = v^2 \nabla^2 u \tag{F.29}$$

with $v =$ velocity which in this case is $c_l = 3 \times 10^8 ms^{-1}$, $u \equiv u(x, y, z, t)$ is a function describing the wave shape and all other symbols have their usual meanings, is solved for such a system, the result is that only standing waves of quantised values are allowed in the cavity. All other oscillations will die away[27]. These quantised values of wavelength $\{\lambda_n\}$ are given by

$$\lambda_n = \frac{2L}{n} \tag{F.30}$$

where n is a positive non zero integer and L is the length of the cavity[23]. Since $L = L(T)$ due to thermal expansion, it follows that $\{\lambda_n\} = \{\lambda_n(T)\}$. Thus the principal mode is temperature dependant. Temperature normally changes slowly and so this effect is normally seen in that after period of time, the laser frequency will drift to a new value, in some cases it may jump to the next mode supported by for that current, temperature and angle set.

Appendix G

Further Results

G.1 Power Supply Noise

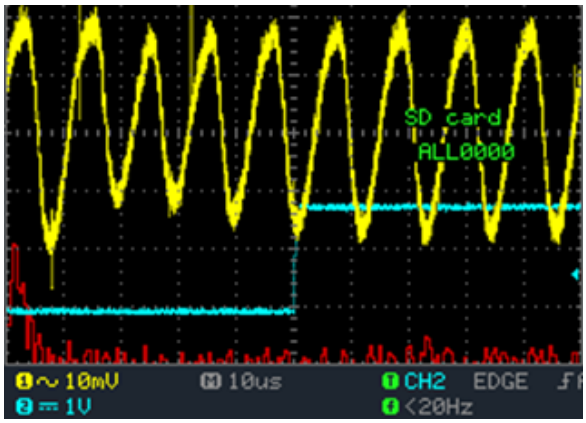
In addition to the noise discussed in subsection 3.1.2, there was also noise on the power supply. This was reduced with the addition of a $1mF$ capacitor on the $5V$ supply and a $1\mu F$ capacitor on the $2.5V$ as shown in Figure G.1 and G.2. The same procedure was carried out for all other voltage supply lines to ensure best stability.

G.2 Transient Response

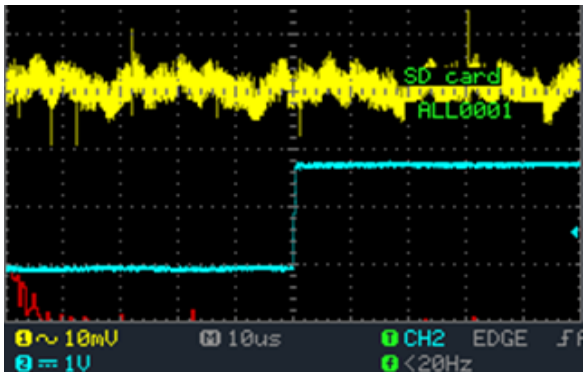
The design goal was to achieve a shift though the system of less than 1000 of the shift in the controller.

G.2.1 Time delay in the controller

The controller can be programmed to flip the state of an LED on every loop of the control algorithm. By doing this the time delay in the controller can be determined measured as shown in Figure G.3. The exact length of the process depends on the number of times the 16 Bit ADC is read as this takes $4ms$ per channel read. By enabling continuous conversion this can be eliminated if only one channel is read.

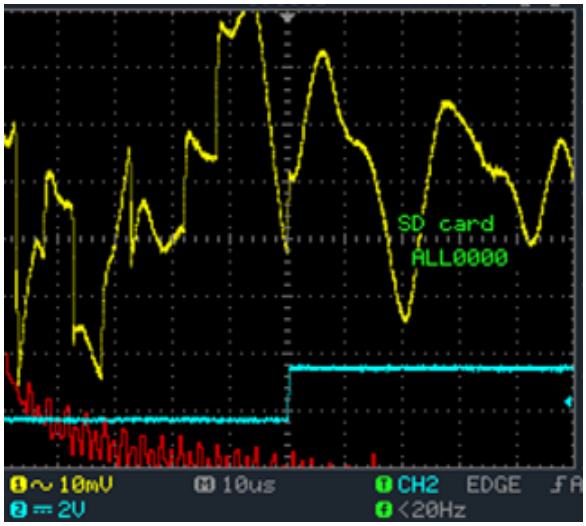


(a) Without $1mF$ capacitor

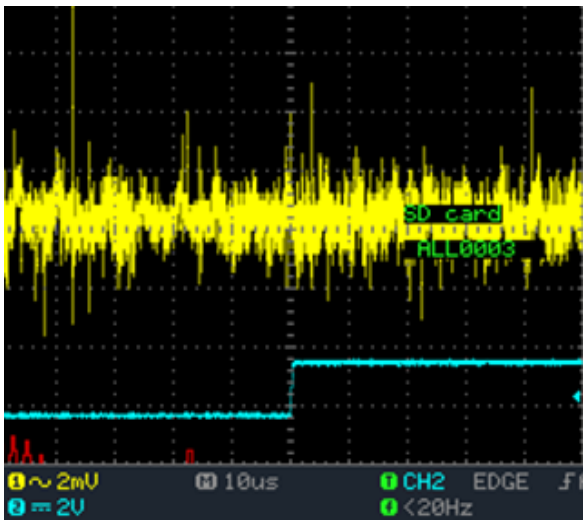


(b) With $1mF$ capacitor

Figure G.1: Noise on the stable $5V$ voltage lines. The blue line shows loops of the control program. The yellow line shows the signal on the $5V$ line and the red line is a FFT of this. A significant reduction in noise is visible with the addition of the capacitor.



(a) Without $1\mu F$ capacitor



(b) With $1\mu F$ capacitor. Note the increase in magnification

Figure G.2: Noise on the stable $2.5V$ voltage lines. The blue line shows loops of the control program. The yellow line shows the signal on the $5V$ line and the red line is a FFT of this. A significant reduction in noise is visible with the addition of the capacitor.

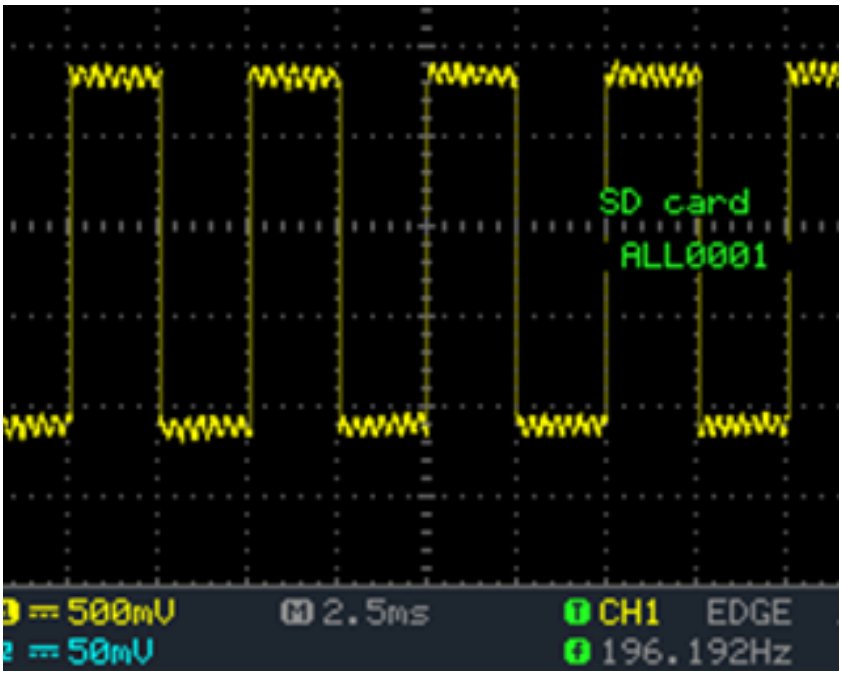


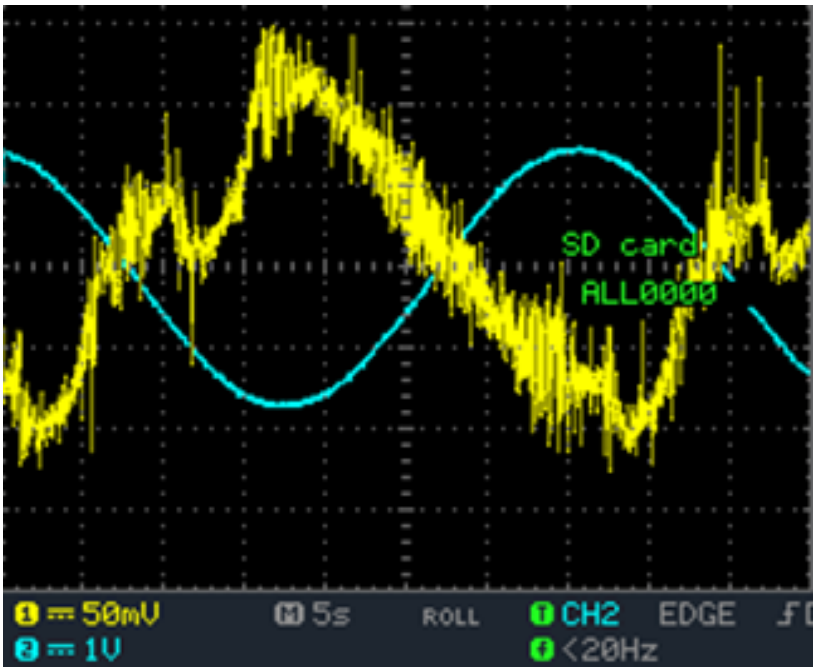
Figure G.3: The output clock from the Arduino. This clock changes state on every loop. Taking account of additional process that don't run on every loop by assigning a very liberal error gives a loop time of $(2.5 \pm 0.1)ms$. In this setup the 16 Bit ADC was not used at all.

G.2.2 Phase Response of the System

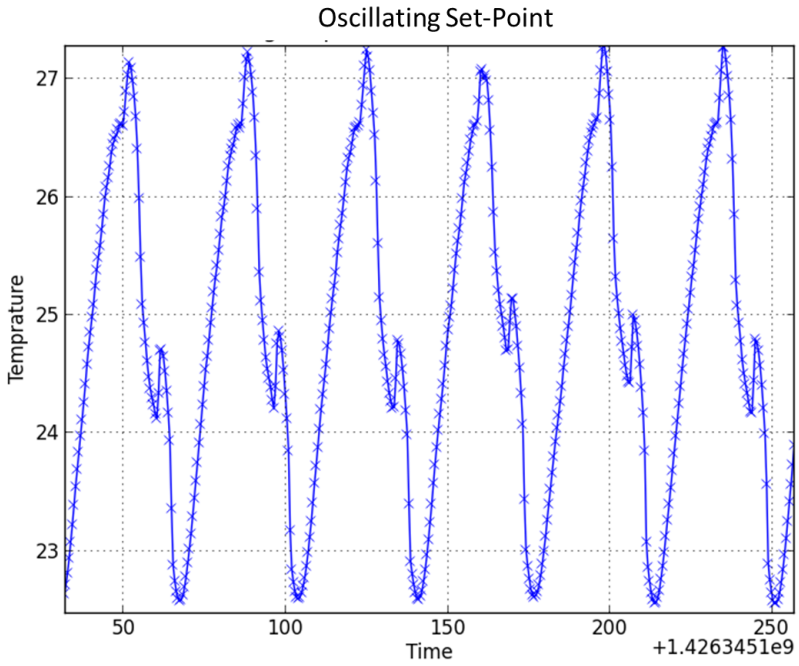
As discussed in section 2.1.2 there will be an additional response to the calculated 90^o phase shift in the system as it does not thermalize instantly. By modulating the set point through $(3.2 \pm 0.2)^oC$ peak to peak with a varying time period, it was found that $t = (38 \pm 1)s$ resulted in a π phase shift with no amplitude reduction as shown in Figure G.4. The average was disabled for this measurement. The theory in section 2.1 predicts that a decreasing amplitude response would result if the system was driven at a higher frequency than $t = (38 \pm 1)s$, this was found to be true as shown in Figure G.5, suggesting that the proportional constant was sufficiently low and did not lead to unstable positive feedback.

G.2.3 Phase Conclusions

The time delay in the controller was found the be around $\sim ms$ whereas the time delay in the system was found to be around $\sim 10s$, this means that the system meets its transient goal, of a controller delay less than a 1000^{th} of the system delay.

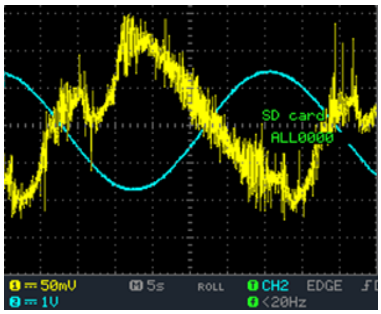


(a) The blue line shows the set point modulation around 25^oC, $1V$ is programmed to equal 1^oC change. The yellow line was input signal to the 24 Bit ADC, because the average was disabled, noise was amplified though the controller leading the very noisy signal shown.

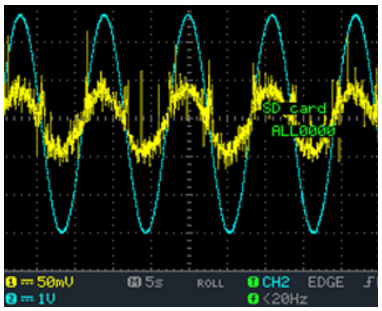


(b) This is the temperature response as seen by the digital Arduino when the set point was modulated as in a).

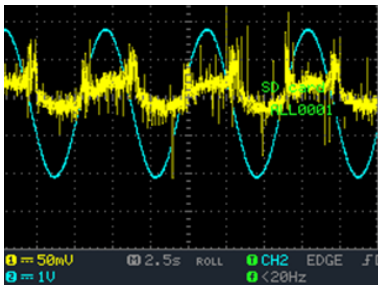
Figure G.4: Graphs showing the response of the system to a oscillating set point.



(a) Setpoint modulation at $t = (38 \pm 1)s$. The system is π out of phase with the controller and at full amplitude.



(b) Setpoint modulation at $t = (11 \pm 0.5)s$. The system is 2π out of phase with the controller with reduced amplitude.



(c) Setpoint modulation at $t = (7 \pm 0.2)s$. The system is $2\pi + \pi/4$ out of phase with the controller with significantly reduced amplitude.

Figure G.5: Graphs showing the effect of driving the system above the cutoff frequency. The blue line shows the set point modulation around 25^oC, $1V$ is programmed to equal 1^oC change. The yellow line was input signal to the 24 Bit ADC.

Appendix H

Code

For this project I have developed and written a number of pieces of code, these are shown below.

H.1 Key Programs

In this section I will present the code for the key programs developed for this controller.

H.1.1 Arduino File.ino

This file contains the temperature controller program to run on the Arduino.

```
/*#=============================================
#~~~~~  Arduino Temprature Controller ~~~~~~~
#---------------------------------------------
# Interface Code
# V1.0
# Author -  Aaron Jones
# Date: 25/03/2015
# Copyright Aaron Jones, 2015
#---------------------------------------------
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU Lesser General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
```

```
13 #    (at your option) any later version.
14 #
15 #    This program is distributed in the hope that it will be useful,
16 #    but WITHOUT ANY WARRANTY; without even the implied warranty of
17 #    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18 #    GNU General Public License for more details.
19 #
20 #    You should have received a copy of the GNU Lesser General Public License
21 #    along with this program.  If not, see <http://www.gnu.org/licenses/>.
22 #-----------------------------------------
23 #
24 # The following code is designed to be used
25 # with an Arduino Nano and the custom circuit
26 # designed and built by Aaron Jones, 4th Yr MSci
27 # Physics Student, University of Birmingham.
28 #
29 # There are two modules. The first module is called
30 # the user module and runs on a computer
31 # connected to the Arduino and allows the setting
32 # of variables and output of performace data.
33 # This is the second module called the control module
34 # and that runs on the Arduino actually controllingDominic
35 # the temprature.
36 #
37 # Concept
38 # The concept is that the Arduino controls
39 # the temprature using a 16 Bit ADC using an I2C
40 # link. Proccess this infomation in this script,
41 # and then outputs this to a DAC back over the I2C
42 # link.  The voltage from the DAC is proccessed by a
43 # analouge circuit to control a peltier junction.
44 #
45 # A report was written on this program
46 # as part of a 4th Yr Project. This explains
```

```
47 # the method in more detail
48 # and contains circuit diagrams.
49 #
50 # Dependancies
51 # - User_Interface.py
52 # - Control Module (This one)
53 # - Wire libary
54 #
55 # Acknowlagements
56 # The code used to access the timer2 function is from:
57 # http://forum.arduino.cc/index.php?topic=62964.0-
58 # The author is davekw7x
59 #
60 # The code used for the 24 Bit ADC is modified from Iain MacIvers
61 # work in 2014 for the University of Birmingham, Cold Atoms,
62 # GG-TOP group studies
63 #
64 # The PID has been modified from http://playground.arduino.cc/Code/PIDLibrary
65 # and is written by Brett Beauregard, contact: br3ttb@gmail.com
66 # Because it is under a stricter licence, it is stored sepertley
67 # The software can be used with an alternative algorithm under the LGPL
68 # or the PID under the fuller GPL
69
70 # Command Characters
71 # The following characters are used for sending commands
72 # between the two programs
73 # - m -> Message, the next line will be a message to the user
74 # - h -> Handshake, please acknowledge that the script is running
75 # - d -> Data
76 #
77 # Constants
78 # In this section there are a number of user tuneable constants
79 # These are globally accessible and called when required
80 # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
# */
/* ------ Common Constants ------- */
// After each reset, the following constants
// will be set by the Arduino. These can be
// overridden by a connected PC, but the change
// will not be permentant until set here

// Set Temprature
// --------------
// The default set temprature for the Arduino
// in degrees celius
// Typical Value = 25;
const double DEFAULT_SET_TEMPRATURE = 25;

// Desired Accuracy
// --------------
// The accuracy with which the temprature must be stable to.
// If inside the range
// (SETPOINT - ACCURACY) < Temp < (SETPOINT + ACCURACY)
// the system will be regarded as stable and the stable
// led will light. Upon exiting this range the user will
// be informed
// Typical Value = 0.01;
const double ACCURACY = 0.01;

// Proportional Constant
// --------------
// Typical Value = 4;
const double DEFAULT_PROPORTIONAL_CONSTANT = 2;

// Intergral Constant
// --------------
// Due to the high sample frequency this typically has a low value
// Typical Value = 0.001;
```

```
115 const double DEFAULT_INTERGRAL_CONSTANT = 0.001;
116
117 // Differntial Constant
118 // --------------
119 // Due to the high sample frequency this typically has a low value
120 // Typical Value = 0.1;
121 const double DEFAULT_DIFFERNTIAL_CONSTANT = 0.0;
122
123 /* ------- Less Common Constants ----- */
124 // These are constants which are tuneable, but typically
125 // do not vary in every implemtation of the system
126 // They are still tuneable
127
128 // 16 or 24 Bit
129 // ------------------------
130 // There are two potential options for tihs circuit
131 // using a 24 bit ADC or a 12 bit ADC.
132 // Set to true if using the 24 bit system.
133 // Typical Value = false
134 const bool TWENTY_FOUR_BIT = false;
135
136 // The value of Celsius in kelvin
137 // -----------------------------
138 // The PID actually works in kelvin and converts to
139 // celsius on reciept and transmission of data
140 const double CELSIUS = 273.15;
141
142 // Average Time
143 // --------------
144 // The time (in milliseconds) over which to peform a rolling average
145 // of the temprature data from the sensor.
146 // This is a method of rejecting electrical noise
147 // Note: 5 <= AVERAGE_TIME <= 1275, and it must be a multiple of 5
148 // Typical Value = 250;
```

```
149 const int AVERAGE_TIME = 500;
150
151 // Report Frequency
152 // --------------
153 // Please specify a time in milliseconds between
154 // the the Arduino sending data to the PC
155 // If REPORT_FREQUENCY_TIME = 0 The Arduino will output
156 // after every measurement
157 // Typical Value = 500;
158 const int REPORT_FREQUENCY_TIME = 500;
159
160 // Stable Count
161 // --------------
162 // The number of successive 'good' measurements
163 // before we can consider the system stable. Should be
164 // at least twice the value of the (average time/5)
165 // 1 second is approx 200
166 // Typical Value = 100;
167 const int STABLE_COUNT = 400;
168
169 // Flash Frequency
170 // --------------
171 // Pin 13 flashes after every X interations of
172 // the stabilisation loop. Ie if X = 1
173 // then every time a new tempratue value is
174 // read the LED will change state (e.g. on -> off)
175 // This helps to detect if the program has crashed
176 // Typical Value = 1; (with ossiloscope)
177 // Typical Value = 100 (without scope)
178 const int FLASH_FREQUENCY = 200;
179 /*
180 # ================================================
181 */
182 // #### Header #####
```

```
183 // ==== Dependancies =====
184 #include <Wire.h>
185 // ==== Class Declarations =====
186 // ---- USB Class ----
187 class io
188 {
189   private:
190     int mode;  // mode = 0: no pc connected
191                // mode = 1: PC connected, user script not running
192                // mode = 2: PC connected, user script running, output debug messages only
193                // mode = 3: supress all output
194                // mode = 4: output data and debug in human readable format
195                // mode = 99: connection yet to be determined
196   void clearin();
197
198   public:
199     void ioSetup();
200     void sendCommand(char command);
201     void printMessage(String message);
202     void sendData(double data);
203     boolean openPartMessage();
204     //void printPartMessage(String message);
205     void closePartMessage(); // Disable this functionallity
206     char readchar();
207     double readfloat();
208     int readint();
209     char checkmsg();
210     void outMode();
211     void checkPC();
212     void Connect();
213     void updateUser(double tempC, double Vout_Signed, double Vin);
214 };
215
216 // ---- DAC Class ----
```

```
217 class DAC
218 {
219   private:
220     int address;
221     int output(uint8_t command, double& value);
222   public:
223     DAC(int addressIN);
224     int Dsetup();
225     int output_A(double value);
226     int output_B(double value);
227     int output_both(double value);
228 
229 };
230 
231 // ---- 16 Bit ADC Class ----
232 class myADC  //The global namespace has been poluted by wire.h and consequently ADC has been used
        elsewhere
233 {
234   private:
235     int address;
236     boolean singleshot;
237     unsigned int configAll;
238     unsigned int configA0;
239     unsigned int configA1;
240     unsigned int configA2;
241     unsigned int configA3;
242   public:
243     myADC(int addressIN,boolean single);
244     double readADC(int port);
245 };
246 
247 class ADC24 // Class for 24 bit ADC
248 {
249   private:
```

```
250     byte PSCLK; // Bytes to hold the pins for the clock and data
251     byte PDATA;
252
253     void AwaitPulse();  //Functions to control timings
254     void AwaitAdc();    // for the ADC
255
256   public:
257     ADC24(byte SCLKpin, byte dataPin);
258     void ResetAdc();
259     float Update();
260
261 };
262
263 // ==== Function Declarations =====
264 double volt2Temp(double voltage, double fiveV, int arraySize, double temps[]);
265 void D11clock();
266 void constantError(String message);
267
268 void setup()
269 {
270     Serial.begin(9600); // Setup serial communiction
271     Serial.println("mArduino staring up..."); //Inform user
272     Serial.println("Checking Constants");
273     {
274         if (AVERAGE_TIME < 5)
275            {constantError("The averaging time must be a multiple of 5 AND must be >5");}
276         float f_av = float(AVERAGE_TIME);
277         if (AVERAGE_TIME/5 != f_av/5.0)
278           {constantError("The averaging time must be a multiple of 5 AND must be >5");}
279     }
280     if(TWENTY_FOUR_BIT) {D11clock();}  // Setup a hardware clock output on pin 11
281     Serial.println("m Entering main function");   // Inform user
282 }
283
```

```
284 // #### Function Defonitions ####
285 // ==== Main Program =====
286 void loop()
287 {
288   // ####  Setup ####
289   // ==== Call Functions ====
290   Wire.begin(); //Join I2C bus as master
291
292   // ==== Set up classes ====
293   // ---- IO ----
294   io ser;    // Open class for IO communication via serial
295   ser.ioSetup();  // move initialise into separate function (lets you re-initialise
296               // without recreating variables
297   ser.printMessage("Setting up variables...");
298
299   // ---- DAC ----
300   DAC dac(0x0F);    //Set up a class for the DAC
301   dac.Dsetup();
302
303   // ---- ADC's ----
304   myADC adc(0x48,false);    // Set up class for 16 Bit ADC, no continous conversion not required
305   ADC24 myLaser(4,3);      // Delcare the ADC24 out here so it is visible
306
307   if (TWENTY_FOUR_BIT) {myLaser.ResetAdc();}      // Reser ADC if it will be used this time
308
309   // ---- PID ----
310   //Define the PID variables
311   double Setpoint, Input, Output;
312
313   double accuracy = ACCURACY; // Load the default accuracy value
314   boolean stable = false;
315
316   //Specify the links and initial tuning parameters (split over two lines)
317   PID myPID(&Input, &Output, &Setpoint,
```

```
318             DEFAULT_PROPORTIONAL_CONSTANT,
319             DEFAULT_INTERGRAL_CONSTANT,
320             DEFAULT_DIFFERNTIAL_CONSTANT,
321             DIRECT);
322
323   //initialize the variables we're linked to
324   Setpoint = CELSIUS + DEFAULT_SET_TEMPRATURE;     // Set to 20 deg C
325   double pcSetpoint = Setpoint;
326   Input = Setpoint;     // Set Input = Setpoint so nothing will happen just yet
327
328   // Override some default values
329   myPID.SetOutputLimits(-3.5, 3.5);  // Set limits to 0 -> 3.5V, can send this straight to DAC
330
331   //turn the PID on
332   myPID.SetMode(AUTOMATIC);
333
334   // ==== Set Pins ===
335   const int HCpin = 12;     // The pin that controlls hot / cold
336   const int runningPin = 7;  //Pin to flash when running
337   const int stablePin = 6;
338   int runningLoop = 0;      //Loop counter to bring flash freq down to human readable
339   const int lockSetpoint = 2;
340
341   pinMode(HCpin, OUTPUT);  // Set pin to be an output
342   pinMode(runningPin,OUTPUT);
343   pinMode(stablePin, OUTPUT);
344   pinMode(lockSetpoint, INPUT);
345   digitalWrite(HCpin, HIGH);  // Set pin to high, it needs a value, I pick high
346   digitalWrite(runningPin,HIGH);  //Output the state to the pin, after this we will use port level
        IO for speed
347   digitalWrite(stablePin, stable); //Set the LED to the current stablility value
348
349   myPID.clearHistory();  // Clear the PID history to ensure no weirdness
350
```

```
351   ser.printMessage("Begin temprature control loop...");
352
353   while(true) // Loop forever
354   {
355     // Check that the PC is still there
356     ser.checkPC();
357     // Check if the PC wants to change a value
358     char pcVal = ser.checkmsg();
359     ser.outMode(); //If the PC is connected okay, illumenate the pin 13 LED, else off
360     switch (pcVal)
361     {
362       case 'v':
363       {            // Case v, the PC whishes to know the values of Kp, Ki, Kd and the Setpoint
364         if(ser.openPartMessage()) {
365           Serial.print("V=");
366           Serial.print(myPID.GetKp(),3);
367           Serial.print(','); //  csv delimited
368           Serial.print(myPID.GetKi(),3);  //repete
369           Serial.print(','); //  csv delimited
370           Serial.print(myPID.GetKd(),3);
371           Serial.print(','); //  csv delimited
372           Serial.print(Setpoint-CELSIUS,3);
373           Serial.print(',');
374           Serial.print(accuracy);
375           ser.closePartMessage();
376         }
377       }
378         break;
379
380       case 'V':                     //PC wishes to set new values of Kp, Ki, and Kd
381       {
382         float Kp = ser.readfloat();    //Read the new values
383         float Ki = ser.readfloat();
384         float Kd = ser.readfloat();
```

```
385         float SetIn = ser.readfloat();
386         float accuracyIn = ser.readfloat();
387         char endline = ser.readchar();  //Check character
388         if (endline == 'V')  // If check isnt okay, ignore
389         {
390           pcSetpoint = SetIn + CELSIUS;  //otherwise set the new values, SetPiont is passed by
                pointer, so just set the new value
391           myPID.SetTunings(Kp, Ki, Kd);  //Use function to set new tunings
392           accuracy = accuracyIn;
393         }
394       }
395         break;
396     }
397
398     //Serial.println("m PC Checks Complete");
399
400     // Check Set Point
401     if ((PIND & 0b00000100) == 0b00000100 ) {
402        Setpoint = pcSetpoint + (5.0/1024.0)*analogRead(2)/*adc.readADC(1)*/ - 2.5;
403     } else {
404        Setpoint = pcSetpoint;
405     }
406
407     // Read ADC's
408     double voltage;
409     double fiveV = adc.readADC(2);
410     if(TWENTY_FOUR_BIT)
411     {
412       double Vref = adc.readADC(3);
413       double adcOut = myLaser.Update();
414       double Vmin = Vref;
415       voltage = Vmin + Vref*adcOut;
416       /*Serial.print("m");
417       Serial.print(fiveV);
```

```
418       Serial.print(";");
419       Serial.print(Vref);
420       Serial.print(":");
421       Serial.print(voltage);
422       Serial.print(";");
423       Serial.println(Input);*/
424     }
425     else {
426       voltage = adc.readADC(0);
427     }
428
429     //Serial.println("m Voltage Read");
430
431     // Convert to temprature and average
432     const int averageLenght = AVERAGE_TIME/5; //Number of tempratures to average over
433     static double allTemps[averageLenght];  //array to hold temprature value to be averged over
434
435     Input = volt2Temp(voltage,fiveV,averageLenght,allTemps); // Pass the 5V value to this function,
          if using constant current just pass a zero, its not required
436
437     // Compute the new PID parameters
438     myPID.Compute();
439
440     double dac_A = Output;
441     if (Output > 0) { digitalWrite(HCpin, HIGH); dac_A = Output;}
442     else {digitalWrite(HCpin, LOW); dac_A = 0.0-Output;}
443
444     // output to DACS
445     dac.output_A(dac_A);
446
447     //Flash LED to show that the program is looping the loop counter is to allow some gearing
448     runningLoop++;
449     if(runningLoop == FLASH_FREQUENCY)
450     {
```

```
451       PORTD ^= (1 << runningPin);
452       // This is using port level IO
453       // To toggle a pin otherwise uses two lines of code
454       // and looks a bit ugly.
455       // The command is ^ is an exclusive OR
456       // (1 << X) means bit shift
457       // 0000 0001 left by X places. The effect is a toggle
458       // Search toggle a bit c++
459       // PORTD is the register that outputs to the pins
460       // This is 80x faster than digitalWrite
461       runningLoop = 0;
462     }
463
464     // Print data - Only print every second
465     // The rpi is pretty slow and even then serial isnt so quick
466     static unsigned long int timelast = 0;
467     unsigned long int timenow = millis();
468
469
470     /* Stability Monitor, this indicated to the user whether the system has exceeded
471     its allowed temprature range. I.e. If accuracy 1mk , setpoint is 21 and
472     input = 21.002 the the system is unstable, otherwise its stable
473     */
474     if (stable){  //If the system is supposdly stable then check this
475        if( (Input > Setpoint + accuracy) ||
476               (Input < Setpoint - accuracy )) { // If its not stable then:
477           stable = false;                          // No longer stable :(
478           digitalWrite(stablePin, stable);         // output to LED
479           ser.printMessage("Accuracy exceeded!");  // Inform user
480           ser.sendData(Input - CELSIUS);           // Output data
481           timelast = timenow;                      // Reset timelast, we have just updated the user
482        } // No need for an else statement, if we think its stable and it is stable then we can
483        // just move on
484
```

```
485     } else { // If the system is supposdly unstable, the first we check
486        static int counter = 0;
487        if( (Input > Setpoint + accuracy) ||
488               (Input < Setpoint - accuracy )) { // If its still unstable then reset the stable
                   counter:
489            counter = 0;
490        }
491        else { counter++; }  //We dont want a handful of good readings to trick the user into
               thiking this is stable
492                              // so keep track of the number of stable readings
493
494        if (counter >= STABLE_COUNT){  //If we have had lots of stable readings then the system is
               stable and we can:
495          //Toggle LED
496          ser.printMessage("Stability Regained!");  // Inform user
497          timelast = 0;    // Send all the data
498          stable = true;
499          digitalWrite(stablePin, stable);          // output to LED
500        }
501     }
502
503     if (timenow > timelast + REPORT_FREQUENCY_TIME)
504     {
505       ser.sendData(Input - CELSIUS);
506       ser.updateUser((Input-CELSIUS), Output, voltage);
507
508       timelast = timenow;
509     }
510   }
511   // End of loop
512 }
513
514 // ==== Voltage to Temprature =====
515 double volt2Temp(double voltage, double fiveV, int arraySize, double temps[])
```

```
516 {
517
518   // First Calculate a resitance from a voltage
519
520   // === Using a Simple Potentiometer ====
521
522   const double Rt = 10e3; // Top resistor resistance
523   double resistance = Rt/((fiveV/voltage) - 1.0);  // This is just the potential divider formula
          re-arranged
524
525   // === Or Using a Constant Current Source ====
526   //const double current = 66.74e-6; // Constant Current, this is a little dodgy, my current seemed
          to vary a little near the limits of operation
527   //double resistance = voltage/current;  // V=IR
528
529   // Next calculate a temprature (in Kelvin!) from a voltage
530   const double R25 = 10000; // Resistance at 25 Deg C (298.15 Deg K)
531   const double constTerm = 3.3540170e-3;
532   double linearTerm = (2.5617244e-4)*log(pow(resistance/R25,1));
533   double quadraticTerm = (2.1400943e-6)*log(pow((resistance/R25),2));
534   double cubicTerm = (-7.2405219e-8)*log(pow((resistance/R25),3));
535
536   double tNow = 1/(constTerm+linearTerm+quadraticTerm+cubicTerm);
537
538  //Now do a N-Point moving average to eliminate some noise
539   static int arrayIndex = 0;  //Point to the element in the array to be updated
540   static bool firstCall = true;  //Just for array initislisation
541   //If this is the first time then let all the variables have this temprature
542   //and initilise the array
543   if (firstCall)
544   {
545      int i = 0;
546      while ( i < arraySize)
547      {
```

```
548         temps[i] = tNow;
549         i++;
550       }
551       firstCall = false;
552   }
553   else
554   {
555     if(arrayIndex == arraySize) arrayIndex = 0;
556
557     temps[arrayIndex]=tNow;
558     arrayIndex++;
559   }
560   double tempSum = 0;
561   for (int i = 0; i<arraySize; i++) { tempSum = tempSum + temps[i];}
562
563   return (1.0/arraySize)*tempSum;
564 }
565
566 // This function is modified from davekw7x, see the acknowlegemets section for details
567 // Use of timer2 to generate a signal for a particular frequency on pin 11
568 void D11clock()
569 {
570     // Give the pin connected to the 0C2A comparitor register a name
571     const int freqOutputPin = 11;   // OC2A output pin for ATmega328 boards
572     //const int freqOutputPin = 10; // OC2A output for Mega boards
573
574     // Constants are computed at compile time
575
576     // If you change the prescale value, it affects CS22, CS21, and CS20
577     // For a given prescale value, the eight-bit number that you
578     // load into OCR2A determines the frequency according to the
579     // following formulas:
580     //
581     // With no prescaling, an ocr2val of 3 causes the output pin to
```

```
582     // toggle the value every four CPU clock cycles. That is, the
583     // period is equal to eight slock cycles.
584     //
585     // With F_CPU = 16 MHz, the result is 2 MHz.
586     //
587     // Note that the prescale value is just for printing; changing it here
588     // does not change the clock division ratio for the timer!  To change
589     // the timer prescale division, use different bits for CS22:0 below
590     const int prescale  = 1;
591     const int ocr2aval  = 3;
592     // The following are scaled for convenient printing
593     //
594
595     // Period in microseconds
596     const float period    = 2.0 * prescale * (ocr2aval+1) / (F_CPU/1.0e6);
597
598     // Frequency in Hz
599     const float freq      = 1.0e6 / period;
600
601     pinMode(freqOutputPin, OUTPUT);
602
603     // Set Timer 2 CTC mode with no prescaling.  OC2A toggles on compare match
604     //
605     // WGM22:0 = 010: CTC Mode, toggle OC
606     // WGM2 bits 1 and 0 are in TCCR2A,
607     // WGM2 bit 2 is in TCCR2B
608     // COM2A0 sets OC2A (arduino pin 11 on Uno or Duemilanove) to toggle on compare match
609     //
610     TCCR2A = ((1 << WGM21) | (1 << COM2A0));
611
612     // Set Timer 2  No prescaling  (i.e. prescale division = 1)
613     //
614     // CS22:0 = 001: Use CPU clock with no prescaling
615     // CS2 bits 2:0 are all in TCCR2B
```

```
616     TCCR2B = (1 << CS20);
617
618     // Make sure Compare-match register A interrupt for timer2 is disabled
619     TIMSK2 = 0;
620     // This value determines the output frequency
621     OCR2A = ocr2aval;
622
623     Serial.println("m Outputting on pin D11 with");
624     Serial.print("mPeriod    = ");
625     Serial.print(period);
626     Serial.println(" microseconds");
627     Serial.print("mFrequency = ");
628     Serial.print(freq);
629     Serial.println(" Hz");
630 }
631
632
633 void constantError(String message)
634 {
635    while(true)
636    {
637        Serial.println('m' + message);
638        Serial.println(
639          "m The Arduino will not run untill recompiled with the correct constants");
640        delay(5000);
641    }
642 }
643
644
645 // ##### Define Classes ######
646 // ==== Define IO Class =====
647 void io::ioSetup()
648 {
649   pinMode(13, OUTPUT);
```

```
650   //Serial.begin(9600); This is called in setup
651   clearin();     // clear anything waiting on the buffer
652   Connect();
653 }
654 void io::Connect()
655 {
656   sendCommand('h');  // Send hello/handshake command
657   delay(50);
658   // dont use the readchar function it might not return
659   char rv = Serial.read();
660   if (rv == 'h') {mode =2;}
661   else { mode = 1;}
662 }
663 // --- Write functions ---
664 void io::sendCommand(char command) {Serial.flush(); Serial.println(command);}
665 //print command and new line
666 void io::printMessage(String message) {if(mode==2) {Serial.flush(); Serial.print('m');
      Serial.println(message); }}
667 // print message character followed by the message and a new line
668 void io::sendData(double data) {if(mode==2) {Serial.flush(); Serial.print('d');
      Serial.print(data,8); Serial.println(); }}
669 //Sending floats at accuracies < 1e-8 causes problems so we will round off here to avoid any
      problems
670 void io::updateUser(double tempC, double Vout_Signed, double Vin) {
671   if(mode==2) {
672     Serial.flush();  //Wait for any output to finish
673     Serial.print('m');  //Signal message for screen
674     Serial.print("T=");  //Write first few characters
675     Serial.print(tempC,3);  //Output temprature to 3dp
676     Serial.print("C DA: ");
677     Serial.print(Vout_Signed,3);  //Output voltage out
678     Serial.print(" Ia:");
679     double I = (Vout_Signed/2.11)/0.9;
680     // The current is the (voltage/amplification_of_signal)/Resitance(0.7 Ohms + 0.2 track on PCB)
```

```
681     Serial.print(I,3);  //Output current to 3dp
682     Serial.println();
683   }
684 }
685 boolean io::openPartMessage() {if (mode==2) {Serial.print('m'); return true;} else {return false;}}
686 // start new line with message character
687 //void io::printPartMessage(String message) {Serial.print(message);}
688 //print message
689 void io::closePartMessage() {Serial.print('\n');}
690 //print new line
691
692 // --- Read functions ---
693 double io::readfloat() {float f; if(!Serial.available()) {delay(10);} Serial.readBytes((char*)&f,
        sizeof(f)); return f;}
694 // declare float, wait until data becomes available then read it from USB
695 int io::readint() { if(!Serial.available()) {delay(10);} return Serial.read();}
696 // wait for data to become available the return and cast to integer
697 char io::readchar() { if(!Serial.available()) {delay(10);} return Serial.read();}
698 // wait for data to become available the return and cast to integer
699 void io::clearin() { while(Serial.available()) {Serial.read();}}
700 void io::checkPC()
701 {
702    static int timeLast = 0;
703    static int timeNow = 0;
704    timeNow = millis();
705    if ((timeNow - timeLast)>5000)  //Run every 5 seconds
706    {
707      Connect();
708      timeLast = timeNow;
709    }
710 }
711 char io::checkmsg()
712 {
713   if(Serial.available())
```

```
714   {
715     char val = readchar();
716     if (val == 'h') {mode = 2; return 'n';}
717     else return val;
718   }
719   else return 'n';
720 }
721
722 void io::outMode()
723 {
724   if(mode == 2){ digitalWrite(13,HIGH);}
725   else { digitalWrite(13,LOW);}
726 }
727
728 // ==== Define DAC Class =====
729 DAC::DAC(int addressIN) { address = addressIN;}
730
731 // Setup function
732 int DAC::Dsetup()
733 {
734    int tranmission_status = 0;
735
736    //Reset the DAC registers
737    Wire.beginTransmission((uint8_t)address);
738     Wire.write((uint8_t) 0x28);  //write reset commmand
739     Wire.write((uint8_t) 0x00);
740     Wire.write((uint8_t) 0x01);
741   tranmission_status = Wire.endTransmission();
742   if (tranmission_status != 0) {return tranmission_status;}
743
744   //Disable LDAC -- The LDAC pin is grounded so it shouldnt matter anyway
745   Wire.beginTransmission((uint8_t)address);
746     Wire.write((uint8_t) 0x30);
747     Wire.write((uint8_t) 0x00);
```

```
748     Wire.write((uint8_t) 0x03);
749   tranmission_status = Wire.endTransmission();
750   if (tranmission_status != 0) {return tranmission_status;}
751
752   //Enable internal referance - Disabled Awaiting Spares
753   /*Wire.beginTransmission((uint8_t)address);
754     Wire.write((uint8_t) 0x38);
755     Wire.write((uint8_t) 0x00);
756     Wire.write((uint8_t) 0x01);
757   tranmission_status = Wire.endTransmission();*/
758
759   return tranmission_status;
760 }
761
762 int DAC::output(uint8_t command, double& value)  // value should be between 0 and 5
763 {
764   const unsigned int DAC_bits = 0b1111111111111111;
765   unsigned int binary_out = (value/5.0)*DAC_bits;  //Convert to binary, since using an internal
        referance do this conversion here
766
767   int tranmission_status;
768   Wire.beginTransmission((uint8_t)address);
769     Wire.write(command);  //write to config register
770     Wire.write((uint8_t)(binary_out>>8));  //write first 8 bits (MSB)
771     Wire.write((uint8_t)(binary_out & 0xFF));  //write to second 8 bits (LSB)
772   tranmission_status = Wire.endTransmission();
773   return tranmission_status;
774 }
775
776 int DAC::output_A(double value) {return output((uint8_t)0b00000000, value);}
777 int DAC::output_B(double value) {return output((uint8_t)0b00000001, value);}
778 int DAC::output_both(double value){return output((uint8_t)0b00000111, value);}
779
780 // ==== Define myADC Class =====
```

```
781 myADC::myADC(int addressIN, boolean single)
782 {
783
784   address = addressIN;
785   singleshot = single;
786   if(single)
787   {
788     // Setup single shot conversion
789     configAll = 0x0003 | // Disable the comparator (default val)
790                          0x0000 | // Non-latching comparator (default val)
791                          0x0000 | // Alert/Rdy active low (default val)
792                          0x0000 | // Traditional comparator (default val)
793                          //0x0080 | // 128 samples per second (default) (8ms Conversion time)
794                          0x00E0 | // 860 Samples per second (1ms conversion time)
795                          0x0100 | // Single-shot mode (default)
796                          0x0000 |  // Set PGA/voltage range to 2/3 gain (full range of input)
797                          0x8000;  //Signal ADC to start conversion
798
799   }
800   else
801   {
802     //Setup contonous conversion
803     configAll = 0x0003 | // Disable the comparator (default val)
804                          0x0000 | // Non-latching comparator (default val)
805                          0x0000 | // Alert/Rdy active low (default val)
806                          0x0000 | // Traditional comparator (default val)
807                          // 0x0080 | // 1600 samples per second (default) (8ms conversion time)
808                          0x00E0 | // 860 Samples per second (1 ms conversion time)
809                          0x0000 | // Continous Conversion
810                          0x0000 |  // Set PGA/voltage range to 2/3 gain (full range of input)
811                          0x8000;  //Signal ADC to start conversion
812   }
813     configA0 = configAll | 0x4000; //Add A0 address
814     configA1 = configAll | 0x5000; //Add A1 address
```

```
815     configA2 = configAll | 0x6000; //Add A2 address
816     configA3 = configAll | 0x7000; //Add A3 address
817 }
818
819
820 double myADC::readADC(int port)  // value should be between 0 and 1
821 {
822   unsigned int config_out; // variable to hold configuration to be output
823
824   static int lastPort = 4;  // variable to hold the lastPort, set to 4 so the next line will
        evalulate false on the first loop
825
826   // If this the config register has not been changed, dont bother updating it
827   if (lastPort != port)
828   {
829     switch (port)
830     {
831        case 0:
832           config_out = configA0;
833           break;
834        case 1:
835           config_out = configA1;
836           break;
837        case 2:
838           config_out = configA2;
839           break;
840        case 3:
841           config_out = configA3;
842           break;
843     }
844
845     Wire.beginTransmission(address);
846       Wire.write((uint8_t) 0x01);  //write to config register
847       Wire.write((uint8_t)(config_out>>8));  //write first 8 bits (MSB)
```

```
848         Wire.write((uint8_t)(config_out & 0xFF));  //write to second 8 bits (LSB)
849       Wire.endTransmission();
850     }
851
852
853     // Request reading of data
854     if((singleshot) || (lastPort != port)){delayMicroseconds(3000);}  // Wait 1.2ms for conversion to
            complete (if in continous this step is not required)
855
856     Wire.beginTransmission(address);  //Open transmission to ADC
857       Wire.write(0x00);             //Select register to read from
858     Wire.endTransmission();
859
860     // request data
861     Wire.requestFrom((int)address,2); // Request 2 bytes
862     double response = ((Wire.read() << 8) | Wire.read());
863
864     // Convert to Voltage
865     const float PGA = 6.144; // Set PGA range for volatage conversion - //Since using internal ref
            deal with this here
866     const int ADC_bits = 0b0111111111111111;  // Set number of bits used
867     return PGA*((1.0*response)/(1.0*ADC_bits));
868
869     //Set last Port for next loop
870     lastPort = port;
871 }
872
873 // ==== Define ADC24 Class =====
874 ADC24::ADC24(byte SCLKpin, byte dataPin)
875 {
876     // Bytes to hold the pins for the clock and data
877       PSCLK = SCLKpin;
878       PDATA = dataPin;
879
```

```
880   // Setup pins for output
881     pinMode(PSCLK, OUTPUT);
882     pinMode(PDATA, INPUT);
883 }
884
885 // With SCLCK left low, the ADC will only pulse when data is ready
886 // in this case it will low, then resume high
887 void ADC24::AwaitPulse() {
888   while(!(PIND&(1<<PDATA)));
889   while(PIND&(1<<PDATA));
890 }
891
892 // Wait the ADC to complete at least one cycle
893 void ADC24::AwaitAdc() {
894   AwaitPulse();
895   AwaitPulse();
896 }
897
898 // To reset the ADC pull the SCLK high for 5 data periods
899 void ADC24::ResetAdc()
900 {
901   PORTD |= (1 << PSCLK);
902   delayMicroseconds(1440);
903   PORTD &= 0b11111111^(1<<PSCLK);
904 }
905
906 float ADC24::Update()
907 {
908   // Setup variables
909   long int inputVar = 0;  //result variable
910   long int mask = 1L<<23;  //mask to add data (write int 1, force to be long, then bitshift by 23)
911
912   AwaitAdc(); // Wait for ADC to become ready
913   delayMicroseconds(2);
```

```
914
915   // Loop through and recieve each bit
916   for(int i = 0; i < 24; i++)
917   {
918     PORTD |= (1 << PSCLK);  //Toggle SCLK high to send new bit
919     delayMicroseconds(2);  // Allow line to settle
920     inputVar |= (PIND&(1<<PDATA))?mask:0;  // OR assignment, (if new bit is high, OR with mask,
          else 0)
921     delayMicroseconds(2);  // Allow line to settle
922     PORTD &= 0b11111111^(1<<PSCLK);  //Return SCLK low
923     mask = mask>>1;  // Decrement
924   }
925
926   // Declare number to convert 0 - 2^23 integer -> 0-1 decimal
927   const float divisor = 8388607;  // (2^23 - 1) but you cannot store it like that
928
929   // Identify if negative (two's compliment)
930   if((1L<<23) & inputVar) //Mask to identify if the number is negative
931   {
932      inputVar = ~inputVar;  // Invert all the bits
933      inputVar = inputVar + 1;  // Add one
934      inputVar &= 0x007fffff; // Set bits 23 - 31
935      // This is required because we are dealing with a 24 bit number stored in a 32 bit integer
936
937      // We now have a positive number and can return it as normal as long as we make it negative
938      return -((float)inputVar)/divisor;
939   }
940   else {return ((float)inputVar)/divisor;}
941 }
```

The algorithm used

This was the algorithm used.

```
1 /*#=========================================
```

```
#~~~~~  PID Library ~~~~~~~
#------------------------------------------
# Interface Code
# V1.0.1
# Author - Brett Beauregard
# Date: 12/2012
# Copyright: Brett Beauregard, 2012, br3ttb@gmail.com
#
# Modified for use with the Arduino Temprature Controller
# by Aaron Jones, 2015
#------------------------------------------
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ------------------------------------------*/


// ---- PID Class ----
class PID
{
  public:

  //Constants used in some of the functions below
  #define AUTOMATIC 1
```

```
36  #define MANUAL  0
37  #define DIRECT  0
38  #define REVERSE  1
39
40  //commonly used functions
        **************************************************************************
41    PID(double*, double*, double*,        // * constructor.  links the PID to the Input, Output, and
42        double, double, double, int);     //   Setpoint.  Initial tuning parameters are also set
              here
43
44    void SetMode(int Mode);               // * sets PID to either Manual (0) or Auto (non-0)
45
46    bool Compute();                       // * performs the PID calculation.  it should be
47                                          //   called every time loop() cycles. ON/OFF and
48                                          //   calculation frequency can be set using SetMode
49                                          //   SetSampleTime respectively
50
51    void SetOutputLimits(double, double); //clamps the output to a specific range. 0-255 by
          default, but
52            //it's likely the user will want to change this depending on
53            //the application
54
55  //available but not commonly used functions
        *******************************************************
56    void SetTunings(double, double,       // * While most users will set the tunings once in the
57                    double);            //   constructor, this function gives the user the option
58                                          //   of changing tunings during runtime for Adaptive
                                                control
59    void SetControllerDirection(int);   // * Sets the Direction, or "Action" of the controller.
          DIRECT
60            //   means the output will increase when error is positive. REVERSE
61            //   means the opposite.  it's very unlikely that this will be needed
62            //   once it is set in the constructor.
63    void clearHistory();                  // Added by AJ on 15/12/2014, clears history by setting
```

```
        ITerm = 0;
64
65  //Display functions ****************************************************************
66  double GetKp();             // These functions query the pid for interal values.
67  double GetKi();             //  they were created mainly for the pid front-end,
68  double GetKd();             // where it's important to know what is actually
69  int GetMode();              //  inside the PID.
70  int GetDirection();           //
71
72  private:
73  void Initialize();
74
75  double dispKp;        // * we'll hold on to the tuning parameters in user-entered
76  double dispKi;        //   format for display purposes
77  double dispKd;        //
78
79  double kp;                  // * (P)roportional Tuning Parameter
80        double ki;                  // * (I)ntegral Tuning Parameter
81        double kd;                  // * (D)erivative Tuning Parameter
82
83  int controllerDirection;
84
85        double *myInput;              // * Pointers to the Input, Output, and Setpoint variables
86        double *myOutput;             //   This creates a hard link between the variables and the
87        double *mySetpoint;           //   PID, freeing the user from having to constantly tell us
88                                      //   what these values are.  with pointers we'll just know.
89
90        double ITerm, lastInput;
91
92        double outMin, outMax;
93        bool inAuto;
94 };
95
96 // ==== Define PID Class =====
```

```
97  PID::PID(double* Input, double* Output, double* Setpoint,
98          double Kp, double Ki, double Kd, int ControllerDirection)
99  {
100
101     myOutput = Output;
102     myInput = Input;
103     mySetpoint = Setpoint;
104   inAuto = false;
105
106     PID::SetOutputLimits(0, 255);       //default output limit corresponds to
107                 //the arduino pwm limits
108
109     PID::SetControllerDirection(ControllerDirection);
110     PID::SetTunings(Kp, Ki, Kd);
111
112 }
113
114
115 /* Compute() **********************************************************************
116  *     This, as they say, is where the magic happens.  this function should be called
117  *   every time "void loop()" executes.  the function will decide for itself whether a new
118  *   pid Output needs to be computed.  returns true when the output is computed,
119  *   false when nothing has been done.
120  **********************************************************************************/
121 bool PID::Compute()
122 {
123    if(!inAuto) return false;
124    //Serial.println("Actually ran!");
125
126     /*Compute all the working error variables*/
127     double input = *myInput;
128     double error = *mySetpoint - input;
129     ITerm+= (ki * error);
130     if(ITerm > outMax) ITerm= outMax;
```

```
131     else if(ITerm < outMin) ITerm= outMin;
132     double dInput = (input - lastInput);
133
134     /* Serial.println(error);
135      Serial.println(kp*error);
136      Serial.println(ITerm);
137      Serial.println(kd*dInput);*/
138
139     /*Compute PID Output*/
140     double output = kp * error + ITerm- kd * dInput;
141      //Serial.println(output);
142     if(output > outMax) output = outMax;
143     else if(output < outMin) output = outMin;
144     *myOutput = output;
145
146     return true;
147 }
148
149
150 /* SetTunings(...)*************************************************************
151  * This function allows the controller's dynamic performance to be adjusted.
152  * it's called automatically from the constructor, but tunings can also
153  * be adjusted on the fly during normal operation
154  ******************************************************************************/
155 void PID::SetTunings(double Kp, double Ki, double Kd)
156 {
157    if (Kp<0 || Ki<0 || Kd<0) return;
158
159    dispKp = Kp; dispKi = Ki; dispKd = Kd;
160
161    kp = Kp;
162    ki = Ki;
163    kd = Kd;
164
```

```
165   if(controllerDirection ==REVERSE)
166    {
167       kp = (0 - kp);
168       ki = (0 - ki);
169       kd = (0 - kd);
170    }
171 }
172
173 /* SetOutputLimits(...)****************************************************
174  *     This function will be used far more often than SetInputLimits.  while
175  *  the input to the controller will generally be in the 0-1023 range (which is
176  *  the default already,)  the output will be a little different.  maybe they'll
177  *  be doing a time window and will need 0-8000 or something.  or maybe they'll
178  *  want to clamp it from 0-125.  who knows.  at any rate, that can all be done
179  *  here.
180  **************************************************************************/
181 void PID::SetOutputLimits(double Min, double Max)
182 {
183    if(Min >= Max) return;
184    outMin = Min;
185    outMax = Max;
186
187    if(inAuto)
188    {
189      if(*myOutput > outMax) *myOutput = outMax;
190      else if(*myOutput < outMin) *myOutput = outMin;
191
192      if(ITerm > outMax) ITerm= outMax;
193      else if(ITerm < outMin) ITerm= outMin;
194    }
195 }
196
197 /* SetMode(...)****************************************************************
198  * Allows the controller Mode to be set to manual (0) or Automatic (non-zero)
```

```
199  * when the transition from manual to auto occurs, the controller is
200  * automatically initialized
201  ******************************************************************************/
202 void PID::SetMode(int Mode)
203 {
204     bool newAuto = (Mode == AUTOMATIC);
205     if(newAuto == !inAuto)
206     {  /*we just went from manual to auto*/
207         PID::Initialize();
208     }
209     inAuto = newAuto;
210 }
211
212 /* Initialize()****************************************************************
213  *  does all the things that need to happen to ensure a bumpless transfer
214  *  from manual to automatic mode.
215  ******************************************************************************/
216 void PID::Initialize()
217 {
218    ITerm = *myOutput;
219    lastInput = *myInput;
220    if(ITerm > outMax) ITerm = outMax;
221    else if(ITerm < outMin) ITerm = outMin;
222 }
223
224 /* SetControllerDirection(...)*************************************************
225  * The PID will either be connected to a DIRECT acting process (+Output leads
226  * to +Input) or a REVERSE acting process(+Output leads to -Input.)  we need to
227  * know which one, because otherwise we may increase the output when we should
228  * be decreasing.  This is called from the constructor.
229  ******************************************************************************/
230 void PID::SetControllerDirection(int Direction)
231 {
232    if(inAuto && Direction !=controllerDirection)
```

```
233     {
234      kp = (0 - kp);
235        ki = (0 - ki);
236        kd = (0 - kd);
237     }
238     controllerDirection = Direction;
239 }
240 
241 /* Status Funcions*************************************************************
242  * Just because you set the Kp=-1 doesn't mean it actually happened.  these
243  * functions query the internal state of the PID.  they're here for display
244  * purposes.  this are the functions the PID Front-end uses for example
245  ******************************************************************************/
246 double PID::GetKp(){ return  dispKp; }
247 double PID::GetKi(){ return  dispKi;}
248 double PID::GetKd(){ return  dispKd;}
249 int PID::GetMode(){ return  inAuto ? AUTOMATIC : MANUAL;}
250 int PID::GetDirection(){ return controllerDirection;}
251 void PID::clearHistory() {ITerm = 0;}
```

H.1.2 User_Interface.py

This file contains the user interface.

```
1  #=============================================
2  #~~~~~  Arduino Temprature Controller ~~~~~~~
3  #---------------------------------------------
4  # Interface Code
5  # V1.0
6  # Author -  Aaron Jones
7  # Date: 25/03/2015
8  # Copyright Aaron Jones, 2015
9  #---------------------------------------------
10 #    This program is free software: you can redistribute it and/or modify
11 #    it under the terms of the GNU Lesser General Public License as published by
```

```
12 #    the Free Software Foundation, either version 3 of the License, or
13 #    (at your option) any later version.
14 #
15 #    This program is distributed in the hope that it will be useful,
16 #    but WITHOUT ANY WARRANTY; without even the implied warranty of
17 #    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18 #    GNU General Public License for more details.
19 #
20 #    You should have received a copy of the GNU Lesser General Public License
21 #    along with this program.  If not, see <http://www.gnu.org/licenses/>.
22 #
23 #    The inlcluded file GUI.py uses PyQt4 which is under the fuller GPL
24 #-------------------------------------------
25 # The following code is designed to be used
26 # with an Arduino Nano and the custom circuit
27 # designed and built by Aaron Jones, 4th Yr MSci
28 # Physics Student, University of Birmingham.
29 #
30 # There are two modules. The first module is called
31 # the user module and runs on a computer
32 # connected to the Arduino and allows the setting
33 # of variables and output of performace data.
34 # This is the second module called the control module
35 # and that runs on the Arduino actually controlling
36 # the temprature.
37 #
38 # Concept
39 # The concept is that the Arduino controls
40 # the temprature using a 16 Bit ADC using an I2C
41 # link. Proccess this infomation in this script,
42 # and then outputs this to a DAC back over the I2C
43 # link.  The voltage from the DAC is proccessed by a
44 # analouge circuit to control a peltier junction.
45 #
```

```
46 # A report was written on this program
47 # as part of a 4th Yr Project. This explains
48 # the method in more detail
49 # and contains circuit diagrams.
50 #
51 # Dependancies
52 # - User_Interface.py (This one)
53 # - Control Module (Arduino_TempController.ino)
54 # - Python Libaries as inported below
55 # - GUI.py
56 #
57 # Acknowlagements
58 # The code used to access the timer2 function is from:
59 # http://forum.arduino.cc/index.php?topic=62964.0-
60 # The author is davekw7x
61 #
62 # The code used for the 24 Bit ADC is modified from Iain MacIvers
63 # work in 2014 for the University of Birmingham, Cold Atoms,
64 # GG-TOP group studies
65 #
66 # The PID has been modified from http://playground.arduino.cc/Code/PIDLibrary
67 # and is written by Brett Beauregard, contact: br3ttb@gmail.com
68 #
69 # Command Characters
70 # The following characters are used for sending commands
71 # between the two programs
72 # - m -> Message, the next line will be a message to the user
73 # - h -> Handshake, please acknowledge that the script is running
74 # - d -> Data
75 #
76 # Constants
77 # In this section there are a number of user tuneable constants
78 # These are globally accessible and called when required
79 # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
#
# --------------------------
# User Editable Parameters
# --------------------------
#
#
# Serial Connection
# -----------------
# Please uncomment the port on which the Arduino is connected
# This can be found by opening the Arduino IDE and looking in the lower right corner
device_link = '/dev/ttyUSB0'  # Linux - Normally '/dev/ttyUSB0' or '/dev/ttyUSB1' etc
#device_link = 'COM7'   # Windows - Normally 'COM7'/'COM8' etc
#device_link = '/dev/pts/13'  # Linux - Imitation Arduino
#
# Buffer Length
#-----------------
# When data is read in from the Arudino, the last X
# values are stored in an array. This is the lenght of
# that array.
# Typical Value = 100
CONSTANT_BUFFER_LENGHT = 75
#
# Graph Length
#-----------------
# When a graph is called this is the number of points
# to plot on the X axis. This must be less than
# or equal to the buffer lenght
# Typical Value = 75 (normally equal to buffer lenght)
CONSTANT_GRAPH_LENGHT = 75
#
# Check on buffer lenght
if (CONSTANT_BUFFER_LENGHT < CONSTANT_GRAPH_LENGHT):
  print "Please change buffer lenght to a value greater than"
  print "or equal to the graph lenght in the file computer.py"
```

```
114   raise SystemExit
115 #
116 #================================================
117
118
119 # ====================
120 # PreAmble
121 # ====================
122
123 #Import required libaries
124 import serial
125 import thread
126 import struct
127 import sys
128 import time
129 import numpy as np
130 import pylab
131 import matplotlib.pyplot as plt
132
133
134 # Set up globals and locks
135 user_last_response = ""
136 user_last_response_lock = thread.allocate_lock()
137
138
139 # ====================
140 # Definitions
141 # ====================
142
143 # ------------------
144 # Main Functions
145 # ------------------
146 def prelims():
147   # Welcome the user
```

```
148   print('=============================================')
149   print('UoB Digital Temprature Controller UI Program')
150   print('---------------------------------------------')
151   print('Author: Aaron Jones')
152   print 'Version ', version
153   print "NB The GUI for this application is only available"
154   print "under the full terms of the GPL."
155   print('=============================================')
156
157   while True:
158     isGUI = raw_input("\nWould you like to use the GUI for this session? (y/n):") #Ask user whether
          to use the GUI
159     if (isGUI == 'y'):
160       isGUI = True
161       return isGUI
162     elif (isGUI == 'n'):
163       isGUI = False
164       return isGUI
165     else:
166       print ":( Sorry I can only accept the characters 'y' or 'n'. Lets try again?"
167
168 # Between these functions three classes, arduino, user and temprature
169 # are globally defined and initilised in global scope
170
171 def main(isGUI):
172   global user
173   global arduino
174   global temprature
175
176   if(not(isGUI)):
177     # Call worker thread, program will exit when this exits
178     worker(False)
179   else:
180     print "Setting up GUI..."
```

```
181     import GUI
182     from PyQt4 import QtGui, QtCore
183     print "Imports Ok!"
184     app = QtGui.QApplication(sys.argv)  #Open QT Workspace
185     print "Qt running. Now drawing window..."
186     window = GUI.MainWindow(arduino,temprature,user)
187     print "Drawn! Calling worker thread..."
188     # Call a worker thread to handle the Arduino and Termainal requests
189     # The terminal is output only, so only certain functions will be called
190     # This reduces the number of locks required
191     GUI_Worker = GUI.Thread(worker)
192     GUI_Worker.start()
193     print "Displaying GUI in 3,2,1!"
194     sys.exit(app.exec_())
195
196 # -----------------
197 #  Functions
198 # -----------------
199 def worker(GUI):
200
201   # Import some global variables.
202   global user_last_response
203
204   global user
205   global arduino
206   global temprature
207
208   # Connect to the Arduino
209   arduino.connect()
210
211   if (GUI):
212     user_last_response_lock.acquire() # Aquire lock, blocking any input
213   else:
214     # spawn new thread to capture user input
```

```
215     thread.start_new_thread(user.captureInput,())
216
217   while (not(user_last_response_lock.locked())):
218     time.sleep(1e-9)  #Sleep for 1 nanosecond. (Not actually possible)
219
220   user.print2('Listening for data...')
221   # Enter a Loop
222   while(True):
223     sys.stdout.flush()
224     #Check to see if the user has requested control, is GUI open this will always be false
225     if (not(user_last_response_lock.locked())):
226       #Ask Arduino to go into silent mode
227       user.respond(user_last_response,temprature) # Call response function. No need to lock as
              currently single threaded
228       thread.start_new_thread(user.captureInput,()) # ... this will relock the variable untill the
              user is ready
229
230
231     # Read the next line from the Arduino
232     command, value = arduino.getLine()
233
234     # If blank, skip
235     if (command == 0):
236       time.sleep(1e-6)
237     # If it is a message just print it
238     elif (command == arduino.msg_char):
239       user.print2('Device says:')
240       user.print2(value)
241
242     # Else if it is data add it to the data
243     elif (command == arduino.data_char):
244       temprature.addData(time.time(),value)
245       if(not(GUI) and temprature.graph and arduino.bufferok()):
246         temprature.reDraw() # Only redaw if the buffer is okay
```

```
247         # Dont redraw if the GUI is open, let Qt handel that
248 
249     elif(command == arduino.handshake):
250       arduino.ser.write(arduino.handshake)
251 
252 def lotsChar(char,Num):
253   string = ''
254   for i in range(0,Num):
255     string = string + char
256   return string
257 # -----------------
258 # Classes
259 # -----------------
260 
261 # User Input Class
262 class USB:
263 # ------------------------------
264 # Date: 30/12/14, Last Modified - 30/12/14
265 # This class holds functions for interacting with the Arduino
266 # ------------------------------
267   def __init__(self, device_link): #Initilise class
268     print 'Opening connection on', device_link
269     try:
270       self.ser = serial.Serial(device_link, 9600)
271     except:   #print error message if this does not work
272       print '\n\n\n'
273       print 'No devices are available on ', device_link
274       print 'Please check update where the Arduino located'
275       print '(The Arduino IDE can do this, try Arduino IDE -> Tools -> Serial Port)'
276       print 'and update this file'
277       print '\n\n\n'
278       raise
279 
280     #Set up signal characters
```

```
281     self.handshake = 'h'
282     self.msg_char = 'm'
283     self.data_char = 'd'
284     self.getVars = 'v'
285     self.getConstantsRunning = False
286 
287   # Arduino Connect
288   def connect(self):
289   # This function waits untill a Arduino with the correct software running is connected
290     print "Attempting to find Arduino Temprature Controller"
291     print "\tTry resetting the Arduino if this doesn't work"
292     print "\tor 'ctrl + c' to exit the program"
293     loop = True
294     while(loop):
295       command = self.ser.read(1)  #wait until Arduino contacts
296       if (command == self.handshake):
297         self.ser.write(self.handshake)  #test for handshake and respond
298         while (self.ser.inWaiting() > 0):
299           self.ser.read(1)
300         loop = False  #exit loop
301         print "Found!"
302 
303   def getLine(self):
304     while(self.getConstantsRunning):  #Check the buffer isn't locked
305       time.sleep(1e-6)
306     if (self.ser.inWaiting() < 2):    #Data always arrives in a minimum pair of two, so wait
307       return (0,0)
308     return (self.ser.read(1), self.ser.readline())
309 
310   def bufferok(self):
311     if (self.ser.inWaiting() < 1):
312       return True
313     else:
314       return False
```

```
315
316   def getConstants(self):
317     self.getConstantsRunning = True # Lock the buffer so the other processes cannot read it
318     self.ser.write(self.getVars) #Write notification to Arduino
319     #Wait for line to be sent and read
320     response = self.ser.readline()
321     self.getConstantsRunning = False  # Once we have our information unlock the buffer
322     #print '\n ', response
323     values = response[3:]
324     Kp,Ki,Kd,Setpoint,Accuracy = values.split(",")
325     return (Kp),(Ki),(Kd),(Setpoint),(Accuracy)
326
327   def sendConstants(self,Kp,Ki,Kd,SetPoint,Accuracy):
328     #print Kp, ' ',Ki, ' ',Kd, ' ',SetPoint, ' '
329     self.ser.write('V')
330     self.ser.write(struct.pack('@f', Kp))
331     self.ser.write(struct.pack('@f', Ki))
332     self.ser.write(struct.pack('@f', Kd))
333     self.ser.write(struct.pack('@f', SetPoint))
334     self.ser.write(struct.pack('@f', Accuracy))
335     self.ser.write('V')
336 class UI:
337 # ------------------------------
338 # Date: 30/12/14, Last Modified - 30/12/14
339 # This class holds functions for interacting with the user
340 # ------------------------------
341   def __init__(self,GUIin): #Initilise class
342     self.clear_msg = '                                                  '  #A set of spaces used
          to overwrite messages in terminal
343     self.bottomLine = 'Please choose an option (press m and return for menu):'
344     self.GUI = GUIin
345
346   # User Input Function
347   def captureInput(self):
```

```
348   #Function to capture key presses from the user, this should be started in a new thread
349     global user_last_response #declare variable as global to allow editing
350     user_last_response_lock.acquire() #aquire lock
351     user_last_response = raw_input("")  #wait for user input and read it in
352     user_last_response_lock.release() #release lock, allowing main to see its contents
353
354   def print2(self,message):
355   # This function will output data one line above some text output to the user if the GUI is not
          active
356     if(self.GUI):
357       print message
358     else:
359       print '\r', self.clear_msg, #overwrite last 'Please choose...' with spaces
360       print '\r', message    #print the message
361       print self.bottomLine,  #re-output the bottom line
362       sys.stdout.flush()  #flush so the user can actually see it
363
364   def respond(self,response,data_type): #Function to handel what happens when the user makes a
          request
365     global arduino
366     global temprature
367     if(response == 'm'):    #Test for menu option
368       self.display_menu()
369     elif(response == 't'):
370       ok = data_type.startGraph('Seconds since the epoch','Temprature (Celsius)','Temprature of
            Component')
371       if ok == 1:
372         data_type.killGraph()
373     elif(response == 's'):    #Test for menu option
374       statOut = data_type.stats()
375       if (not(statOut)):
376         print '\nSorry no data gathered yet'
377       else:
378         print '\nLast Recorded Time: ', time.ctime(statOut[0])
```

```
379         print 'Last Recorded Temprature: ', statOut[1]
380         print 'Mean Temprature: ', statOut[2]
381         print 'Standard Deviation: ', statOut[3]
382         print 'Peak to peak: ',statOut[4]
383       elif(response == 'v'):
384         OutVal = arduino.getConstants()
385         print '\nKp = ', OutVal[0]
386         print 'Ki =', OutVal[1]
387         print 'Kd =', OutVal[2]
388         print 'Setpoint = ', OutVal[3]
389       elif(response == 'f'):
390         print '\n'
391         fileName = raw_input('Please enter a short filename: ')
392         desc = raw_input('Please enter a longer description of this measurement: ')
393         temprature.startFout(fileName,desc)
394       elif(response == 'F'):
395         temprature.killFout
396       elif(response == 'e'):
397         if(temprature.fout):
398           temprature.killFout
399         raise SystemExit
400       else:       #if its not valid inform the user
401         print "Oops, I sorry I don't recognise that option"
402       print self.bottomLine,
403       sys.stdout.flush()
404
405   def display_menu(self):
406     print '\n',
407     while(True):
408       print '======== Menu ========'
409       print ' - m -> display this menu'
410       print ' - t -> Toggle Temprature graph'
411       print ' - s -> Output Statistics'
412       print ' - v -> Get Variables from Arduino'
```

```
413       print ' - f -> Start outputing data to file'
414       print ' - F -> Stop outputting data to file'
415       print ' - e -> Exit program'
416       print '======================'
417       print 'NB Alway press return after an input'
418       print 'the progam can only accept one character'
419       print 'at a time!'
420       print 'Please press q to quit'
421       response = raw_input("or a character to find out more...")
422       if (response == 'q'):
423         return
424       elif (response == 'm'):
425         print 'The m function displays this menu'
426         print 'and lets you viewed detailed help'
427         print 'on each function'
428       elif (response == 't'):
429         print 'Toggle the temprature graph between on and off'
430       elif (response == 's'):
431         print 'Output the following statistics'
432         print 'Last Recorded Time and Temprature, Mean Temprature and standard deviation'
433       elif (response == 'v'):
434         print 'Load the proportional, intergral and differntial constanants'
435         print 'and the setpoint from the Arduino and display them'
436       elif (response == 'f' or response == 'F'):
437         print 'Start or stop outputting data to file'
438       elif (response == 'e'):
439         print 'Exit the program fully'
440       else:
441         print "Sorry I don't recongnise that option"
442       raw_input("Please press return")
443
444 # Data Class
445 class Data:
446 # ----------------------------
```

```
447 # Date: 23/12/14, Last Modified - 30/12/14
448 # This class holds data and handles the plotting of graphs
449 # ------------------------------
450   def __init__(self,maxGraphPoints,maxStorePoints): #Initilise function
451     self.xdata = [] #Create blank array for x and y data
452     self.ydata = []
453     self.graph = False  #No graph present yet
454     self.fout = False #Do not output data to file
455     self.fHandle = False
456     if (maxGraphPoints > maxStorePoints): #The amount of data in the store must be greater than the
          number of store points
457       raise StandardError('Number of Store points must be >= number of graph points')
458     self.gPoints = maxGraphPoints # Load maximum number of points for graph and store
459     self.sPoints = maxStorePoints
460
461   def addData(self,x,y):
462     self.xdata.append(float(x)) #Append new data
463     self.ydata.append(float(y))
464     #print '\n', self.ydata
465     while (len(self.xdata)>self.sPoints): #Delete old data to make room in memory for new data
466       self.xdata.pop(0)
467       self.ydata.pop(0)
468     if(self.graph):
469       self.l.set_data(self.xdata[-self.gPoints:],self.ydata[-self.gPoints:])
470       # update graph data. self.l.set_data = A pointer function that allows setting of data
471       # The function accepts two arguments, xdata and ydata
472       # Select the data from the store, and select the last points using slicing
473       # It should fail for npoints<gPoints, but python is forgiving
474       self.axes.relim()   # Recalculate limits
475       self.axes.autoscale_view(True,True,True) #Autoscale
476     if(self.fout):
477       self.data2F(x,y)
478
479   def reDraw(self):
```

```
480     plt.draw()     # Redraw
481
482   def startGraph(self,xlabel,ylabel,title):
483     if (self.graph):
484       return 1 #test to see if the graph is already open
485
486     plt.ion()    # Enable interactive mode, if not already
487     self.fig = plt.figure() # Create figure, store pointer in class
488     self.axes = self.fig.add_subplot(111) # Add subplot (dont worry only one plot appears)
489     self.axes.set_autoscale_on(True)  # enable autoscale
490     self.axes.autoscale_view(True,True,True)
491     self.l, = plt.plot([], [], 'r-')  # Plot blank data
492     plt.xlabel(xlabel)       # Label Axis
493     plt.ylabel(ylabel)
494     plt.title(title)       # Add title
495     plt.grid()         # Add grid
496     self.graph = True
497     return 0
498
499   def killGraph(self):
500     plt.clf() #Clear all objects related to the figure, leave figure open
501     plt.close() # Close figure
502     self.graph = False  # Inform other functions that the figure is closed
503
504   def stats(self):
505     if(not(self.xdata)):
506       return
507     lastX = self.xdata[-1]
508     lastY = self.ydata[-1]
509     meanY = np.mean(self.ydata)
510     stdY = np.std(self.ydata)
511     peak = (np.amax(self.ydata)) - (np.amin(self.ydata))
512
513     return (lastX,lastY,meanY,stdY,peak)
```

```
514
515   def startFout(self,fileName,note): #Start outputing data to a file
516     if (self.fout):
517       global user
518       user.print2('It looks like data is already being output\n')
519       return
520
521     startTime = time.gmtime() #Determine current time
522
523     fName = 'Data/' + str(startTime[0]) + str(startTime[1]) + str(startTime[2]) + '_'
524     # Generate Filename 'YYYYMMDD_HHSS-fileName
525     fName = fName + str(startTime[3]) + str(startTime[4]) + '-' + fileName
526
527     bufferData = open(fName+'-buff.dat','w')
528     buffSize = len(self.xdata)
529     for i in range(0,buffSize):
530       bufferData.write(str(self.xdata[i]) + ',' + str(self.ydata[i]) + '\n')
531     bufferData.close()
532
533     info = open(fName+'.info','w')
534     info.write(lotsChar('=',30)+'\n')
535     info.write('\t UOB Temprature Controller Output File\n')
536     info.write(lotsChar('=',30)+'\n\n')
537
538     info.write('Date Reading Taken: ')
539     info.write(time.strftime('%c',startTime)+'\n')
540     info.write('Notes: ' + note + '\n')
541
542     info.write(fName+'-buff.dat contains all the data on the buffer at the time that this
          measurement started\n')
543     info.write(fName+'.dat contains data collected since the time of measurement\n')
544     info.write(fName+'.info (This file) is an automatically generated infomation file\n')
545     info.write('Column 1 of the data files is the date and time the column 2 data was recieved,
          expressed in seconds since the UNIX epoch (1st Jan 1970)\n')
```

```
546     info.write('Column 2 is the data, this is temprature data expressed in degrees celsius\n')
547     info.write('The data is comma seperated with a new line character expressing a new
            measurement\n')
548     info.close()
549
550     self.fHandle = open(fName+'.dat','w')
551     self.fout = True
552
553   def data2F(self,xdata,ydata):
554     self.fHandle.write(str(xdata) + ',' + str(ydata) + '\n')
555
556   def killFout(self):
557     self.fHandle.close()
558     self.fHandle = False
559     self.fout = False
560
561 # ====================
562 # Call Main Function
563 # ====================
564 GLOBAL_GUI = prelims() # Call a preliminary function to start loading things
565
566 # Create Classes Globally, just about every function needs them
567 user = UI(GLOBAL_GUI) # Open class to hold UI functions
568 arduino = USB(device_link)
569 temprature = Data(CONSTANT_GRAPH_LENGHT,CONSTANT_BUFFER_LENGHT)
570
571 main(GLOBAL_GUI) # Call Main to enter main loop
```

H.1.3 Readme.txt

The README.

```
1 #============================================
2 #~~~~~  Arduino Temprature Controller ~~~~~~~
3 #--------------------------------------------
```

```
A simple(ish) program for very high stability
temprature control for scientific purposes,
using only hobbyist components

This folder should contain the following files and directories

Key Files
-------------
  - User_Interface.py
  - Arduino_TempController
    - Arduino_TempController.ino
  - README.txt
  - Project Report

The file User_Interface.py is a python 2.7 program
that can run on a host computer and communicate
with the Arduino, graphing tempratures and changing
set points on the fly. NOTE any changes to set points
will be lost when/if the arduino is reset.

Arduino_TempController.ino is the Arduino sketch (program)
that needs to be compiled and loaded onto the Arduino.
This program is standalone and once the Arduino is running
no further input from the PC is required. This is the file
to permanentley edit the setpoints for the temprature controller

README.txt is this file

Other Required Programs
-----------------------
  - GUI.py
  - Data

GUI.py is a python module required for displaying the
```

```
38 Graphical User Interface for the User_Interface program.
39 This requires PyQt which is released under the full GPLv3!
40
41 Data is the Data folder.
42
43 Utilities
44 ---------
45   - EpochToTime.py
46   - plotData.py
47   - ArduinoSim.py
48   - R2T.py
49   - T2R.py
50   - cpustatus.sh
51
52 EpochToTime is a simple python utility to convert the
53 time used for data (seconds since the UNIX Epoch (1st Jan 1970)
54 into a human readable time
55
56 plotData.py This is a python utility to plot data output
57 by the User_Interface program. It accepts one argument on
58 the command line ie python plotData.py Data/FileName.dat
59
60 ArduinoSim.py This is a little python utility for
61 emulating the Arduino. It just generates random numbers
62 near 21 and outputs them over serial. You need to
63 setup a serial loop using socat, this program is LINUX ONLY.
64 This program is based hevily on a Stack Exchange solution
65
66 R2T.py and T2R.py These progams generate a resistance
67 from a temprature or vica a versa for a thorlabs 10k
68
69 cpustatus.sh This is a shell script to moniter the
70 core temprature of a raspberry pi. This is not
71 my work and is based of a stack exchange solution
```

H.1.4 GUI.py

The Graphical User Interface.

```
#============================================
#~~~~~  Arduino Temprature Controller ~~~~~~~
#--------------------------------------------
# Interface Code
# V1.0
# Author -  Aaron Jones
# Date: 25/03/2015
# Copyright Aaron Jones, 2015
#--------------------------------------------
# This file provides a GUI for the python
# file User_Interface.py
# Please see User_Interface for further details
#
# NB This code requires PyQt4 which is released
# under the full GPLv3.
# ============================================

import sys
from PyQt4 import QtGui, QtCore
import time

# This function will spawn a new thread with Qt
class Thread(QtCore.QThread):
    def __init__(self,Function):
        QtCore.QThread.__init__(self)
  self.function = Function

    def run(self):
        self.function(True) #Specify function
        self.exec_()  #run

class MainWindow(QtGui.QWidget):
```

```
33
34      def __init__(self,Ard,Temp,Ui):
35          super(MainWindow, self).__init__()
36
37          self.grid = QtGui.QGridLayout()
38          self.grid.setSpacing(10)
39          self.initUI()
40    self.timer = QtCore.QTimer()
41    self.timer.timeout.connect(self.events)
42    self.MaxtimerVal = 100
43    self.timer.start(self.MaxtimerVal)
44    self.ard = Ard
45    self.temp = Temp
46    self.ui = Ui
47
48      def initUI(self):
49
50    # Set up Window
51    self.setGeometry(100, 100, 500, 100)
52    self.setWindowTitle('Arduino Temprature Controller - Options')
53
54    msg = "Press Read Constants to find out :)"
55
56    # Setup Buttons
57    lastTempText = QtGui.QLabel('Last Recorded Temprature')
58    lastTimeText = QtGui.QLabel('Time of last Recorded Temprature')
59    varTempText = QtGui.QLabel('Variance')
60    avTempText  = QtGui.QLabel('Average')
61    peakText = QtGui.QLabel('Peak to Peak Value')
62    tarTempText = QtGui.QLabel('Target Temprature')
63    allowedDelta = QtGui.QLabel('Desired Stability')
64
65    self.lastTemp = QtGui.QLabel('0')
66    self.lastTime = QtGui.QLabel('0')
```

```
67    self.varTemp = QtGui.QLabel('0')
68    self.avTemp  = QtGui.QLabel('0')
69    self.peakTemp  = QtGui.QLabel('0')
70    self.tarTempEdit = QtGui.QLineEdit(msg)
71    self.allowedDeltaEdit = QtGui.QLineEdit(msg)
72
73    self.grid.addWidget(lastTempText, 1, 0)
74    self.grid.addWidget(self.lastTemp, 1, 1)
75
76    self.grid.addWidget(lastTimeText, 2, 0)
77    self.grid.addWidget(self.lastTime, 2, 1)
78
79    self.grid.addWidget(varTempText, 3, 0)
80    self.grid.addWidget(self.varTemp, 3, 1)
81
82    self.grid.addWidget(avTempText, 4, 0)
83    self.grid.addWidget(self.avTemp, 4, 1)
84
85    self.grid.addWidget(peakText, 5, 0)
86    self.grid.addWidget(self.peakTemp, 5, 1)
87
88    self.grid.addWidget(tarTempText, 6, 0)
89    self.grid.addWidget(self.tarTempEdit, 6, 1)
90
91    self.grid.addWidget(allowedDelta, 7, 0)
92    self.grid.addWidget(self.allowedDeltaEdit, 7, 1)
93
94    i = 10
95
96    kpText = QtGui.QLabel('Proportional Constant')
97    kiText = QtGui.QLabel('Intergral Constant')
98    kdText = QtGui.QLabel('Derivitive Constant')
99
100   self.kpEdit = QtGui.QLineEdit(msg)
```

```
101     self.kiEdit = QtGui.QLineEdit(msg)
102     self.kdEdit = QtGui.QLineEdit(msg)
103
104     self.grid.addWidget(kpText, i, 0)
105     self.grid.addWidget(self.kpEdit, i, 1)
106
107     self.grid.addWidget(kiText, i+1, 0)
108     self.grid.addWidget(self.kiEdit, i+1, 1)
109
110     self.grid.addWidget(kdText, i+2, 0)
111     self.grid.addWidget(self.kdEdit, i+2, 1)
112
113     self.shGraph = QtGui.QPushButton('Show/Hide Graph', self)
114     self.shGraph.clicked.connect(self.ShowGraph)
115     self.grid.addWidget(self.shGraph,i+3,0)
116
117     self.Fout = QtGui.QPushButton('Start Outputting to File', self)
118     self.Fout.clicked.connect(self.fileOutput)
119     self.grid.addWidget(self.Fout,i+3,1)
120
121     self.sendConst = QtGui.QPushButton('Send New Constants', self)
122     self.sendConst.clicked.connect(self.transConst)
123     self.grid.addWidget(self.sendConst,i+4,1)
124
125     self.readConst = QtGui.QPushButton('Read Constants From Device', self)
126     self.readConst.clicked.connect(self.LoadVars)
127     self.grid.addWidget(self.readConst,i+4,0)
128
129     self.setLayout(self.grid)
130     self.show()
131
132   def ShowGraph(self):
133         self.ui.respond('t',self.temp)
134         self.show()
```

```
135
136     def fileOutput(self):
137   if(self.temp.fout):
138     self.temp.killFout()
139     self.Fout.setText('Start Outputting to File')
140   else:
141     fName, ok1 = QtGui.QInputDialog.getText(self, 'Open File - File Name', 'Please enter a short
          filename:')
142     desc, ok2 = QtGui.QInputDialog.getText(self, 'Open File - Description', 'Please enter a short
          description of the measurements:')
143     if(ok1 and ok2):
144       self.temp.startFout(fName,desc)
145       self.Fout.setText('Stop Outputting to File')
146
147     def events(self):
148   bufferGood = self.ard.bufferok()
149   if (bufferGood):
150     if(self.temp.graph):
151       self.temp.reDraw()
152
153     statOut = self.temp.stats()
154     if (statOut):
155       self.lastTemp.setText(str(statOut[1]))
156       self.lastTime.setText(time.ctime(statOut[0]))
157       self.varTemp.setText(str(statOut[3]))
158       self.avTemp.setText(str(statOut[2]))
159       self.peakTemp.setText(str(statOut[4]))
160   self.timer.start(self.MaxtimerVal)
161
162     def LoadVars(self):
163       value = self.ard.getConstants()
164       self.kpEdit.setText(value[0])
165       self.kiEdit.setText(value[1])
166       self.kdEdit.setText(value[2])
```

```
167         self.tarTempEdit.setText(value[3])
168     self.allowedDeltaEdit.setText(value[4])
169 
170       def transConst(self):
171         Kp = self.kpEdit.text()
172         Ki = self.kiEdit.text()
173         Kd = self.kdEdit.text()
174         setpoint = self.tarTempEdit.text()
175     accuracy = self.allowedDeltaEdit.text()
176     try:
177       Kp = float(Kp)
178       Ki = float(Ki)
179       Kd = float(Kd)
180       setpoint = float(setpoint)
181             accuracy = float(accuracy)
182     except Exception:
183         QtGui.QMessageBox.about(self, 'Error','Input can only be a number')
184         return
185     self.ard.sendConstants(Kp,Ki,Kd,setpoint,accuracy)
```

H.2 Widgets

The following programs were useful to convert data between different formats.

H.2.1 EpochToTime.py

A small widget to convert a time since unix epoch into a human readable date and time.

```
1 from time import ctime
2 
3 t = float(raw_input("Please enter seconds since the Epoch: "))
4 
5 print "Time is: ", ctime(t)
```

H.2.2 plotData.py

A widget to plot data output by the program.

```
## File to produce plot from data files

#============================================
#~~~~~  Arduino Temprature Controller ~~~~~~~
#--------------------------------------------
# Interface Code
version = 1
# Author -  Aaron Jones
# Date: 12 Jan 2015
# Copyright Aaron Jones, 2015
#--------------------------------------------
# What does this file do?
# This file reads data generated by the
# controller and prints a graph
# How to use
#   Pass the main file as the first argument
# when running the script. The file will
# also find and read the buffer file automatically
# Example
# python plotData.py "file1.dat"
# This will load file1-buff.dat followed by
# file1.dat and produce a graph
# ============================================

import numpy as np
import sys
import pylab
import matplotlib.pyplot as plt

mainFile = sys.argv[1]  # Use value from the command as the main file
index = mainFile.find('.dat') #Identify the insertion point
buffFile = mainFile[:index] + '-buff' + mainFile[index:] #and generate the name of the buffer file
```

```
33
34 xdata, ydata = np.loadtxt(buffFile, dtype='float', comments='#', delimiter=',', converters=None,
       skiprows=0, unpack=True, ndmin=0)
35 # Load the buffer file
36 xdata1, ydata1 = np.loadtxt(mainFile, dtype='float', comments='#', delimiter=',', converters=None,
       skiprows=0, unpack=True, ndmin=0)
37 #Load the main file
38
39 time = np.append(xdata,xdata1)
40 temp = np.append(ydata,ydata1)
41 #Append the main data onto the buffer data
42
43 meanT = np.mean(temp)
44 stdT = np.std(temp)
45
46 print "Mean Temprature: ", meanT
47 print "Standard Deviation: ", stdT
48
49 plt.ion()    # Enable interactive mode, if not already
50 fig = plt.figure()  # Create figure, store pointer
51 axes = fig.add_subplot(111) # Add subplot (dont worry only one plot appears)
52 l, = plt.plot(time, temp,'x-')  # Plot data
53 axes.set_autoscale_on(True) # enable autoscale
54 axes.autoscale_view(True,True,True)
55 axes.relim()         #Calculate limits
56 plt.xlabel('Time')       # Label Axis
57 plt.ylabel('Temprature')
58 plt.title('Temprature data for target temprature equal to ambiant temprature')  # Add title
59 plt.grid()         # Add grid
60 plt.show(block = True)       # Enable block to graph stays open
```

H.2.3 ArduinoSim.py

A widget to simulate the Arduino on Linux (sorry windows users).

```
# ============================
#    Arduino Simulator for Linux
#============================
# Copyright Aaron Jones, 2015
# The following code sends data over a serial emulator as described in the following answer
# http://stackoverflow.com/questions/52187/virtual-serial-port-for-linux
#Complementing the @slonik's answer.
#You can test socat to create Virtual Serial Port doing the following procedure (tested on Ubuntu 12.04):
#Open a terminal (let's call it Terminal 0) and execute it:
# socat -d -d pty,raw,echo=0 pty,raw,echo=0
#The code above returns:
# 2013/11/01 13:47:27 socat[2506] N PTY is /dev/pts/2
# 2013/11/01 13:47:27 socat[2506] N PTY is /dev/pts/3
# 2013/11/01 13:47:27 socat[2506] N starting data transfer loop with FDs [3,3] and [5,5]
#Open another terminal and write (Terminal 1):
# cat < /dev/pts/2
#Open another terminal and write (Terminal 2):
# echo "Test" > /dev/pts/3
#Now back to Terminal 1 and you'll see the string "Test".

import serial
import struct
import time
import numpy as np
ser = serial.Serial('/dev/pts/10', 9600)
ser.write('h\n')
while True:
  out = 21 + np.random.random()
  print out
  ser.write('d')
  ser.write(str(out))
  ser.write('\n')
  time.sleep(1)
```

```
34 #Comment
```

H.2.4 R2T.py

A widget to convert a resistance of a ThorLabs 10K to a temperature

```
1  # Suitable for 3,599 < R < 32,770
2
3  from math import log, pow
4
5  a = 3.3540170e-03
6  b = 2.5617244e-04
7  c = 2.1400943e-06
8  d = -7.2405217e-08
9  Rt = 10000
10
11 R = float(raw_input("Please enter resistance"))
12 const = a
13 linear = b*(log(R/Rt))
14 quad = c*(log(pow((R/Rt),2)))
15 cubic = d*(log(pow((R/Rt),3)))
16
17 T = 1/(const + linear + quad + cubic)
18
19 #print const
20 #print linear
21 #print quad
22 #print cubic
23
24 print "The temprature is ", T, "k"
25 T = T - 273.15
26 print "The temprature is ", T, "C"
```

H.2.5 T2R.py

A widget to convert a temperature to the resistance of a ThorLabs 10K

```
# Suitable for 3,599 < R < 32,770

from math import exp, pow

a = -1.5470381e1
b = 5.6022839e3
c = -3.7886070e5
d = 2.4971623e7
Rt = 10000

T = float(raw_input("Please enter a temprature in C between 0 and 50: "))
T = T + 273.15
const = a
linear = b/T
quad = c/(pow(T,2))
cubic = d/(pow(T,3))

R = Rt*exp(const + linear + quad + cubic)

#print const
#print linear
#print quad
#print cubic

print "The resistance is ", R, " Omhs"
```