University of Southern Queensland

Faculty of Engineering & Surveying

Embedded Control System for Biogas Digesters

A dissertation submitted by

T. Sullavan

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering, Mechatronics

Submitted: October, 2007

Abstract

Biogas is also known as dump gas, marsh gas or sewer gas and is produced in a common, naturally occurring biological decomposition process. It is composed mainly of methane, CH_4, carbon dioxide, CO_2, and hydrogen sulphide, H_2S. Smaller quantities of hydrogen, H oxygen O_2, nitrogen, N and ammonia, NH_3 may also be present.

Methanogens occur naturally in the digestive systems of ruminant animals such as cattle, in marshes, brackish waters and sewage works, where, in addition to laboratories, much knowledge of them has been gained.

Literature review reveals that biogas production and thus rate of decomposition is influenced by temperature, and additionally that the loading conditions of a digester may be deduced by monitoring the concentrations of individual component gasses of the biogas produced.

It is the objective of this project to:

- Develop an electronic nose to monitor the biogas composition,
- Develop a fuzzy logic controller to increase or decrease the temperature of the digester vessel, thus controlling the rate of decomposition, and
- Notify users of the digester state using semaphore style indicator lights.

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

T. Sullavan

0050011624

Signature

Date

Acknowledgments

Special thanks go to the supervisors of this project, Dr Selvan Pather and Mr Mark Phythian for their invaluable guidance, and to my family and fiance for their unending patience and tolerance throughout the duration of this project and my studies. I could have achieved little without them.

T. Sullavan

University of Southern Queensland

October 2007

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

1.1.1 History

The methanogenic bacteria appear to be ancient forms of life. Hydrogen and oxygen outgassed from the Earth's mantle could well have supported the existence of these organisms long before the development of primitive plant species hundreds of millions of years ago. Anaerobic digestion of biomass has occurred throughout the existence of life on Earth, and is therefore expected to be a highly reliable, stable and efficient process in those environments where it occurrence is essential to maintain the natural cycle of organic matter (Smith, Bordeux, Goto, Shiralipour, Wilkie, Andrews, Ide & Barnett 1988).

The identification of a specific gas from the anaerobic digestive process was first made by Volta in 1776. 'Inflammable air' was found to be readily generated (Stafford et al. 1981). It has been used for at least a century in developing countries for the decomposition of animal and human waste, in most cases from a necessity to maximise the benefit from limited available resources in rural areas. While the combustion of cow dung for cooking provided a renewable source combustible fuel, it was at the expense of fertiliser for local crops, and could hardly be pleasant to use in the confines of a small rural kitchen. Exposed human waste encourages vermin and diseases. In response, anaerobic digesters were developed to decompose the waste to provide a rich organic fertiliser for crops and a fuel gas to power household lighting and stoves.

1.1.2 Ethics

As it stands at the moment, the time has come to address serious shortcomings in our current socio-economic direction. Numerous mineral fuel supply interruptions, price spikes, bloody international conflicts, and more recently, revelations concerning global warming and environmental degradation have stimulated interest in current energy sources. The human race is facing increasing problems associated with disposing of waste produced directly by the population, and by the agricultural and industrialised systems we use to feed and otherwise occupy ourselves. Soils used by mass commercial crop cultivation methods are also increasingly showing signs of degradation due to intensive farming paractices reliant on the extended use of chemical fertilisers and pesticides, often also based on mineral oil derivatives.

Reflection on the benefits and surrounding issues has led to belief that the utilisation of anaerobic digestion to produce biogas could play an important and profitable role in addressing a wide variety of social and environmental issues faced by human beings as we enter the 21st century. The production of human and animal waste is independent of crude oil supplies and available to a much wider demography. Biogas combustion produces only a fraction of the pollution of fuels and energy systems widely used today, and competition for the raw materials required for biogas production are unlikely to cause conflict. The effluent of anaerobic digesters can be used to organically nourish and restore fertility to agricultural soils and hydroponics, without the health concerns associated with chemicals. While techniques to achieve anaerobic decomposition have been ongoing areas for research across the world for some time, many installations could benefit from modern technology and safety features, resulting in improved biogas yields, improvements in plant safety, and reduced and better targeted operational requirements.

1.2 Objectives

The objective of this project is to improve the safety and performance of small digester systems by:

1. Developing and constructing an array of gas sensors otherwise known as an e-nose to monitor biogas composition,
2. Developing an embedded controller to monitor and control a digester vessel,
3. Adding user interface to notify personnel of the state of the digester.

1.3 Methodology

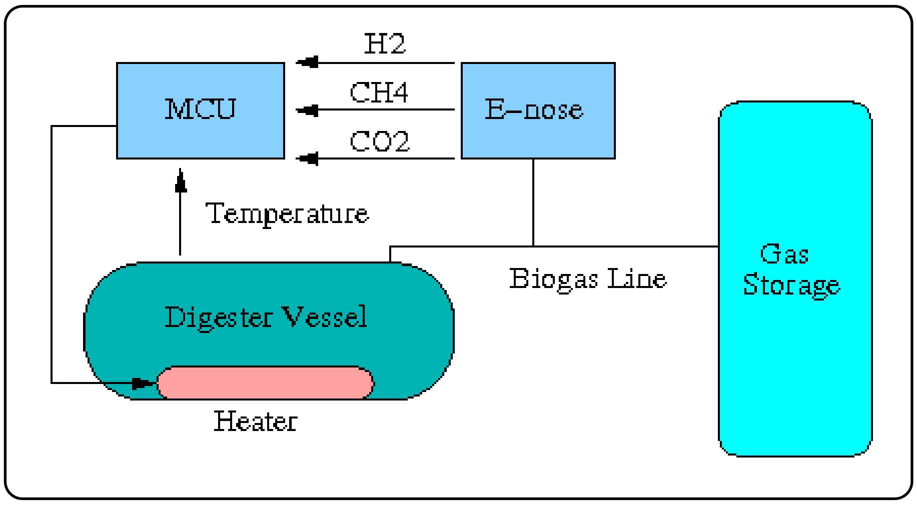

Figure 1.1: Biogas Control System Proposal

1. The monitoring of hydrogen H2, carbon dioxide CO2, and methane CH4 in the gas economically and effectively in order to provide sufficient data to allow reliable data concerning the health of the digester.
2. The development of an embedded micro-controller hardware and software using fuzzy logic to control the system by autonomously maintaining appropriate temperature.

3. The microcontroller is to notify personnel of the requirements of the system through a system of semaphore lights. This information should indicate digester health/ overload conditions as appropriate.

A simple digram of this proposal is included in Figure 1.1.

To achieve this, the following course of action should lead to the achievement of the objectives: The system hardware must first be developed to permit monitoring of the target gasses. The sensors may then be connected to the micro-controller. This will permit testing of the gas sensors in the presence of pure samples of the target gas samples and derivation of algorithms to process the readings and estimate gas composition. Software may then be developed interpret the readings and to and set a suitable temperature in response.

Chapter 2

Biogas

2.1 Chapter Overview

This chapter describes details concerning the composition and production of biogas. It then goes into aspects of control of a simple biogas digester system.

2.2 Properties of Biogas

Biogas is also known as dump gas, marsh gas or sewer gas and produced in a common, naturally occurring decomposition process. It is composed mainly of methane CH_4, carbon dioxide CO_2, and hydrogen sulfide H_2S. Smaller quantities of hydrogen H_2 oxygen O_2, nitrogen N, and ammonia NH_3 may also be present. The component most responsible for the flammability of biogas is the methane, which usually makes up to 70% of the composition. Carbon dioxide makes up the majority of the remainder with smaller quantities of hydrogen sulfide, which gives raw biogas its distinctive rotten egg smell. This can easily be removed by simple scrubbing techniques. Otherwise the biogas resembles natural gas in composition, with a slightly lower energy content. AS 4564-2005 table 3.1 imposes limits on the composition of general purpose natural gas, necessitating additional scrubbing of biogas in most cases for compliance if a biogas system is to supply commercially.

2.3 Biogas Production

Biogas is a product of anaerobic decomposition by methanogens, a group of bacteria characterised by their ability to produce methane. These bacteria flourish only in the absence of oxygen; 0.01mg/L is enough to completely inhibit growth (Stafford et al. 1981). They occur naturally in the digestive systems of ruminant animals such as cattle, in marshes, brackish waters and sewage works, where in addition to laboratories, much knowledge of them has been gained.

Methanogenic decomposition is a complex series of bio-reactions far beyond the scope of this dissertation, but the basic mechanism relies on the co-ordinated use of two different heterogenic groups of bacteria. The first are responsible for the decomposition of lipids, lignins, proteins ands into H_2, CH_2 and volatile fatty acids (VFA's). The second type are the methanogens, which further decompose the products of the first reactions to form CO_2 and CH_4.

Digestion is the creation of an artificial environment to encourage Methanogenic decomposition, about which considerable research has been published.

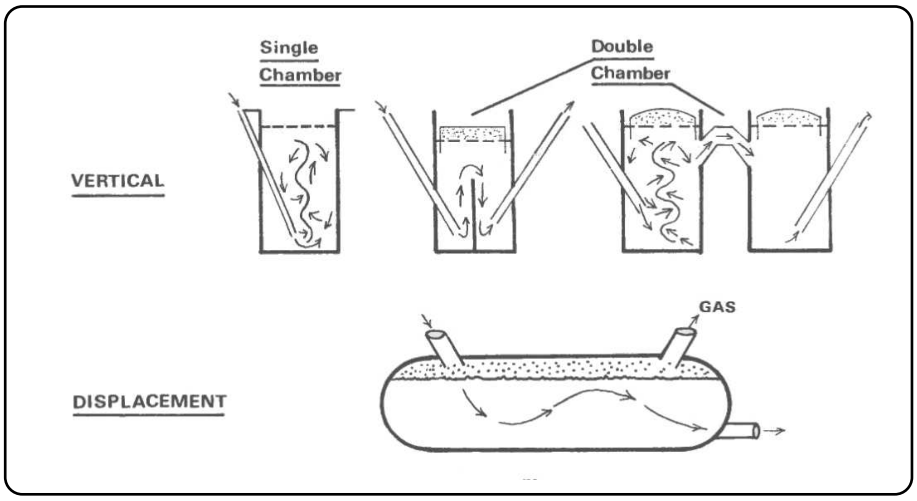

Figure 2.1: Types of continuous flow digesters. (Fry 1973).

Most digesters in operation are regularly fed with human, animal, or organic industrial waste and produce biogas and an organic sludge, low in pathogens and high in protein and nitrogen, which make it suitable for fertiliser, or with de-watering, animal feed. Small scale digesters generally make use of a single reaction vessel, and may be batch fed and sealed until gas production stops, or continually fed, using an internal system of pipes and gravity to maintain a constant operating level as shown in Fig. 2.1. Other types of varying degrees of sophistication are used in larger installations. It is commonly suggested that feed with a carbon to nitrogen ratio, C:N from 20:1 to 30:1 is desirable, and an operating pH of 7 or just over is optimum for the methanogenic bacteria (Hobson, Bousfield & Summers 1981).

There are two temperature ranges over which anaerobic digestion is most effective. The mesophillic, usually from ambient to 42 deg C, and thermophillic from 55 to 70 deg C. In practice, few digesters operate in the thermophillic zone because of the extra energy required for heating and less stable micro-biological environment. A temperature around 35 deg C is common to maximise biogas yield whilst minimising digester retention time and heating energy demands. The microbes are reportedly sensitive to changes in operating conditions, so once they have acclimatised to the conditions of a particular digester, sudden fluctuations in temperature or feed material are not recommended due to the destabilisation of the reactions.

The digestate material must also be agitated to a degree to promote a homogeneous mixture, prevent micro-biologically dead zones and reduced the build up of scum, which is a substrate consisting of lighter fractions and fibrous material such as hair and feathers. The latter can cause significant problems when it hardens on the surface and prevents the escape of gas from the slurry. Careful design of vessel parameters such as operating level, gas outlet position, maintenance and slurry agitation can greatly reduce or eliminate problems with scum.

An additional problem with anaerobic digesters is instability caused by toxic, organic or hydraulic overload (Stafford et al. 1981). Toxic overload can be caused by contamination of the feed by substances which inhibit the digestion process. Organic overload is commonly caused by variations in feed consistency or too short a detention time. Hydraulic overload occurs when the feed material is low in solids and retention time is reduced to compensate, flushing out the microbial population. For each of these conditions, there are indicators which may be monitored to allow anticipation of such failures and corrective action to avoid disruption of the digestion process.

For the purposes of this research project, overload occurs when the methanogens are unable to keep pace with the first group of bacteria responsible for breaking down the more complex compounds. These bacteria tend to be more robust and multiply more rapidly than the methanogens. When this occurs, the methanogens are unable to consume the VFA's as they are produced. This has the effect of lowering the pH of the substrate below that which the methanogens can survive, which is only between pH 6.4 to 7.5. This results in the poisoning of the methanogens and digester failure.

Stafford et al (Stafford et al. 1981, p77-78) compiled an inexhaustive list of these conditions and indicators, and suggest possible remedies as included below. These correspond to the diagram in Figure 2.2.

- Common Faults
  - A) Toxic Overload
  - B) Organic Overload
  - C) Hydraulic Overload
- Typical warning sign
  - i) methane production falls
  - ii) The direction of response curves of several variables when compared with one another can warn of potential failure; for example, VFA rises as the CH4 production falls
  - iii) The sign of the second derivative of common variables with respect to time changes; for example:
    - A concave upward change in the slope of percentage CO2
    - A concave upward change in the slope of pH
    - A concave downward change in slope of rate of CH4 production
- remedies which may be applied

1) Solids recycle
2) Increase frequency of loading
3) Adequate mixing
4) Scrubbing gas to remove CO2 before recycle
5) Addition of a base
6) Recycle micro-organisms
7) Reduce or stop feed
8) Increase initial substrate conditions
9) Add plant effluent to feed

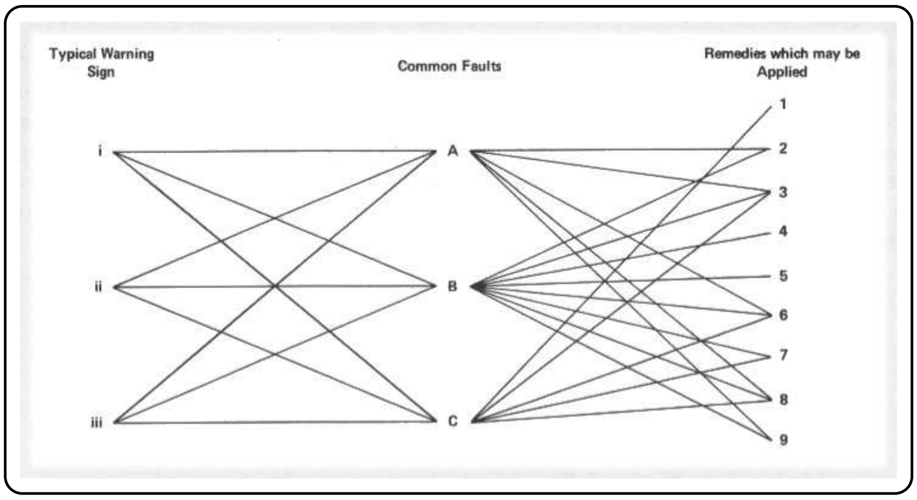

Figure 2.2: Digester indicators, conditions, and remedies. (Stafford et al. 1981).

2.4 Digester Control

More recent research has resulted in something of a trend towards the use of fuzzy logic control systems to monitor and control digester performance. Such systems have successfully been implemented in scale laboratory models of wastewater treatment systems using fluid bed reactor type digesters by Murnleitner et al and Muller et al.

Stafford (Stafford et al. 1981, p78) suggests the monitoring of methane production, VFA levels, CH_4 production, as well as CO_2 and pH levels and their second derivatives with respect to time as a method of monitoring a model fluid bed reactor.

Murnleitner et al agree (Murnleitner et al. 2002), noting the use of several easily monitored parameters using off the shelf type sensors. The monitoring methane and hydrogen levels, pH gas flow and oxidation-reduction potential and conductivity are included. The results of their research show that hydrogen and methane levels in the gas accurately reflect the health of the digester. If gas composition data is unavailable, pH levels are used. They conclude that methane and hydrogen levels, pH and gas flow are all that are needed to recognise overload conditions.

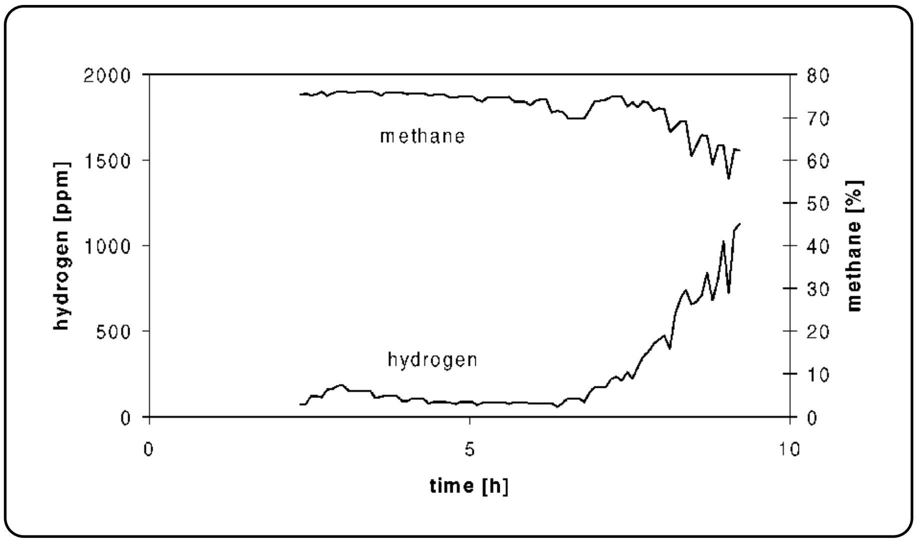

Figure 2.3: Methane and hydrogen levels - no control. (Murnleitner et al. 2002).

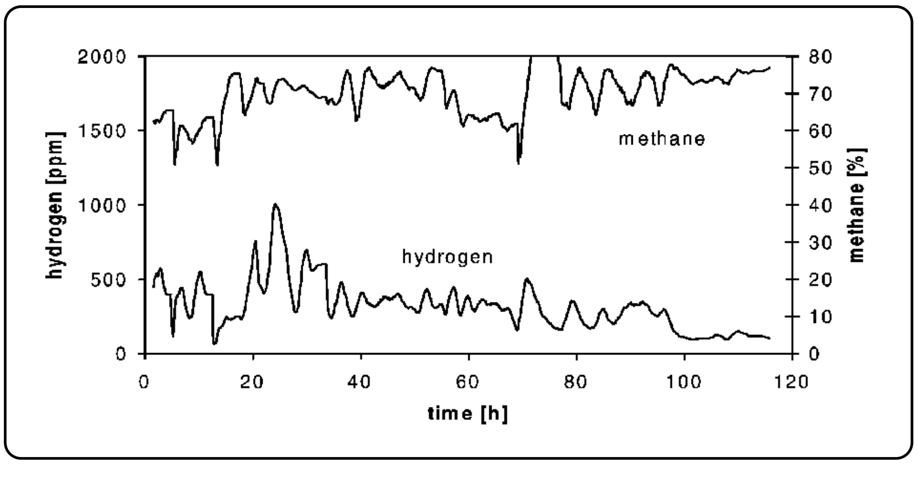

Figure 2.4: Methane and hydrogen levels - with control. (Murnleitner et al. 2002).

Figure 2.3 shows methane and hydrogen levels leading up to an overload. This data was gathered without control. Notice distinctive increases in hydrogen and corresponding drop in methane production. This is due to the inability of methanogens to consume the available hydrogen produced earlier in the fermentation process, and results in a corresponding, though delayed increase in VFA's and reduction in pH. Figure 2.4 shows methane and hydrogen levels with the controller. No such indication of overload is present.

Muller et al (Muller, Marsili-Libelli, Aivasidis, Lloyd, Kroner & Wandrey 1997) adopt an even simpler approach using a fluid bed reactor; hydrogen level and biogas production are used to distinguish between all three overload conditions using this simple method. Problems with this system could arise if feeding is irregular, such as with a batch type digester (Murnleitner et al. 2002).

Whilst these two experiments have made great gains in controlling the digestion process, the fluid bed reactors in use rely on the co-ordinated use of two vessels with sludge recirculation and buffering systems. Further advantage is taken of individual conditions between the stages of the fermentation processes. These systems are able to greatly stabilise the microbial environment and are fine models for automated, large-scale wastewater installations. For the objectives of this project, the laboratory conditions are stable and equipment is considered too complex to implement on small scale community and farm digesters, which are often more simply constructed and operated under less than optimal conditions.

2.5 Safety Considerations

Whilst most digesters operate at pressures far below the limits set for pressure vessels in AS 1210-1997, there is a significant risk of excessive pressure in a digester resulting in the combustion-less explosion of the vessel. This can be due to a number of possible fault conditions, such as a blocked gas or sludge outlet, or incorrect valve operation. Therefore, some kind of safety pressure relief arrangement is commonly recommended, burst disks being about the simplest and most reliable. Being a once-off solution, these have the disadvantage of replacement in the event of a breach.

Methane is an odourless gas and mixtures of 5-15% methane in air pose a substantial risk of explosion resulting in serious injury or death. AS 4654-2005 Table 3.1 imposes an upper limit of 0.2 mol% of oxygen in mains gas supplies. Air can be present during the start-up of a new digester or due to leaks, although the internal pressure of the system usually means a loss of biogas rather than the entrance of oxygen.

Hydrogen sulphide (H_2S) is also extremely toxic to humans and 0.002% is the maximum allowable concentration for prolonged exposure. Higher concentrations of H_2S poses a significant risk of injury or death.

The likelihood of excessive concentrations of CH_4, CO_2, H_2S or any gas in a confined space is slight, but may result in serious illness or death.

Safety measures such as regular inspection for leaks, effective ventilation and elimination of ignition sources minimise the risks and shall be followed at all times throughout the duration of this project. It is also desirable to have an additional device to allow the release of gas and/ or sludge in case of the presence of oxygen, poor quality gas, or excessive pressure in the vessel, regardless of the cause.

2.6 Chapter Summary

This chapter has outlined relevant details surrounding composition and production of biogas, and control of a digester. It concludes with safety precautions which must be adhered to throughout the duration of this project and when dealing with toxic and/ or flammable gas installations in general.

Chapter 3

Electronic Hardware

3.1 Chapter Overview

Embedded systems are small computer systems tailored to a particular application, usually powered by a microcontroller suited to the task. In order to gain useful information regarding the physical environment in which the embedded system is to operate, it must be interfaced to a series of sensors designed to provide such information. As these sensors are the only means for the system to gain this data, the success or failure of an embedded controller rests heavily on signals it receives from them. Such data concerning ambient odours or gas concentrations are often determined in electronic systems by a gas sensor array known as an electronic nose or e-nose.

This chapter contains detailed information regarding micro-controller, temperature and gas detection sensors, and e-nose circuits.

3.2 Microcontroller

Choice of microcontroller for the project was made by comparison of features, programming environment and cost. Accommodating analogue signals from the e-nose necessitates at least four analogue to digital converters (ADC's) to receive analogue

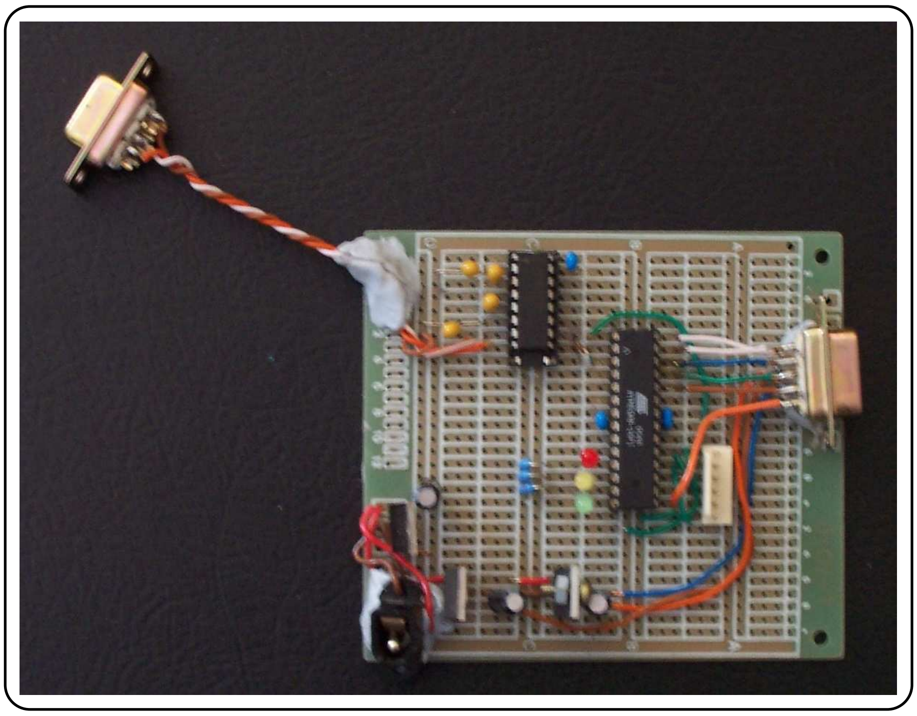

Figure 3.1: Photo of main board with Atmega 8 microcontroller and peripherals.

signals from sensors. Demand for memory space was considered to be modest, whilst large response time constants of most digester vessels places no extraordinary demands on the processor speed. Required outputs are simple digital ones or zeros to control the heater, indicator lights and ancillary equipment. Past experience with several embedded system projects has shown that pulse width modulation (PWM) is often useful if available, while input/ output (I/O) pins are often running scarce towards the later stages of an experimental embedded system project, and also carried some weight in the selection process.

Several microcontrollers were considered for the project, the Motorola HC12 family and Microchip's PIC series were also considered. The Motorola chips, despite having many relevant features and specific fuzzy logic instructions, were considered excessive for this project. The Microchip PICs were also considered, again having many of the required features, but do not lend themselves to high language programming, and such well established development environments are not free.

The Atmega 8 microcontroller was chosen due to its availability, cost, features and versatility of programming using in-serial programming (ISP). It has an internal 1

MHz oscillator, 8 kbytes of programmable flash memory for programs and data, 512 bytes of EEPROM memory, USART for serial communication, 3 timers and a total of 23 programmable pins for miscellaneous I/O, 6 of which may be used as ADC's. Additionally, 3 pins are configurable for PWM which, although considered unnecessary initially for this application, was used to provide the two precision voltages for the H_2 sensor and may be useful for proportional heating at a later stage.

Available software development environment was also considered. For the Atmega 8, the C programming environment the development tools comprised of the avr-gcc compiler, avrlibc, avrdude programmer, and avrsim simulator permit a free software development environment for anyone using a Unix based PC. An MS Windows environment is also available. Significant literature, code snippets, makefiles and tutorials are freely available with the development tools and dedicated webpages.

The circuit diagram in Figures 3.1 and 3.2 shows the embedded controller circuit diagram with peripherals. Notice this main circuit board also provides regulated power supplies to the e-nose through the nine core shielded cable.

3.3 E-nose

E-noses in general have recently received a lot of interest due to their possible application to numerous fields such as process control, air quality monitoring, security and drug detection.

The e-nose construction comprised of three gas sensors for methane, carbon dioxide and hydrogen and a resistive type temperature sensor, all connected to an LMC660CN low power quad channel operational amplifier (op-amp). This arrangement requires regulated voltages to power the amplifier and a total of about 2 watts (W) of power to heating elements in the gas sensors. It plugs directly into the main circuit board via nine core shielded cable in order to receive all power and provide analogue signals to the analogue to digital converters (ADC's) of the micro-controller.

The sensors were placed in a chamber, necessary to allow the three sensors exposure to

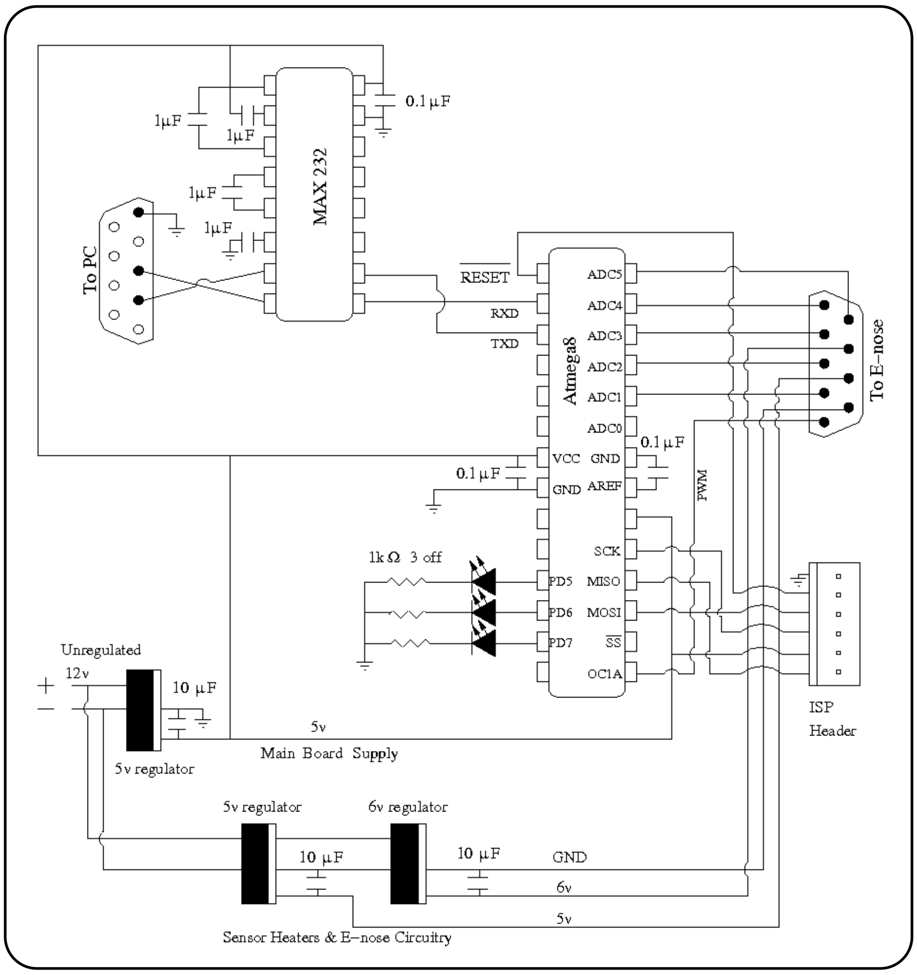

Figure 3.2: Main board with microcontroller, serial line driver and voltage regulators.

a consistant gas composition while samples were taken. To form the sensor chamber, the three gas sensors were inserted into holes in the side of an empty plastic vitamin bottle which had been washed with detergent and thoroughly rinsed. Figure 3.3 shows the completed e-nose assembly.

3.3.1 Amplifier

The LM324 op amp was originally selected for the positioning of individual op-amp channels within the package, its ability to operate from a single supply rail, and familiarity with the device.

The input configurations are covered in the respective gas sensor sections. The outputs

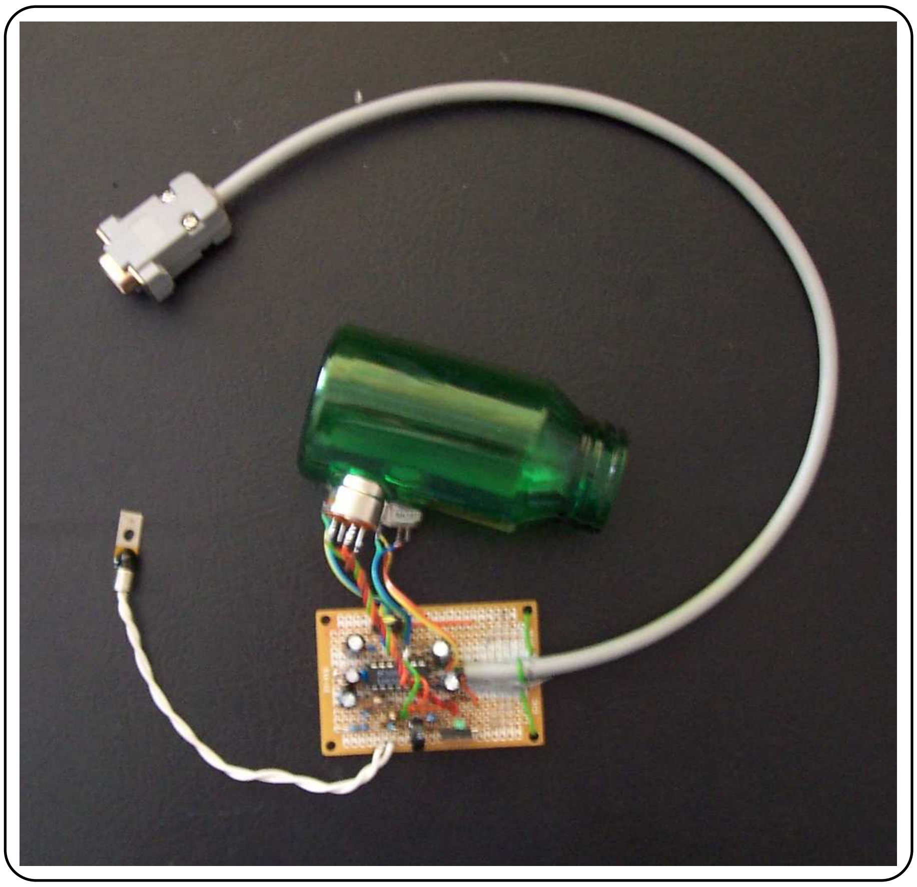

Figure 3.3: E-nose with op-amp, and temperature and gas sensors in chamber.

were connected to the individual strands of 9-core shielded cable via a one kilo ohm (kΩ) resistor to protect the amplifiers from overloading by limiting the current to approximately 5 milliamps (mA). This was then grounded via 100 micro fared (μF) capacitors to short AC components from the signal. This can bee seen in each of the individual sensor circuit diagrams.

Unusually high input impedance demands of the amplifier by the CO_2 sensor resulted in replacement of the LM324 with the LMC660CN op-amp, which is a pin for pin match with the LM324. The input impedance of over 1 Tera ohm (TΩ) and input bias and currents of only 1 and 2 pico amps (pA) respectively meet the sensor's requirements.

3.3.2 Sensors

Gas sensors were selected on the basis of individual gas sensitivity, cost and availability. Three sensors were selected:

- The CO_2 sensor was the MG811 by Hanwei Electronics.
- The MQ7 for H_2, also by Hanwei.
- The TGS2611 by Figaro for CH_4.

Manufacturer's datasheets indicate that all of these sensors return either linear or logarithmic inputs for logarithmic variations in gas concentrations. Ambiguous data sheets meant considerable trial and error to obtain acceptable results.

The temperature sensor was a simple linear resistive sensor with a resistance of 50 kΩ at room temperature. This was placed in series with two other resistors and the voltage measure by a differential feedback amplifier. The circuit diagram is shown in Figure 3.4.

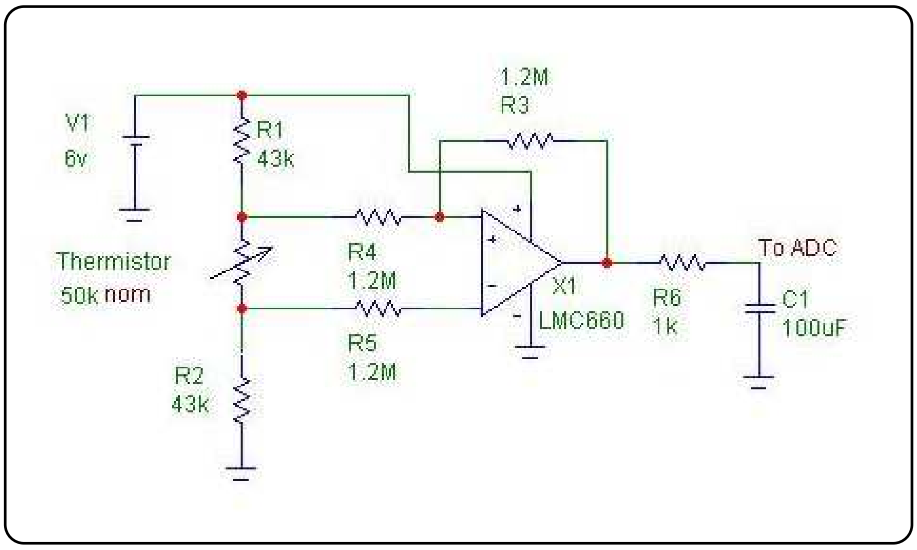

Figure 3.4: Thermistor connections to LMC660CN op-amp.

Of concern to this project is expected cross-sensitivity of the TGS2611 CH_4 sensor to H_2. It is proposed that this be resolved by choosing a hydrogen sensor which is not heavily influenced by CH_4, and then compensating in software by subtracting a scalar multiple of the H_2 reading from the methane reading. Both Figaro and Hanwei's datasheets include information regarding sensitivities to other gases.

Of additional concern is the concentrations at which these sensors can provide valid readings; composition of biogas is measured as percentages, while these sensors are generally operated in concentrations of under 10 000 parts per million (ppm) in air. The simplest method to achieve this is to precisely dilute sampled biogas with air in order to carry out reliable analysis. This is covered in Chapter 5.

3.3.2.1 MG811 Carbon Dioxide Sensor

As CO_2 sensors are the most easily obtained, much development of the e-nose and software was based on trials with these. Great confusion was caused by ambiguity of the datasheet, particularly conflicting output voltages and input impedance required of the signal amplifier. The literature specifies such amplifiers have an input impedance of 100 to 1000 GΩ, which resulted in the premature failure of two of these sensors and hours of troubleshooting.

The CO_2 sensor requires a precision 6 volt (V) supply and 200 mA for the heater. This was provided by a 3 terminal regulator on the main controller board. It is an EMF type sensor, thus producing a voltage inversely proportional to CO_2 concentration. Hanwei states that the maximum sensor output is 300 mV at atmospheric concentrations of CO_2. It is designed to return a linear EMF to logarithmic gas concentrations from 350 to 10 000 ppm. The CO_2 sensor datasheet gives no indication of sensitivity to H_2.

The op-amp was placed in non-inverting configuration with a conservative voltage gain of approximately 7.7. The maximum measured output from the op-amp was about 4.95 V, meaning the sensor was producing $4.9 \div 7.7 \approx 0.64$ v, more than double that stated in the datasheet. This results in slight clipping of the signal by the ADC at atmospheric concentrations of CO_2. Fortunately, the signal is inversely proportional to concentration, so this is of no consequence to this project, as CO_2 concentrations will never be this low. Minimum measured voltage at the output of the op-amp was 0.77 V, giving a voltage swing of 4.18 V. The circuit diagram is shown in Figure 3.5.

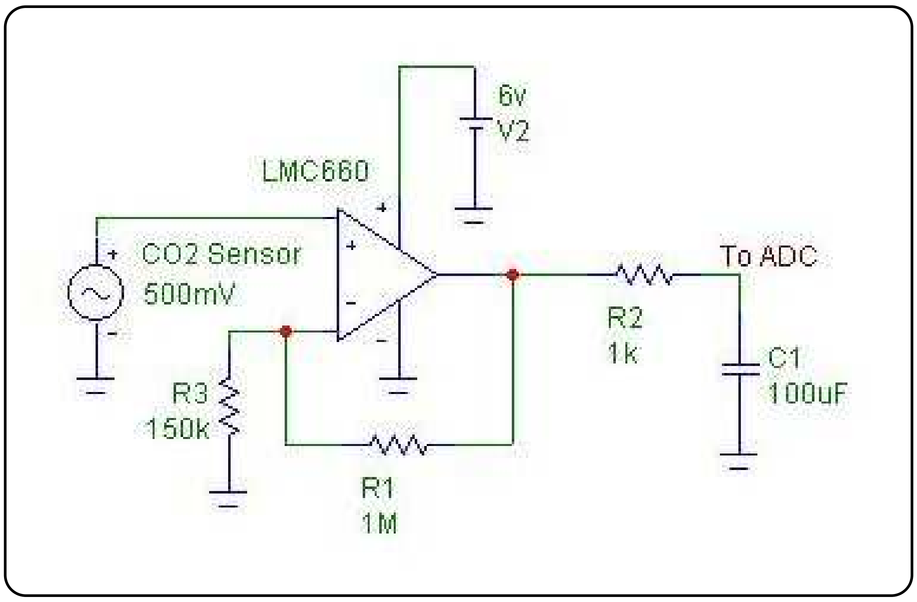

Figure 3.5: MG811 CO_2 sensor connections to LMC660CN op-amp.

The National Semiconductor datasheet gives some useful hints for working with low impedances, one of which was applied here; the output of the MG811 is soldered directly to the input of the LMC660 instead of the usual practice via the printed circuit board. This minimises the effect of leakage currents, as air is an excellent insulator. This was not necessary with any of the other sensors.

3.3.2.2 TGS2611 Methane Sensor

The TGS 2611 methane/ natural gas sensor varies in resistance logarithmically to logarithmic gas concentrations between 500 and 10 000 ppm CH_4, and is similarly sensitive to H_2. It requires a precision 5 V, 60 mA supply for its heater, again provided by the main controller board by a separate 3 terminal regulator. The op-amp was placed in a non-inverting arrangement, with unity gain. Being a resistive sensor, it was placed in an adjustable voltage divider circuit, whose output was fed into the op-amp. The resulting output voltage ranged from 0.39 to 2.67 V. The circuit is shown in Figure 3.6.

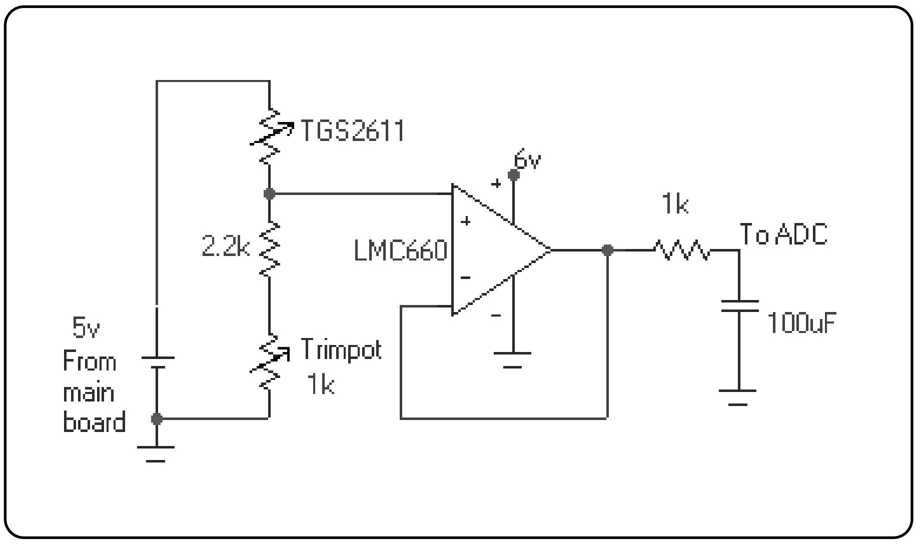

Figure 3.6: TGS2611 CH_4 sensor connections to LMC660CN op-amp.

3.3.2.3 MQ7 Hydrogen Sensor

The MQ7 H_2 sensor was the most complex to set up. Hanwei specifies a 150 second heating cycle consisting of two precision heating voltages, the first being 5 V at 150 mA for 60 seconds, and the second 1.4 V at 43 mA for 90 seconds. This ruled out the convenient use of 3 terminal regulators. Instead, one of the micro-controller's timers and PWM channels were used with a power MOSFET on the e-nose circuit board to provide the required power and timing. The MQ7's resistance varies logarithmically in response to logarithmic gas concentrations up to 2000 ppm. The op-amp was set up in a non-inverting configuration with a gain of 3, which saturated the op-amp during practical trials detailed in chapter 5 The gain was then reduced to 2, which again led to saturation. Unity gain finally resulted in an acceptable signal from 0.31 to 4.33 V. The circuit is shown in Figure 3.7.

3.4 Testing

Trails with this e-nose arrangement consistently showed noisy readings. A sample of such a signal is shown in Figure 3.8. Investigation with an oscilloscope lead to the conclusion that sources of noise were 50Hz hum and digital spikes through the power

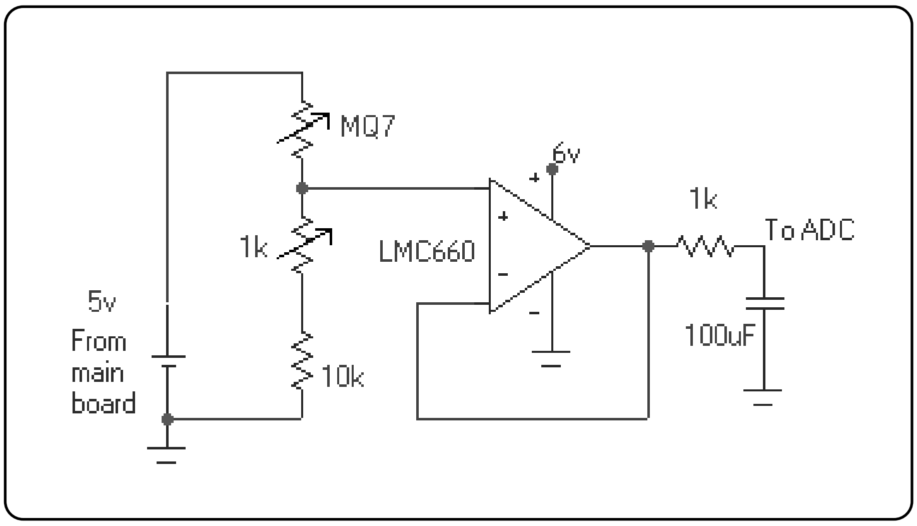

Figure 3.7: MQ7 H_2 sensor connections to LMC660CN op-amp.

supply due to the switching of the serial communication and pulse-width modulation (PWM) used to control the heater voltage of the H_2 sensor. This is of little surprise as no special precautions against noise were considered short of standard decoupling capacitors and minimal length circuit connections. The PWM spikes were approximately 0.03 V, considered acceptable and of little consequence due to compensation in the micro-controller software described in Chapter 4, and serial communications are really only necessary during software development, not during normal operation. The 50 Hz noise was practically eliminated by choosing another power supply, and remaining noise is considered negligible for the purposes of this project.

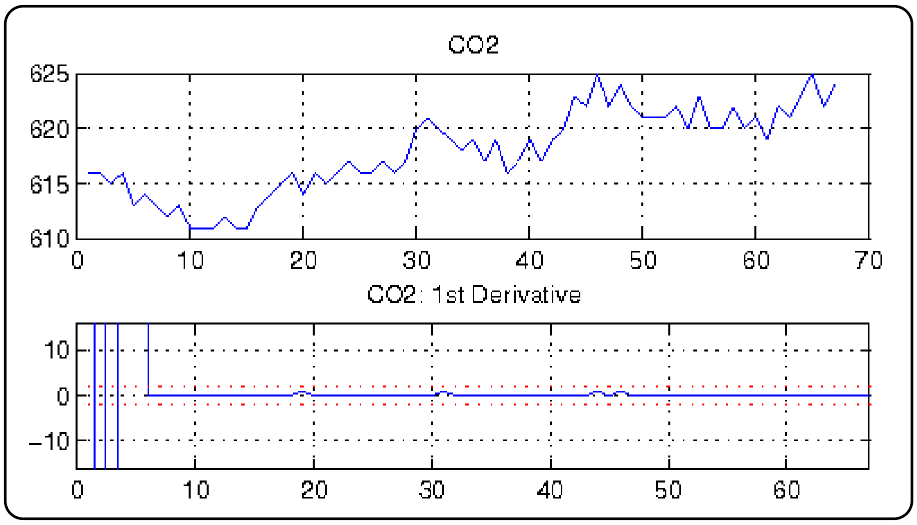

Figure 3.8: An example of a noisy signal from the CO_2 sensor.

3.5 Chapter Summary

This chapter has detailed the components, and circuits in which they have been used, as well as management, adjustments and precautions taken to assist effective function throughout the development and testing phases of the system.

It will be seen in successive chapters how the signals obtained from these circuits are used to indicate the composition of gas samples for the purpose of controlling digester temperature.

Chapter 4

Software

4.1 Chapter Overview

This chapter demonstrates the steps taken thoughout the software development process.

Three different control algorithms were considered during the initial stages of design. The first was simple on-off control, commonly known as bang bang control, due to the simple objectives of this project; to turn on or off a heater. Classical linear control theory was also briefly considered, but dismissed due to the difficulty in modelling the system due to the necessary number of inputs and the baffling number of variables involved in the biological and physical aspects of the system.

Fuzzy logic control was chosen because of its inherent ability to deal with non-linear systems, multiple inputs and noisy signals, with limited or unclear data available regarding the physical system. Prior expert knowledge may also be used as a starting point to determine the behaviour of the system.

The hardware subroutines were based on examples provided by the avr-libc documentation (Neswold 2006). The structure of the main program is a simple loop, the major steps of which are illustrated by Figure 4.1. Implementation resulted in the files main.c, adc.c, uart.c, timer.c and fuzzyfuncs.c, each so named to indicate the relevant device or other subroutines within. All MATLAB and C source code is included in the appendices.

The program firstly reads the temperature and gas sensors, calculates averages to remove noise, then determines derivatives which are passed to the function `fuzzify();`. The fuzzified data is then de-fuzzified by the function `defuzzify();` to obtain a crisp output which in this case, is an ideal target temperature intended to stimulate the methanogenic bacteria to optimise CH_4 production against digester heating requirements. This temperature setpoint is then compared to the actual digester temperature, semaphore lights and heater being activated or de-activated accordingly.

4.2 Implementation

Initial efforts focussed on the reading of the ADC and placing the value in the USART. This is initiated on the overflow of timer 1 in the micro-controller. Initial sampling time periods during development were every half-second, but were scaled up to 2.5 minutes in the working versions to suit the H_2 sensor. The USART output consists of 8 bit data words with 1 stop bit added at 2600 baud, which once converted to RS232 signal levels, is readable by any serial communications terminal software, Minicom being used here.

The function `void uart_put_num(float)` was written to convert binary numbers to ASCII words for sending over the USART for communication with a PC. One shortcoming of this subroutine is its inability to deal with floating point numbers, which are simply converted to integers. This was used straight from the avr-libc examples and the benefits not considered worth the time or memory necessary to rectify. The implications of this will become apparent later in this chapter.

It was decided to use only the first derivatives as inputs to the controller so that rates of change in gas concentration trigger responses. This eliminates the necessity of accurate calibration of the equipment with respect to an absolute reference. Despite several texts recommending the use of second derivatives, it was found that noise passed from the gas sensors was too similar to the real second derivatives to be reliable.

Derivatives were originally obtained by subtracting the previous ADC reading from

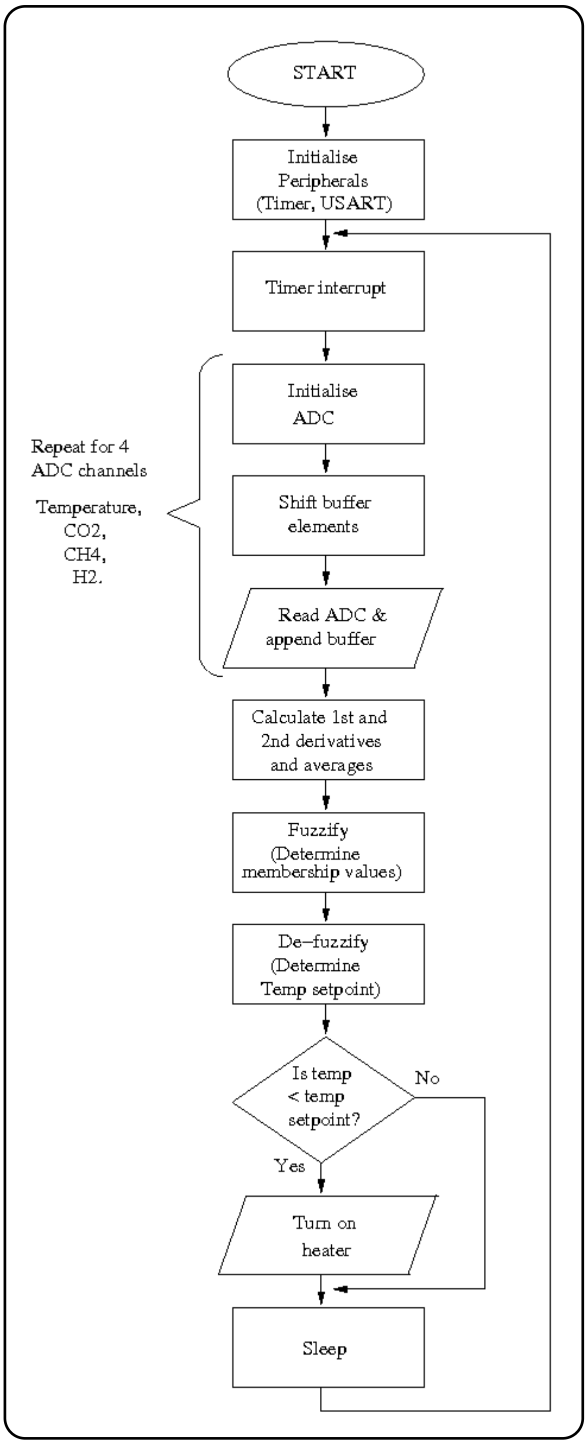
START
Initialise Peripherals (Timer, USART)
Timer interrupt
Initialise ADC
Shift buffer elements
Read ADC & append buffer
Repeat for 4 ADC channels
Temperature, CO2, CH4, H2.
Calculate 1st and 2nd derivatives and averages
Fuzzify (Determine membership values)
De–fuzzify (Determine Temp setpoint)
Is temp < temp setpoint?
No
Yes
Turn on heater
Sleep

Figure 4.1: Main function flowchart.

the latest, as is common practice with computer systems. Cases were noted when noisy signals resulted in incorrect derivative values over longer periods of time. This was rectified by using buffer arrays to provide several consecutive readings. An running average is then taken of the entire buffer and stored in a separate array of 3 consecutive averages for each gas. While only 2 are necessary, a buffer of 3 allows for the calculation of a second derivative if deemed useful later. Although this method slows the response of the controller, it does significantly increase the reliability of detected changes in gas concentrations over longer periods. It was found that buffer sizes of 6, 5 and 6 elements provided adequately stable CO_2, CH_4 and H_2 readings respectively, but these sizes can easily be adjusted in the preprocessor `#DEFINE GAS_buffer_size` lines as necessary.

4.3 Fuzzy Control Software

Fuzzy algorithms were based on Mamdani type fuzzy logic controller, originally developed by Doctor Lofti Zadeh and further developed by Professor Ebrahim Mamdani (Reznik 1997) and (Sowell 2005). The principles of their designs will be detailed throughout this section. The functions are located in the file fuzzyfuncs.c.

Once readings from the ADC's were being successfully read and sent to the USART, the fuzzy logic functions were created to determine an acceptable temperature set point between the stated 21 and 38 degrees C using the centre of gravity (COG) method. These were first implemented using MATLAB script and then ported to C. Additional MATLAB script was developed to graphically display memberships and responses, which greatly simplified the process of matching fuzzy logic parameters to sensors.

4.3.1 Fuzzifying

For the purpose of this system, a detected gas concentration or its derivative may be regarded as low, medium or high. These are assigned based on prior expert knowledge of the system and are known as membership functions. This expert knowledge may be gained from a person familiar with the operating characteristics of the system to be controlled, ideally an experienced operator. Each input is assigned a value between 0 and 1 to each membership function, of which there may be as many as necessary to achieve the desired control characteristics. It is common to have membership to more than one membership function. This implementation monitors chosen variables and assigns them a value of membership between zero and one to the membership functions in the software '`LOW`', '`NORM`', and '`HIGH`'.

The exact assignment of a membership value is dependent on geometry built into the system by the designer. For this implementation, `LOW` and `HIGH` are trapezoidal, while `NORM` is triangular. These shapes were chosen due to their simplicity of implementation and simple area centroidal calculations. The positioning and dimensioning of the geometry lies within the upper and lower expected extremities of the range of the variable being monitored. This range is known as 'the universe of discourse' in fuzzy terminology. The triangular geometry assigns a value which varies from 0 to 1 and back to 0 again as an input value varies from the lower limits of what may be considered normal operating conditions. The trapezoids are suited to outer geometry in this application, as they are able to assign membership for values of 1 outside the universe of discourse, therefore maintaining control outside of normal operating ranges, while gradually reducing membership as an input returns to normal. All geometry is set by three, three-element arrays, '`LOW_membs`', '`NORM_membs`', and '`HIGH_membs`' in the program code. The first element of each array corresponds to the lowest point of the geometry on the universe of discourse, the second element sets the centre point, and the third element dictates the highest point of the geometry within the universe of discourse.

These concepts are best illustrated by example. The arrays used in the initial MATLAB scripts were as follows;

```
LOW_membs  = [-16 -16  -2];
NORM_membs = [-16   0  10];
HIGH_membs = [  2  10  16];
```

providing input membership functions which look like;

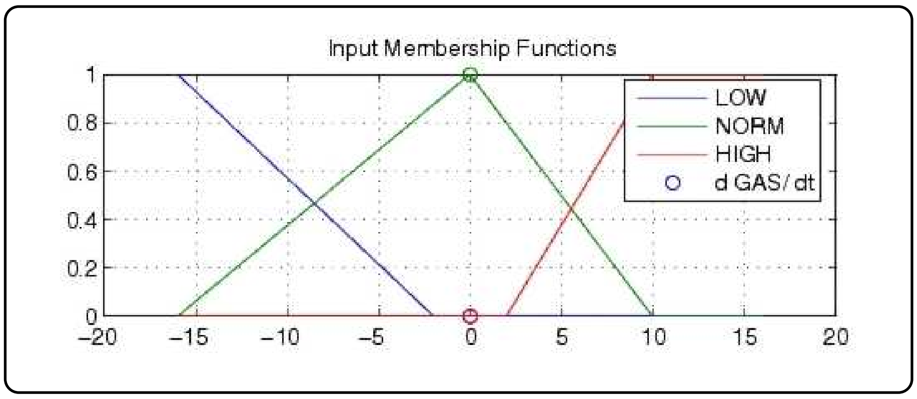

Figure 4.2: Controller initial input membership functions used to fuzzify.

with an input equal to zero, as indicated by circles plotted on each of the membership functions. Here an input of zero returns a membership value of one for NORM, and zero each for LOW and HIGH. Of importance in this example is the correlation between the values within each array and the corresponding point along the corresponding membership function.

Close inspection of the LOW and HIGH membership functions close to zero reveals that membership to these functions does not occur until an input of -/+2 respectively. This was done to provide immunity to noise, for reasons which will become clear later in section 4.3.3.

The first element of the `LOW_membs[]` and the last of the `HIGH_membs[]` arrays are not actually used by the fuzzy algorithms, they are only used in the MATLAB scripts to determine the range of plots.

The code mechanisms used to arrive at a membership value are simply '`if()`' tests followed by statements consisting of linear equations of the form $y = mx + c$, or assignments of one or zero.

4.3.2 De-Fuzzify

The determined membership values `LOW`, `NORM` and `HIGH` are then used as the height in a second group of pre-determined geometric response areas, which may represent appropriate responses such as 'increase temperature', 'maintain temperature', or 'decrease temperature'. Obviously this has a direct impact on the magnitude and distribution of area, and therefore the position of the centroid, which is how the temperature set point is calculated. This is the principle of Mamdani's fuzzy controller. The centroid is calculated using equation 4.1 where A_n is nth area, z_n is the centroid of the nth area, and c_n is a scalar weighting constant of the nth input. This allows a designer to emphasise the influence of a particular input over the others. It may be observed that the formula lends itself well to expansion to include multiple response areas. This allows simple monitoring of multiple inputs to determine a single output.

$$Z = \frac{\sum_{n=1}^{n} c_n A_n z_n}{\sum_{n=1}^{n} c_n A_n} \tag{4.1}$$

The Mamdani fuzzy logic process is not easy to visualise through literature. Plots in Figures 4.3 to 4.7 below show the fuzzification and defuzzification and thus response of the controller to several progressive values of the first derivative of CO_2.

As is the case with the input membership functions, the response geometry is set in the arrays '`DN_TEMP_membs`', '`AB_RITE_membs`' and '`UP_TEMP_membs`', which are included below as implemented in MATLAB script. The upper plot in each of these figures illustrates the membership value for each input function and is plotted as a small circle, consistant with Figure 4.2. The lower plots show the response functions and the temperature set point. Note that the height of each response area is dependent on the value of the membership value of the corresponding input membership function. It is the sum of these areas for which the centroid determines the set point.

CO_2 input:	CO_2 output:
`CO2_LOW_membs  = [-16 -16  -2];`	`CO2_DN_TEMP_membs = [15 15 35];`
`CO2_NORM_membs = [-16   0  10];`	`CO2_AB_RITE_membs = [25 30 37];`
`CO_2HIGH_membs = [  2  10  16];`	`CO2_UP_TEMP_membs = [29 39 45];`

Figure 4.3 shows the controller's response to an input outside the universe of discourse. Notice that the temperature setpoint will never recede below the the COG of the `DN_TEMP` triangle, as the geometry is set in the `DN_TEMP_memb` array.

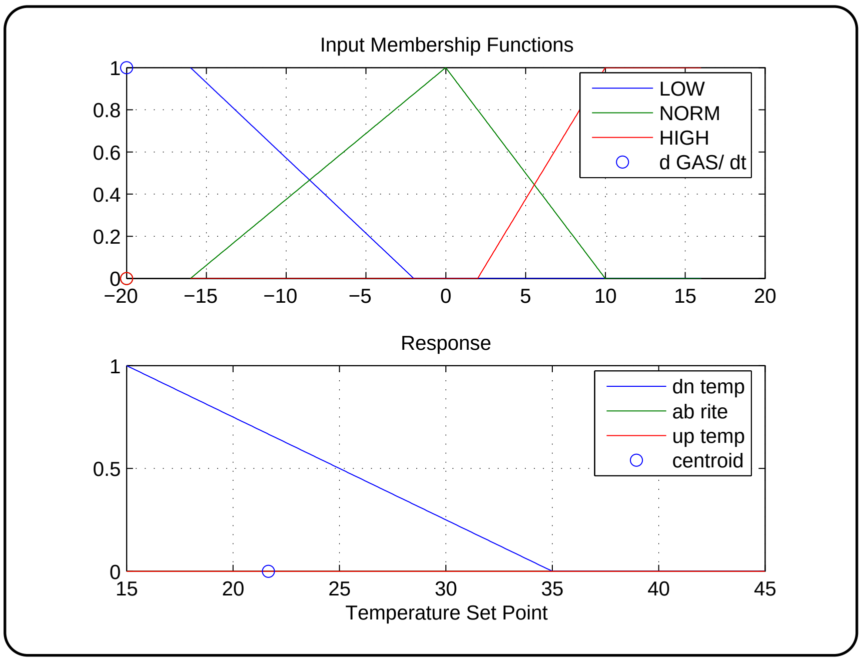

Figure 4.3: Controller response to an input of -20.

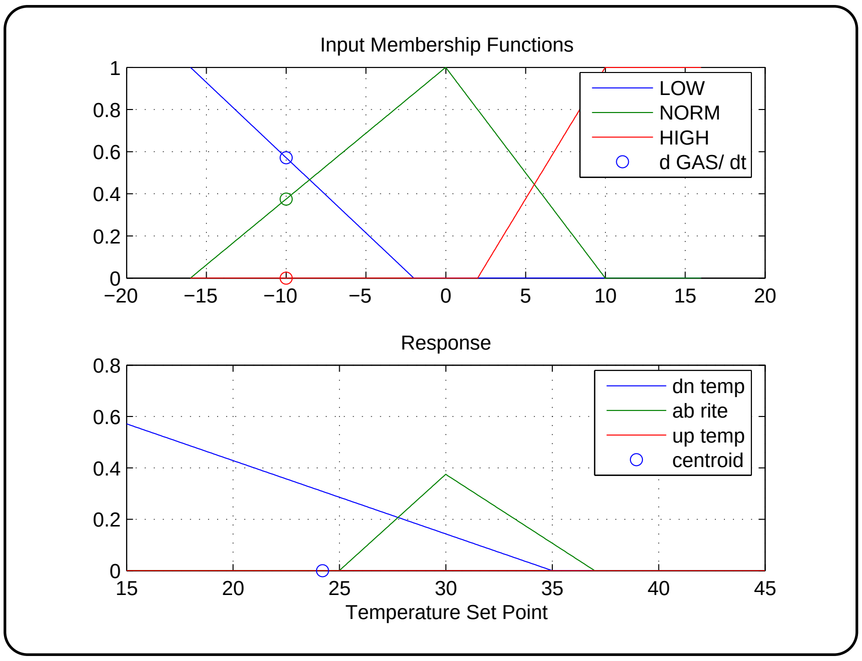

Figure 4.4: Controller response to an input of -10.

Figure 4.4 depicts the input value rising to -10. Notice that the input has now been assigned membership to two functions, `LOW` and `NORM`. These determine the height of the `DN_TEMP` and `AB_RITE` triangles, the areas of which are summed and the COG determined.

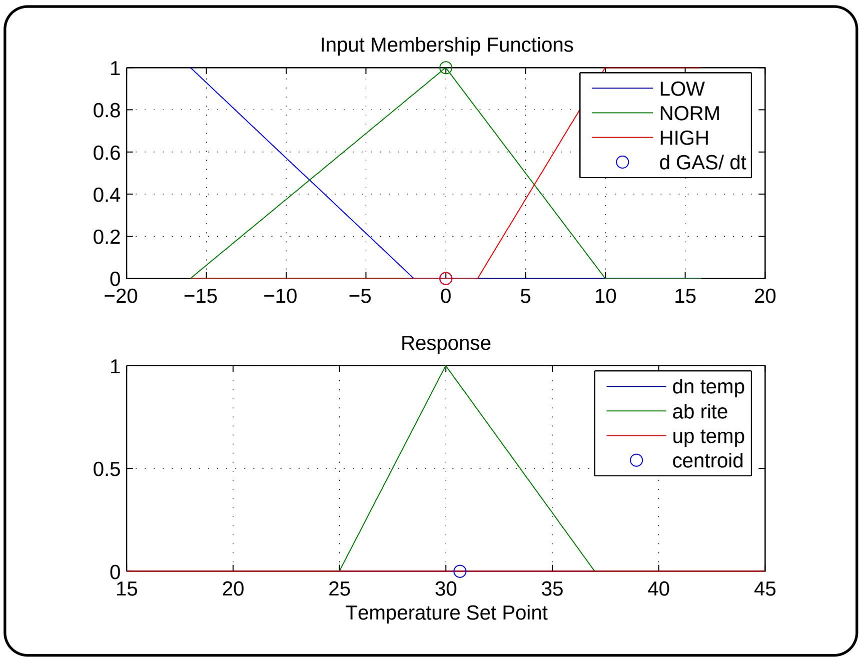

Figure 4.5: Controller response to an input of 0.

Figure 4.5 exhibits the selected ideal operating point of the digester. Small fluctuations of up to ±2 in input will not affect the setpoint, as the `DN_TEMP` and `UP_TEMP` response triangles have a height of 0.

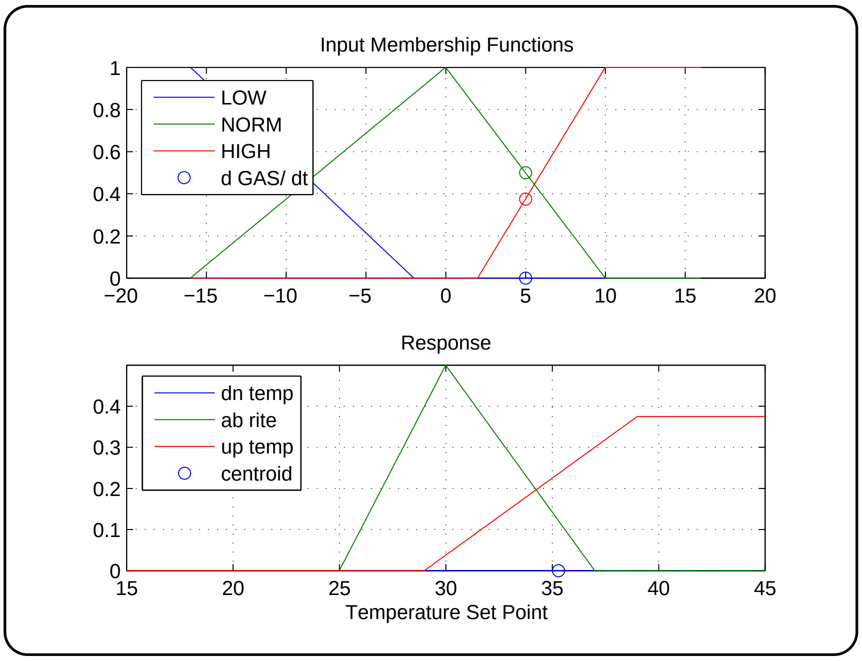

Figure 4.6: Controller response to an input of +5.

The response of the controller to an input of +5 is shown in Figure 4.6. Membership to input function `NORM` and `HIGH` determine height of `AB_RITE` and `UP_TEMP` response triangles.

Figure 4.7 depicts the response of the controller to an input exceeding the universe of discourse. Again, temperature setpoint cannot exceed the upper limit of the COG of the `UP_TEMP` response triangle.

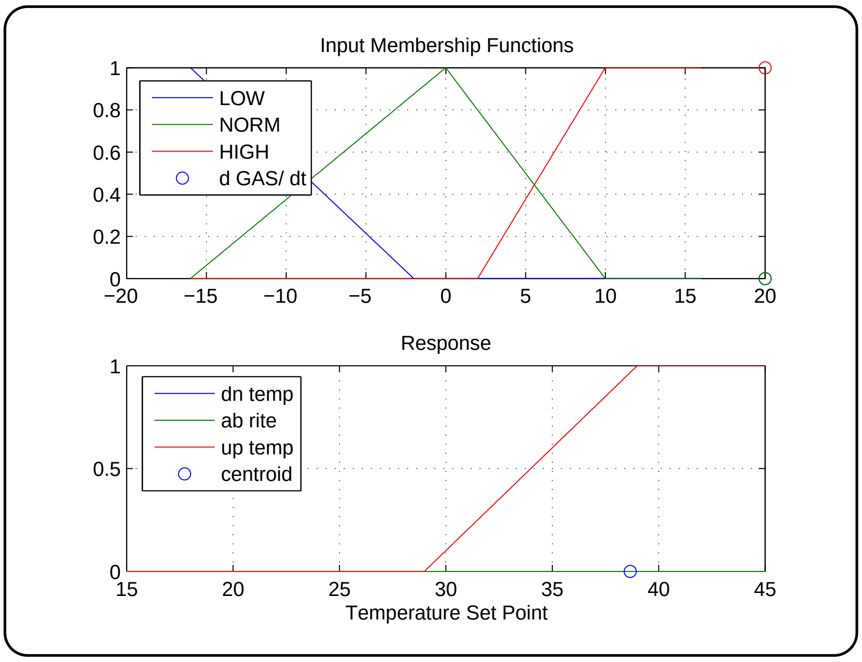

Figure 4.7: Controller response to an input of +20.

4.3.3 Tuning Fuzzy Parameters

Mentioned in the previous section is the point that expert knowledge of a physical system may be used to set the fuzzy parameters. These parameters are present in the `*_membs[]` arrays declared in the main file. Six of these must be declared for each gas, three each for input and output. The program requires values which form trapezoids for two outer membership functions and triangles for those in the centre. The temperature set point for the range of expected inputs or universe of discourse, in this case the first derivative of CO_2, is plotted in Figure 4.8.

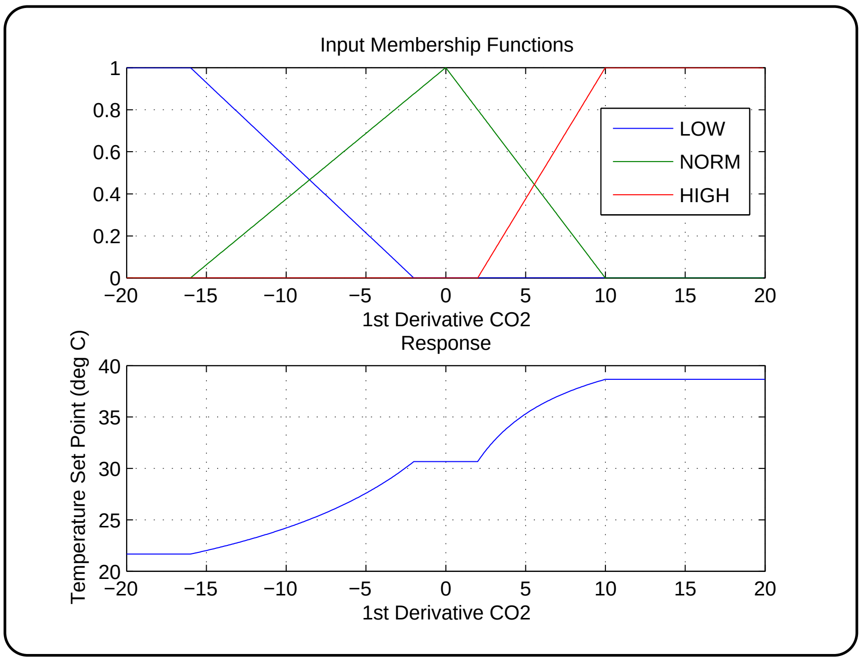

Figure 4.8: Temperature set point for expected d(CO_2)/dt levels.

Of interest are the slopes either side of the above-mentioned dead zone either side of inputs of zero, for two reasons, which will now be discussed.

4.3.4 Slope

The first reason for interest in these two slopes may have already been noticed; the gradient influences the magnitude of the response, which in this case, is the magnitude of the change in temperature setpoint for a given change in input. Figure 4.9 shows an experiment with this relationship using the following input membership arrays:

```
LOW_membs = [-16 -10 -2];
NORM_membs = [-10 0 6];
HIGH_membs = [2 6 16];
```

Comparing Figure 4.9 with 4.8, it may be appreciated while steeper gradient has the effect of increasing the magnitude of the response over small fluctuations in input, it also considerably reduces the range of inputs over which the controller is able to respond.

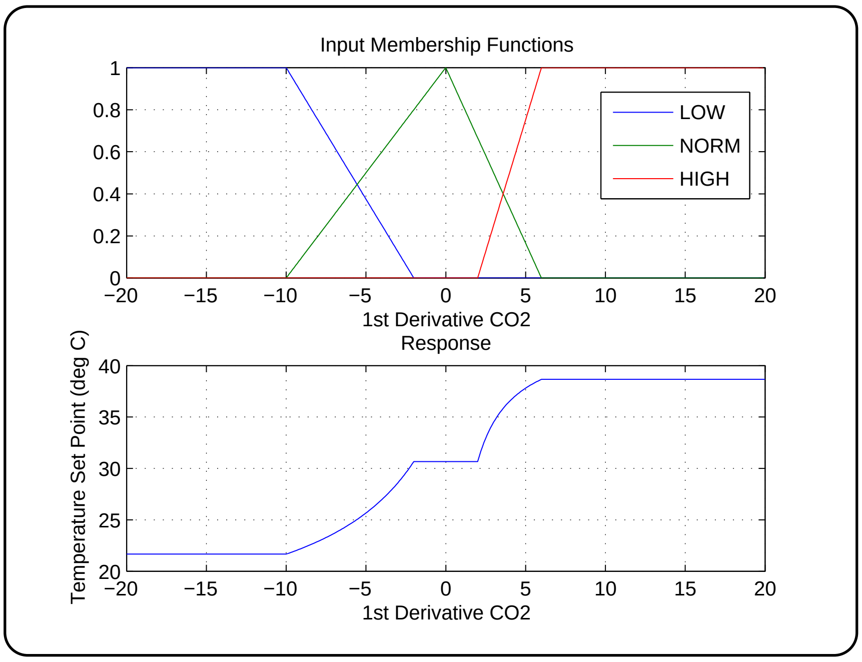

Figure 4.9: Modified Temperature set point for expected d(CO_2)/dt levels.

4.3.5 Concavity

Secondly, while the profile of Figure 4.8 could well result in a functional controller, the concavity of these two curves means that temperature set point adjustments are smaller close to the upper and lower limits of the operating temperature range. Conversely, responses are more dramatic close to the centre flat spot. This situation means that for small inputs outside the dead zone, more dramatic adjustments are made.

Great advantage may be taken of this characteristic. As it is more likely that readings close to zero may be caused by noise, an alternative was investigated by altering parameters in the `*_membs` arrays to reverse the concavity of these two areas as in Figure 4.10. This has the effect of triggering smaller adjustments in set point close to zero, which are likely to be noise, while readings further from this region will trigger more dramatic responses, where they are more necessary.

Figure 4.10 was created by widening the base of the `AB_RITE` response triangle by initialising array `AB_RITE_mems = [10 30 52]`. Further adjustments to input functions are also possible, but responses are limited by the simple geometric algorithms applied.

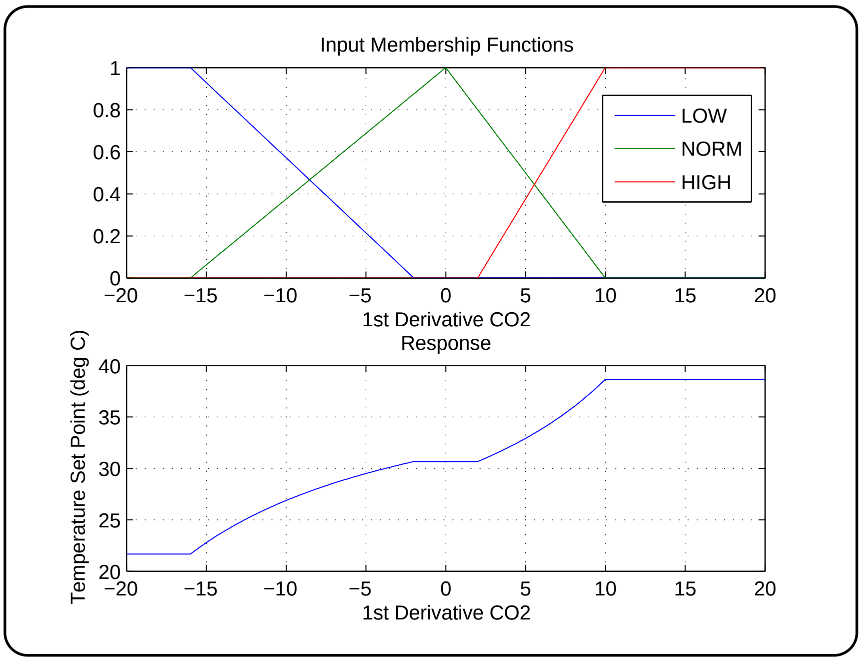

Figure 4.10: Modified temperature set point for expected d(CO_2)/dt levels.

Further exploration into the effects of various membership function shapes are thought to be of limited additional benefit to this project and so are left as specific areas worthy of curiosity and nothing more.

4.4 Micro-controller Implementation

Once the MATLAB routines were ported to C, experimentation with the micro-controller could begin. Minicom was used to capture ASCII text in data files which were then read into a simple MATLAB script for plotting.

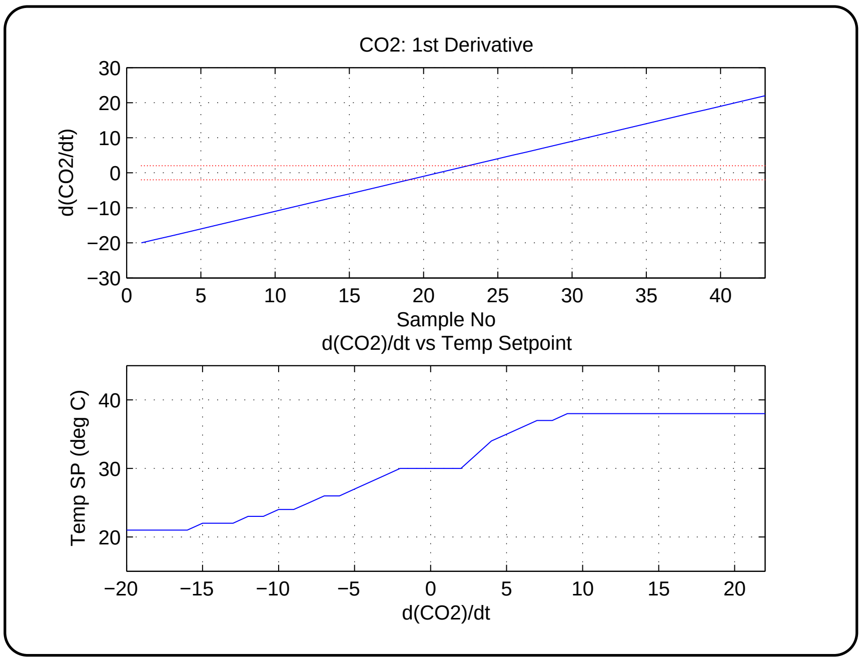

Figure 4.11: Captured temperature set point for expected d(CO_2)/dt levels.

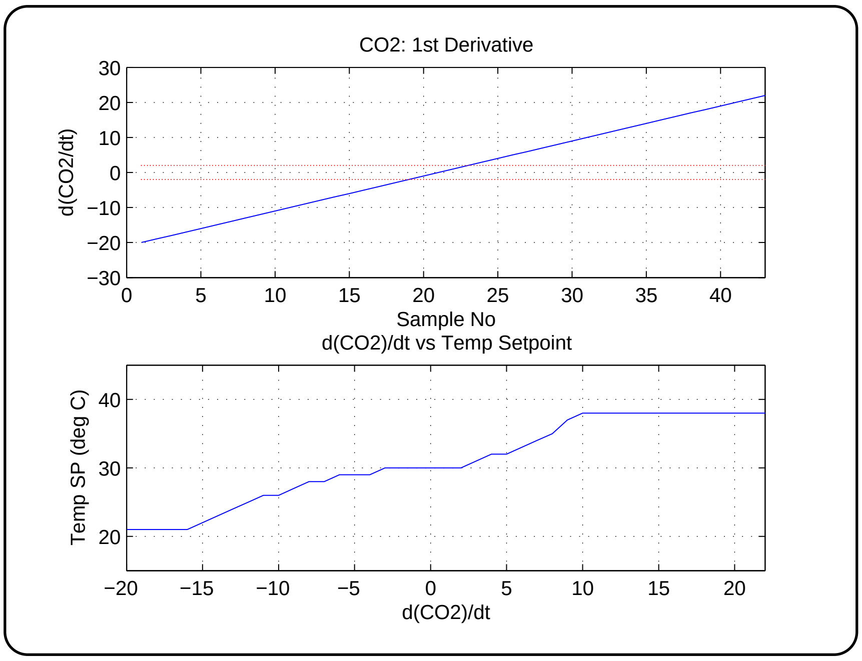

Figure 4.12: Captured modified temperature set point for expected d(CO_2)/dt levels.

It is here that the limitations of the simple function `void uart_put_num(float)` become apparent. However a clear correlation between the plots in Figures 4.11 and 4.8, and 4.12 and 4.10 can be easily identified.

4.5 Chapter Summary

This chapter has detailed the process of software development for the fuzzy logic controller for a biogas system. The process has been illustrated using examples from the first derivative of CO_2 readings. Changes to the gradient of the controller's response and the ease with which parameters may be adjusted to suit sensor response characteristics leads to the conclusion that this procedure may be followed to develop suitable controller responses for all analogue inputs required throughout this project.

Chapter 5

Laboratory Gas Tests

5.1 Chapter Overview

This section explains the equipment and process used to test the response of the sensors to expected gas concentrations and levels. The data is required to tune fuzzy parameters to individual sensors. It concludes with comments regarding the suitability of the signals for use with the `fuzzify()` and `defuzzify()` functions.

5.2 Test Equipment

The necessary function of the sampling equipment is to precisely dilute a gas sample to a known concentration and expose the sensors to a consistant concentration for a period long enough to allow the sensors to respond.

Hypodermic syringes of 6 and 3 mL volume were used to take accurately measured gas samples. Considering the implications of 1mL variation in volumes of the sample gas and air with which to dilute the samples, accurate measuring of air was not considered so crucial as precision. A plastic 600 ml bottle was cleaned and pierced on the bottom to provide a snug fit for the syringes, one of which was then inverted and inserted into a larger vessel containing purified water to form a gasometer. With these two consitant volumes, gas samples could be diluted with sufficient precision to a desired concentration for analysis.

A sketch of the essential components of the apparatus is shown in Figure 5.1. Arrows indicate the direction of applied force during a dilution operation.

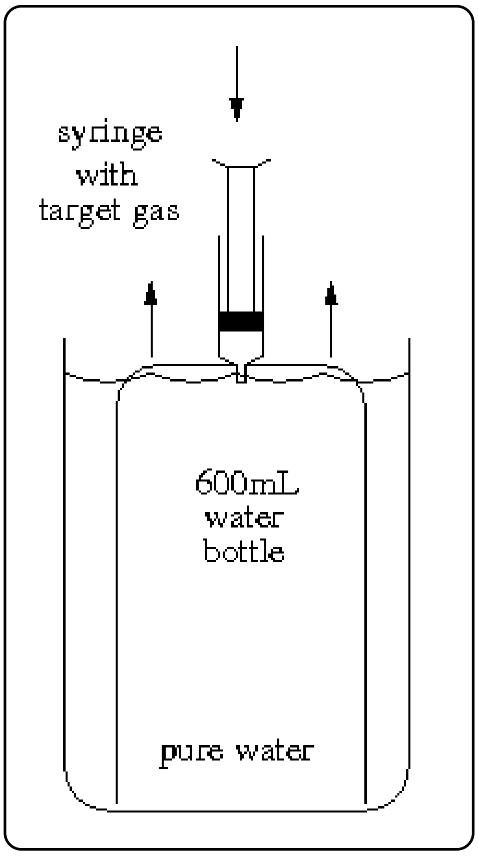

Figure 5.1: Dilution equipment used throughout laboratory tests.

Pure test gas samples of CO_2 and H_2 were provided by the USQ Faculty of Sciences, but CH_4 was not available and so was substituted with town gas. Australian Standard 4564-2000 is somewhat arbitrary with constituent components of natural gas, the implications of which become clear in later in Section 5.4.

5.3 Sampling Procedure

Having two known volumes allowed gas concentration to be determined using the following simple calculation:

$$VOLUME_{syringe} = \frac{VOLUME_{bottle} \times CONCENTRATION(ppm)}{10^6} \tag{5.1}$$

Where $VOLUME_{bottle} = 600$ mL, and $CONCENTRATION$ is the concentration desired.

To sample a gas, a syringe was placed on the outlet of the gas regulator without the plunger and thoroughly purged. The plunger was then replaced and inserted to give the calculated volume, $VOLUME_{syringe}$ at a positive pressure.

The syringe was then quickly inserted into the hole in the base of the 600 mL bottle which was raised slightly to give a negative pressure. The syringe plunger was then depressed into the bottle until empty and the bottle raised to its maximum height in the water. The breach was then covered and great care was taken to ensure the mouth of the bottle remained below the waters surface at a consistant level. The gas sample may then be considered diluted to the greatest precision available under the circumstances.

To test the sample, the inverted pill bottle and sensors were placed above the hole and the 600 mL bottle pushed gently and consistently downward so that the diluted gas entered the sensor chamber. The cap was then promptly replaced and the equipment allowed to sit for analysis.

5.3.1 Microcontroller Configuration

As mentioned in Chapter 3, the MQ7 H_2 sensor requires a 150 second heating cycle. The timer interrupt subroutine was altered to provide the necessary sensor heating states over the required time period. Buffers were implemented as described in Chapter 4. Raw readings, averages and first derivatives were sent via the serial port and captured to file.

5.4 Testing

Unfortunately time constraints permitted only 3 useful tests, each of a single sample. The details of these are listed below.

- Trial 1: 10 000 ppm CO_2
- Trial 2: 1 000 ppm H_2
- Trial 3: 10 000 ppm town gas

From these it was expected that an algorithm based on 3 linear equations could be derived to extract individual gas concentrations, which could then be sent to the fuzzy functions to derive a temperature setpoint.

Plots in Figures 5.2 to 5.10 show the data captured in the trails. The horizontal axis represents the sample number, or time in increments of 150 s, while the vertical axes represent the voltage captured by the micro-controller's ADC's. Apparent lags in averages and derivatives are a result of updates in buffer arrays. Captured data files are included in Appendix C.

5.4.1 Trial 1

Figures 5.2 to 5.4 show data captured in trial 1.

Inspection of these plots show a dramatic response from the CO_2 sensor, with only a slight, but noisy response from the CH_4 sensor, and what may be interpreted as background noise from the H_2 sensor. Having replaced the noisy power supply, the source of this noise is unknown, it is thought that it may be a characteristic of the sensors' cross-sensitivity, or fluctuations in air quality in the laboratory, as USQ staff were cleaning and preparing equipment on the day. This aside, the result was very encouraging, as it appears that CO_2 levels affect the other sensors minimally. It can be seen in the centre plots that the averaging algorithms remove the spikes to an acceptable degree.

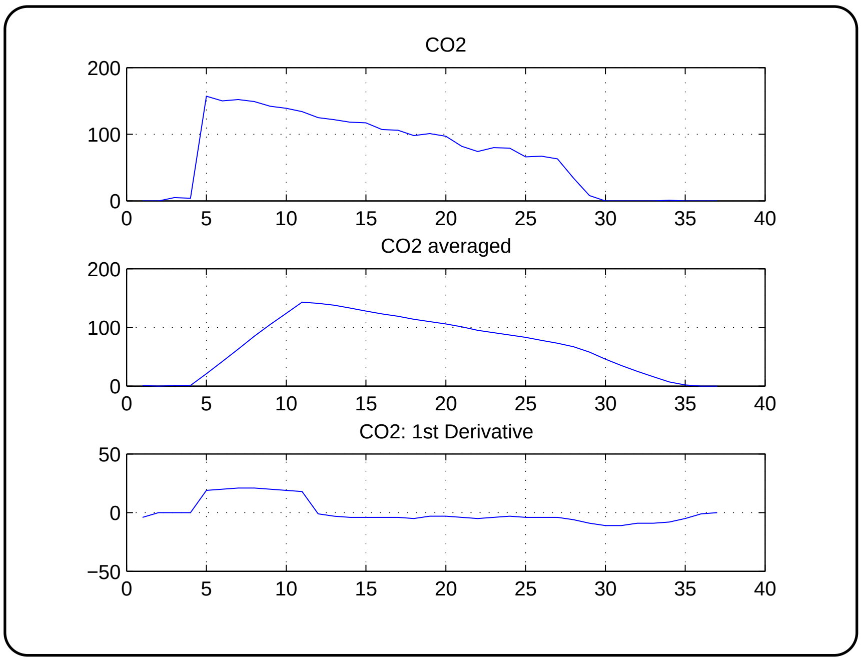

Figure 5.2: Captured data from CO_2 sensor; trial 1.

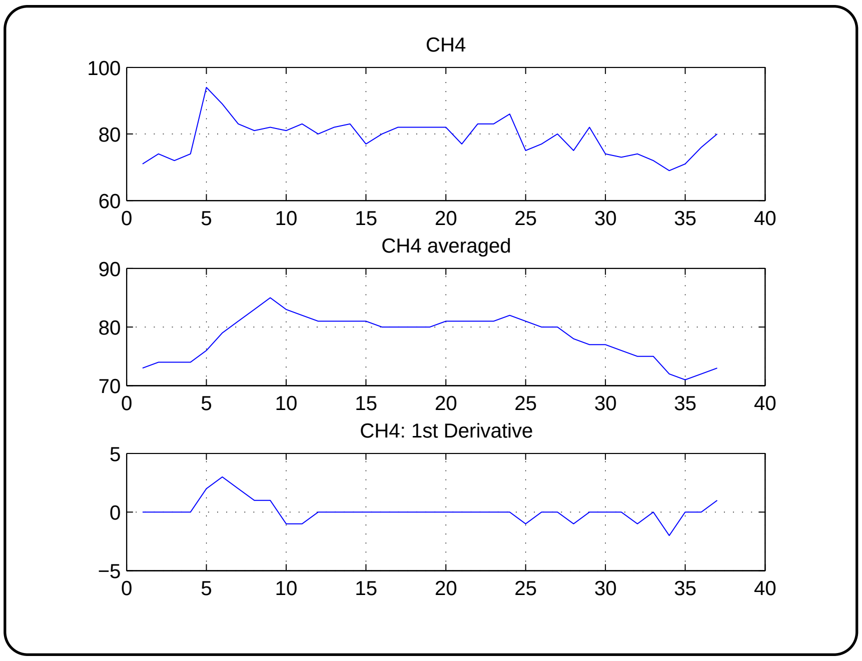

Figure 5.3: Captured data from CH_4 sensor; trial 1.

5.4.2 Trial 2

Trial 2 as defined here did not run as smoothly as expected due to excessive gain in the amplifier as mentioned in Chapter 3. Once this was rectified, the trial carried on

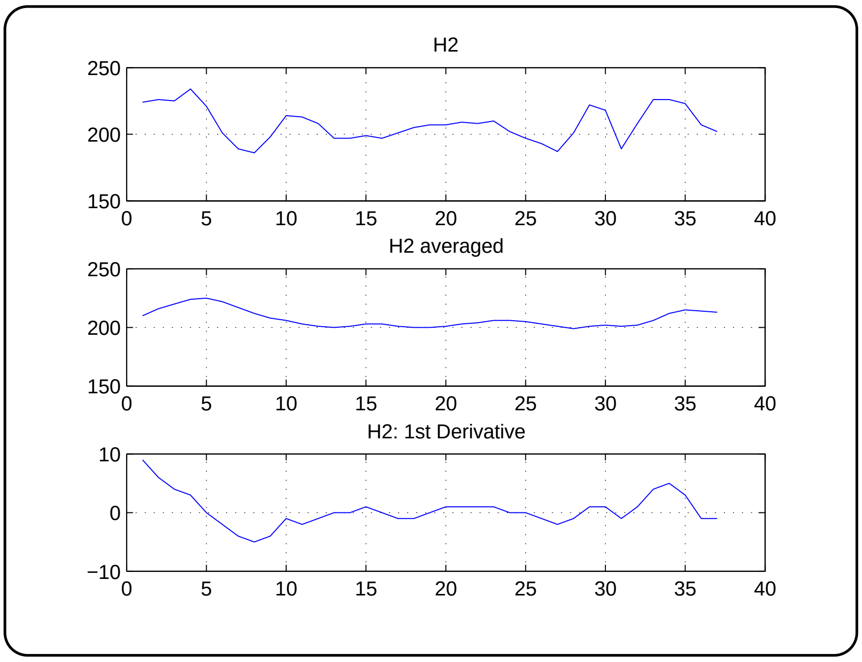

Figure 5.4: Captured data from H_2 sensor; trial 1.

as anticipated, yielding the plots 5.5 to 5.7.

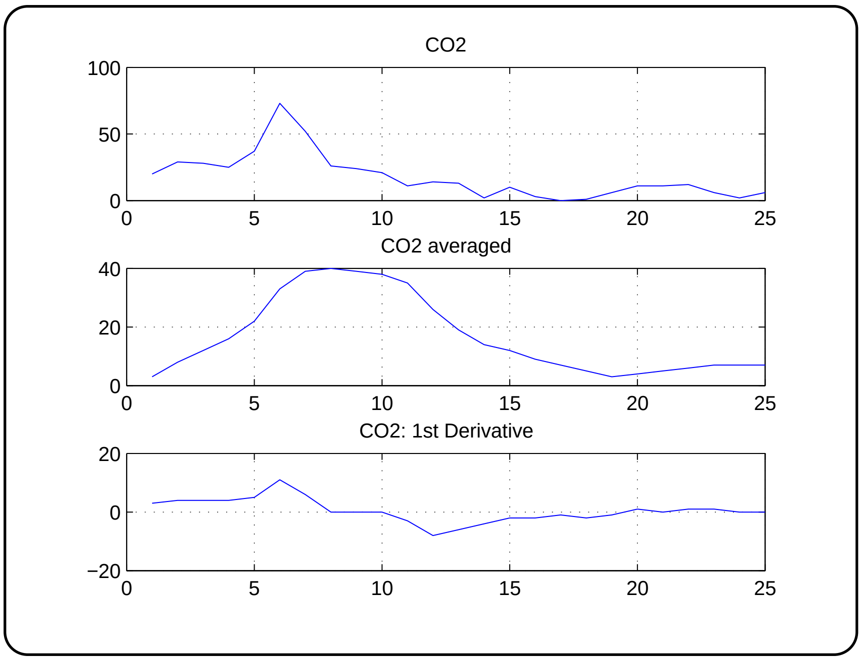

Figure 5.5: Captured data from CO_2 sensor; trial 2.

Comparing Figures 5.5 and 5.7, it appears from trial 2 plots that H_2 appears to affect the CO_2 sensor noticeably, but not excessively. This is unexpected as the MG811 data sheet mentions nothing regarding sensitivity to H_2. The CH_4 in Figure 5.6 is

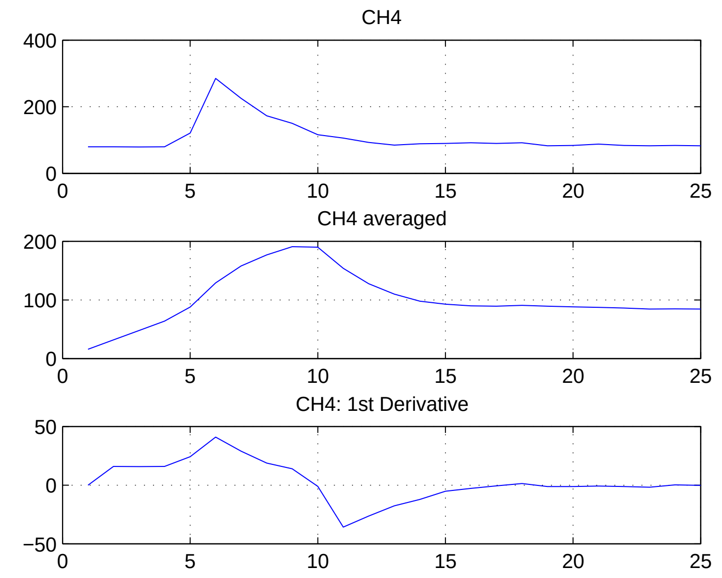

Figure 5.6: Captured data from CH_4 sensor; trial 2.

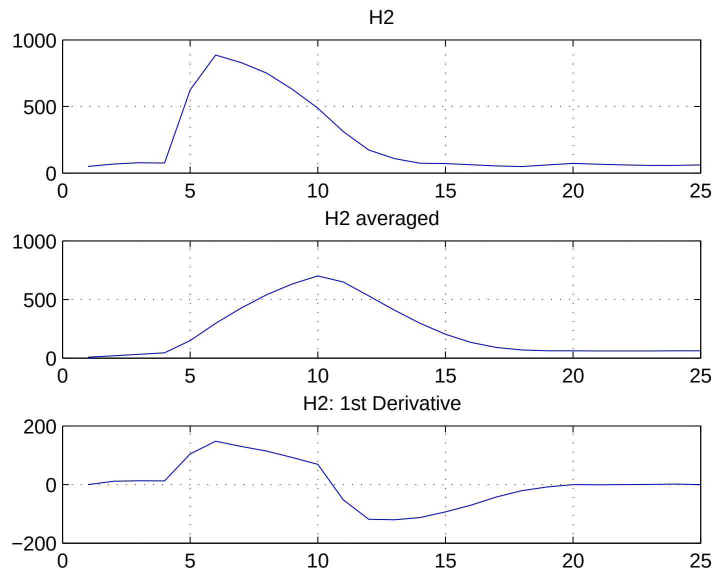

Figure 5.7: Captured data from H_2 sensor; trial 2.

also significantly affected, and this is noted in the TGS2611 datasheet. Looking at the derivative plots, both of these signals are clearly unable to be concealed by simple noise allowance techniques covered in Chapter 4, indicating some kind of compensation is necessary to extract individual gas concentrations.

5.4.3 Trial 3

Trial 3 also suffered from complications, but this time the major source is uncertainty in town gas composition.

Plots are shown in Figures 5.8 to 5.10.

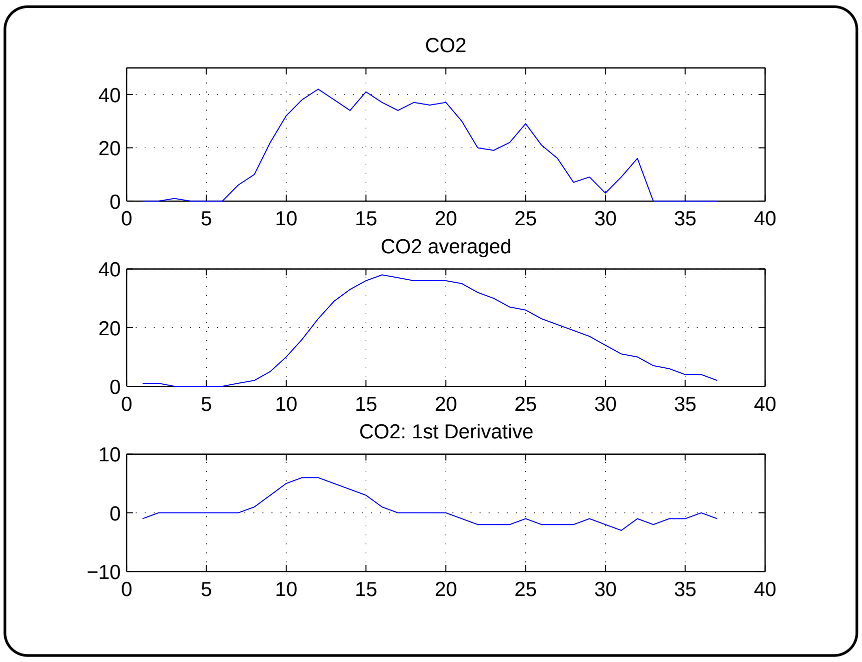

Figure 5.8: Captured data from CO_2 sensor; trial 3.

It is clear that all 3 sensors have again reacted to town gas, but is unclear at this point as to exactly why. It appears the CO_2 sensor's response closely resembles that in trial 2. The CH_4 sensor has also reacted strongly, as should be expected due to the high percentage of CH_4 in town gas. The H_2 sensor has also reacted strongly.

Unfortunately no gas spectrum analyser was available, so it is unclear to exactly what these responses may be attributed, and to what degree it/ these affect each sensor. The Australian Government (Roarty 1998) suggests that natural gas may contain propane, C_3H_8 and butane C_4H_{10}, which are the major constituents of liquefied petroleum gas (LPG). Australian Standard 4564-2000 does not impose limits on the composition of natural gas for consumption; only a minimum calorific combustion value known as the Wobbe Index of between 46 and 52 MJ/m^3. Hanwei inform that the MQ7 H_2 sensor is sensitive to LPG and also slightly to CH_4.

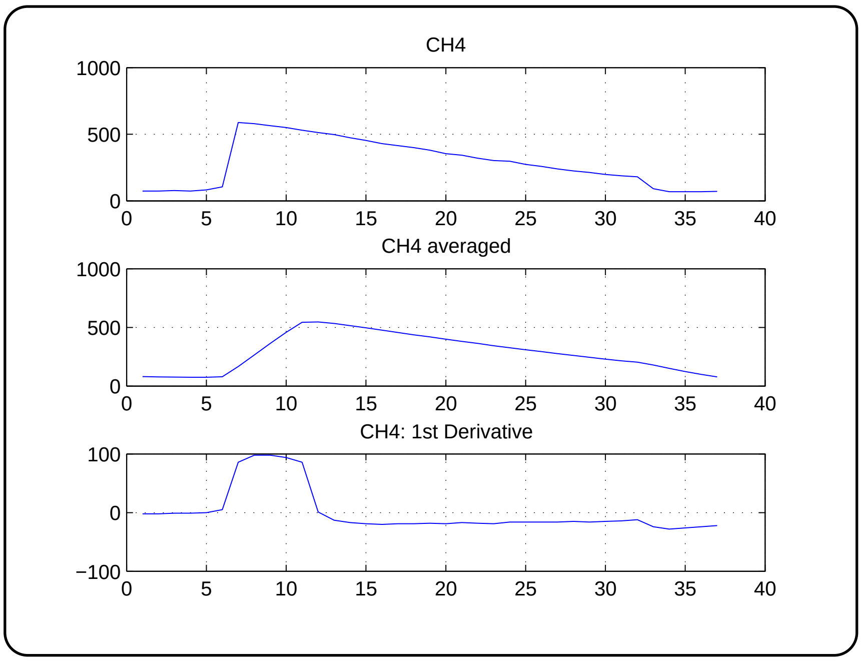

Figure 5.9: Captured data from CH_4 sensor; trial 3.

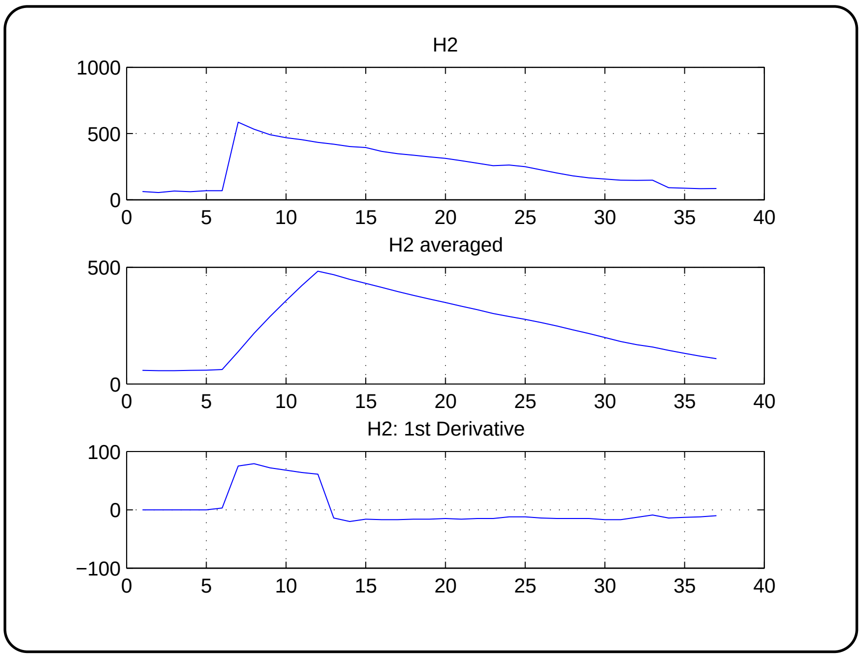

Figure 5.10: Captured data from H_2 sensor; trial 3.

Additionally, it may be expected that in the presence of such a confused array of hydrocarbon gases, the MQ7 may also be affected by other gases present, one of which could well be free H_2. In the absence of any information, excluding the datasheet, it is assumed that the MQ7's response is due to the presence of LPG and/ or free hydrogen.

The significance of this assumption will become clear in Chapter 6.

Similar conclusions may be drawn regarding the response of the CO_2 sensor. It is known that town gas does contain CO_2, but it is uncertain in what quantity, and it is also unknown to what degree the sensor is affected by other gases.

5.5 Chapter Conclusion

This chapter has detailed and commented on the equipment and procedure to test the responses of the sensors to expected gas composition.

On cursory inspection, it may appear that the unravelling of this data to provide useful information concerning individual gas concentrations, as proposed earlier in this dissertation, means that significant further work is necessary in this area, and this is indeed the subject of a great deal of recent research.

For the purpose of this project, two points must be highlighted. The first is that the method used to gather this data is highly inconsistent in nature due to the sudden exposure of the sensors to the sample, therefore resulting in rapid fluctuations in concentration of all gases simultaneously. This is not the case in a real biogas system, which with appropriate feeding and maintenance, will provide a much more consistant flow, and composition of biogas, even during an overload.

The second point is that only the first derivatives are being passed to the fuzzy logic functions. This implies that once the micro-controller's buffers are flushed and averages stabilised, it is expected that individual gas concentrations are largely irrelevant, provided they remain within the limits of the individual sensors. Further modelling in MATLAB may provide more accurate expectations of the behaviour of the controller in response to what here, initially looks like largely unusable data. This is explored in Chapter 6.

Chapter 6

Results

6.1 Chapter Overview

As mentioned in Chapter 5, data measuring the response of the gas sensors contains considerable uncertainty caused by significant cross-sensitivity between the gas sensors and unknown constituents of town gas. This may be accommodated to some degree by the fact that this controller responds only to the first derivatives of the signals. This chapter describes the procedure used to obtain and utilise data written to reflect an overload for use in a MATLAB script written to simulate the microcontroller's response.

6.1.1 Overload Data Determination

Two attempts were made to extract meaningful data for use with the MATLAB simulation scripts. These are explored in the following sections.

6.1.1.1 Logarithmic Technique

Some experimentation into the use of logarithms with linear equations was conducted. MATLAB code is included in Appendix B.1.1. Experiments focussed on the extraction of individual gas concentration in ppm $\times$ 1000, or less conventionally, parts per thousand. Plots using data from trial 3 using town gas in Chapter 5 are included below.

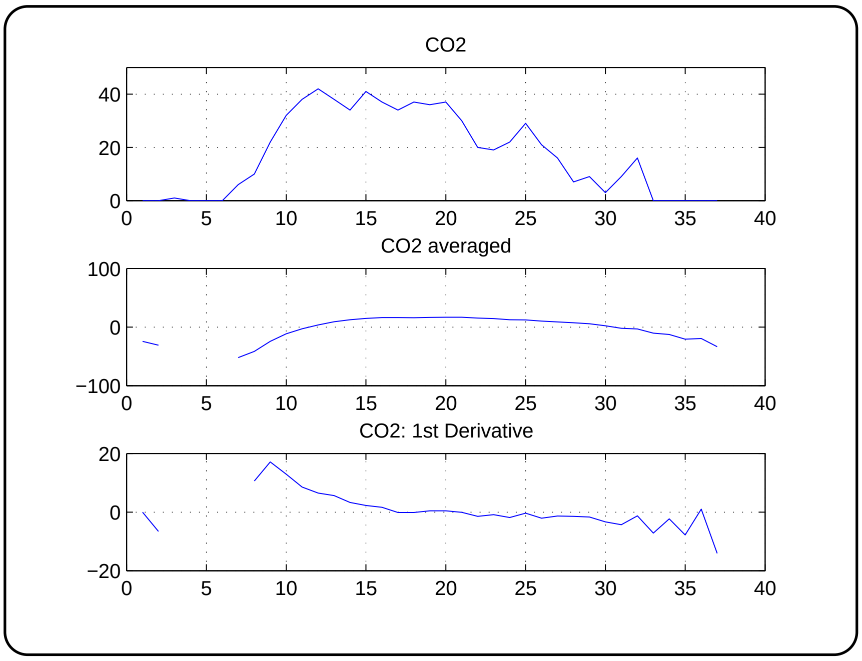

Figure 6.1: Trial 3 CO2 concentrations using logarithms.

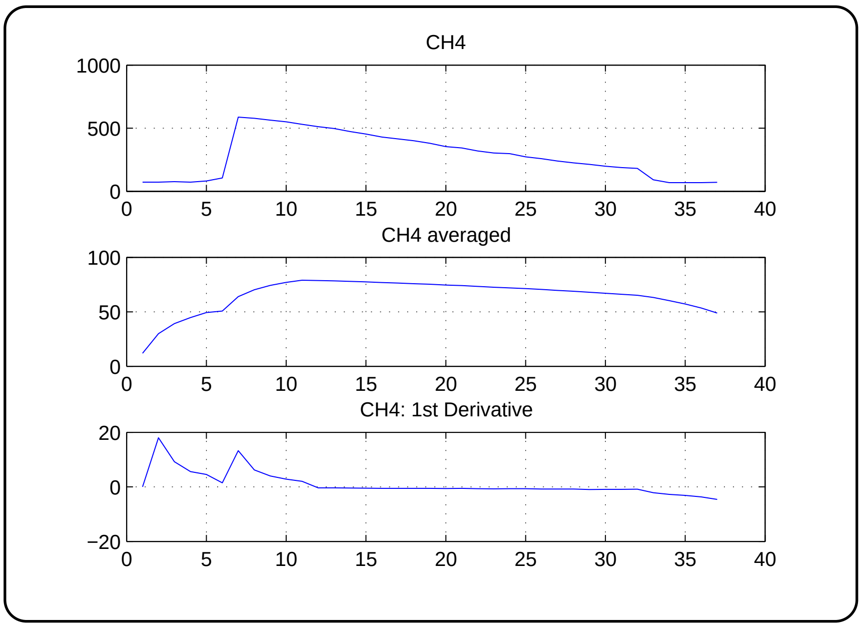

Figure 6.2: Trial 3 CH4 concentrations using logarithms.

One can see that gas concentrations appear to be close those that were expected from trial 3 in Chapter 5. Minimal further experimentation with the algorithm should provide acceptably accurate gas concentration data. Consideration must be made during such investigation, of the limitations of logarithms at zero, as can be seen in Figure

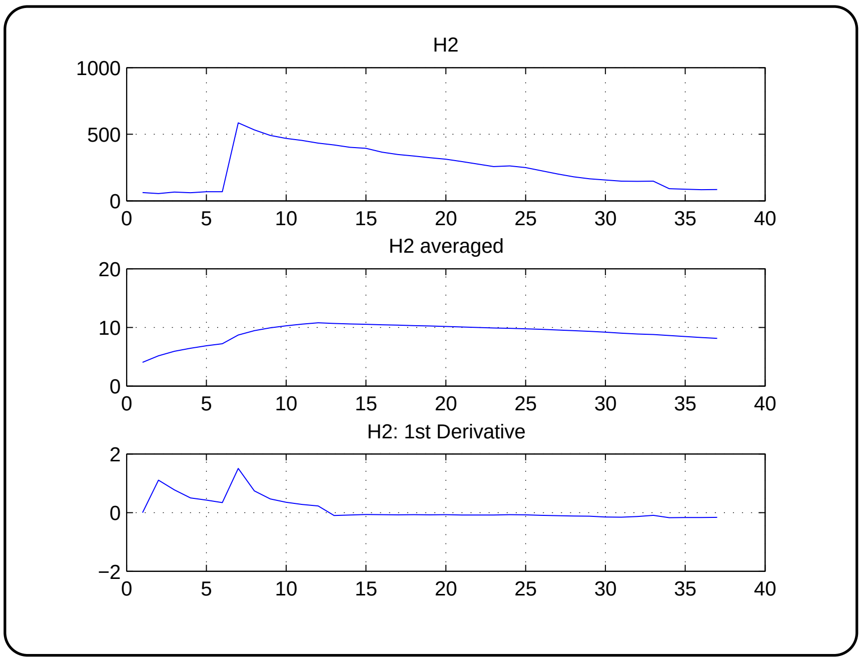

Figure 6.3: Trial 3 H2 concentrations using logarithms.

6.1. It is unknown how the micro-controller might respond to such data. Together with adjustment of the fuzzy geometry parameters as outlined in Chapter 4, it is expected that a robust controller is realisable, even with the sensors employed here.

6.1.1.2 Heuristic Technique

Due mainly to time constraints, considerable simplification of the data was made by taking the readings on face value, thereby ignoring its logarithmic nature. Instead, allowance of this simplification was made by adjusting fuzzy parameters to suit.

Relevant data suitable for the simulation of an overload was performed heuristically. Characteristic curves representing gas concentration levels were based on captured data exhibited in Chapter 5.

It was ascertained in Chapter 5 that the MQ7 H_2 sensor is minimally influenced by the presence of CO_2. There is little evidence to support that it is dramatically affected by CH_4 levels, and this seems further unlikely considering CH_4 sensitivity is included in Hanwei's datasheet and minimal. The effect of CH_4 on the MQ7 is therefore assumed negligible. H_2 increases perceived by the micro-controller's ADC should therefore be sufficiently accurate. Overload level was 701 from Figure 5.7.

The MG811 CO_2 has been proven sensitive to H_2, but not CH_4 according to the datasheet. It is therefore hypothesised that increases in H_2 will be perceived by the micro-controller as simultaneous increases in CO_2, while decreases in CH_4 should have little impact. In addition to the expected increase in CO_2 levels to 143 from Figure 5.2 in trail 1, CO_2 levels were inflated slightly by summing a multiple of $\frac{1}{14}$ of H_2 levels, to reflect the measured cross sensitivity. This scalar value was determined by dividing the CO_2 response by that of the H_2 from trial 2 in Chapter 5. These values were 38 in Figure 5.5 and 546 in Figure 5.7.

Considerable uncertainty surrounds the TGS2611 CH_4 sensor. Both Figaro's datasheet and experimentation show considerable sensitivity to H_2, while experimentation additionally shows slight sensitivity to CO_2. All literature reviewed indicates a reduction in CH_4 levels during overload, while H_2 and CO_2 levels increase. It is therefore uncertain exactly how CH_4 levels will be perceived by the micro-controller through its ADC.

In order to test the controller, 3 possible CH_4 response cases were considered. The first, and most optimistic is the possibility that the presence of CO_2 and H_2 reduce the normal attenuation of the signal from the TGS2611 due to the fall in CH_4 concentration. CH_4 readings were synthesised by the addition of scalar multiples of CO_2 and H_2 to the CH_4 readings to reflect these trends. Under the second possible scenario, CO_2 and H_2 concentrations completely mask the response of the TGS2611, producing a constant, unchanged signal throughout the overload. The third case indicates that the CO_2 and H_2 trigger responses in the TGS2611 stronger than that of the fall in CH_4 levels, resulting in a perceived increase in CH_4 levels. It is predicted that this is the condition most likely to provoke inappropriate temperature setpoints from the micro-controller.

It must be highlighted here that while this approach seems reasonable, data used does not produce trends which approximate the responses of the sensors. The linear characteristics of the collected data have not been completely ignored, however. Original synthesised data more closely resembled the data collected, but allowed little insight into the behaviour of the controller over a range of first derivatives. A plot is included in Figure 6.4. Therefore, while maintaining significant values based on the captured data, the data used here has been deliberately altered to permit such scrutiny. There is much extra work necessary to establish data that replicates with absolute certainty, the responses of the sensors to an actual overload.

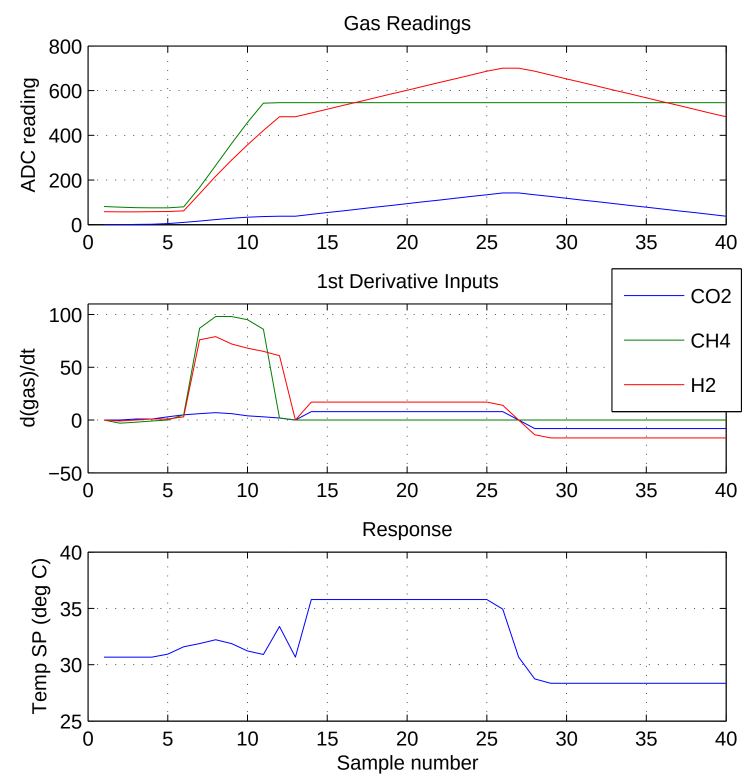

Figure 6.4: Resulting temperature setpoint using captured data.

6.2 Simulation Results

Trends and values for the signals received from the CO_2 and H_2 sensors are considered reasonably well established. However, three simulations were executed, one for each of the possible responses from the CH_4 sensor.

The fuzzy parameters were tuned individually to suit the ranges of the synthesised data and are tabulated below.

CO_2 input: CO_2 output:

```
CO2_LOW_membs  = [-16 -16  -2]; CO2_DN_TEMP_membs = [15 15 35];
CO2_NORM_membs = [-16   0  10]; CO2_AB_RITE_membs = [10 30 52];
CO_2HIGH_membs = [  2  10  16]; CO2_UP_TEMP_membs = [29 39 45];
```

CH_4 input: CH_4 output:

```
CH4_LOW_membs = [-50 -50 -10];  CH4_DN_TEMP_membs = [15 15 35];
CH4_NORM_membs = [-20 0 18];    CH4_AB_RITE_membs = [26 30 36];
CH4_HIGH_membs = [10 80 80];    CH4_UP_TEMP_membs = [29 39 45];
```

H_2 input: H_2 output:

```
H2_LOW_membs = [-60 -60 -5];    H2_DN_TEMP_membs = [15 15 35];
H2_NORM_membs = [-20 0 18];     H2_AB_RITE_membs = [10 30 52];
H2_HIGH_membs = [5 18 50];      H2_UP_TEMP_membs = [29 39 45];
```

Included are two plots for each of the simulations, one each for absolute inputs to the controller and for the first derivatives into the fuzzy functions. These are provided to help the reader visualise the gas levels during the overload.

In all figures, the first 12 data points reflect fluctuations while the controller is started and averaging buffers are yet to be filled. The second 14 data points approximate an overload, and the final 14 represent a return to normal after an overload.

6.2.0.3 Case 1: CH_4 decrease attenuated by CO_2 and H_2 sensitivity.

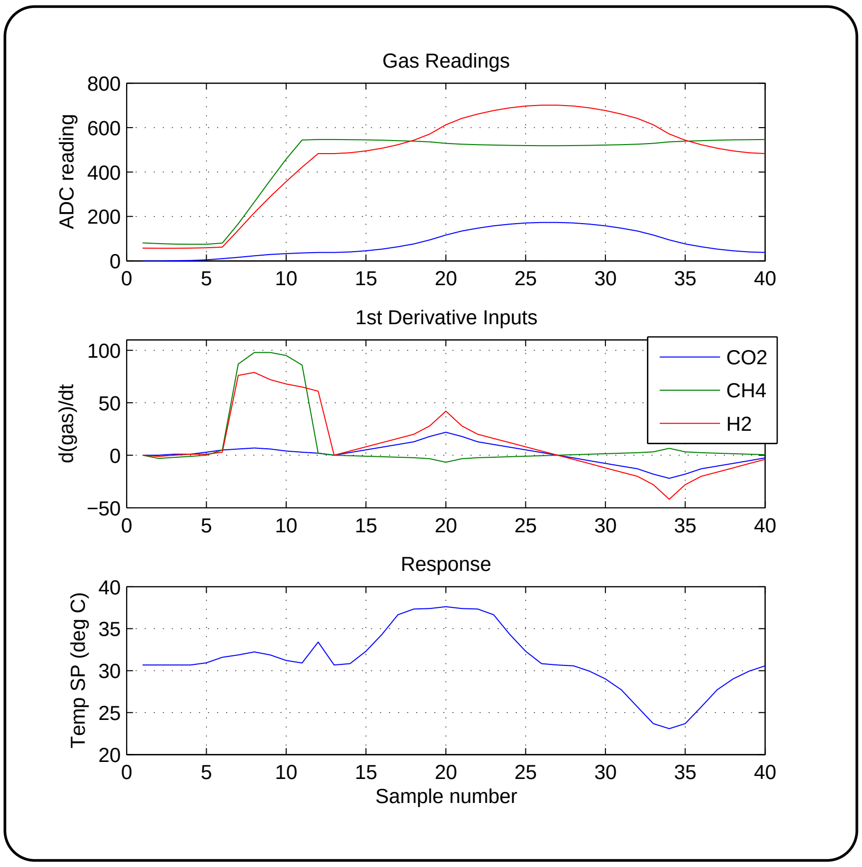

Figure 6.5: Gas inputs against temperature setpoint; case 1.

Figure 6.5 from case 1 shows the most optimistic scenario, in which the normal reduced signal from the TGS2611 CH_4 sensor is attenuated due to its cross-sensitivity with CO_2 and H_2. This situation shows the normal expected fluctuation in temperature setpoint. The controller appears to operate quite confidently under these circumstances.

6.2.0.4 Case 2: CH_4 decrease cancelled by CO_2 and H_2 sensitivity.

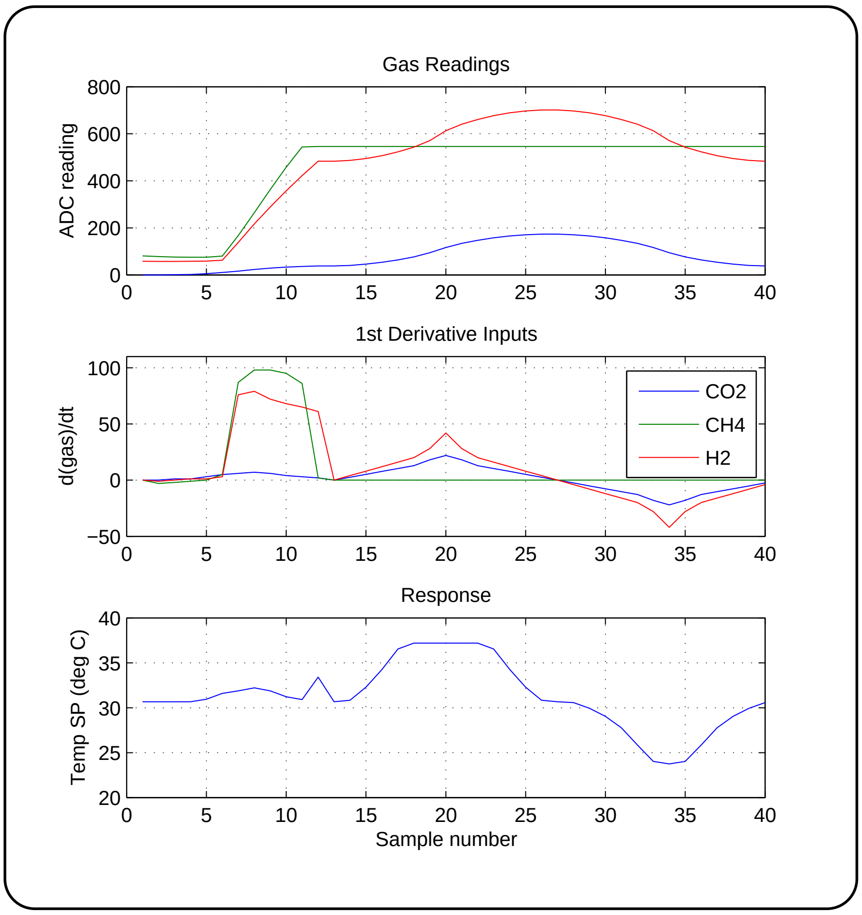

Figure 6.6: Gas inputs against temperature setpoint; case 2.

Figure 6.6 shows case 2 in which cross-sensitivity completely cancels the expected reduction in CH_4 levels. There is a reduction of only 1 degree C in the range of possible temperature setpoints.

6.2.0.5 Case 3: CH_4 decrease hidden by CO_2 and H_2 sensitivity.

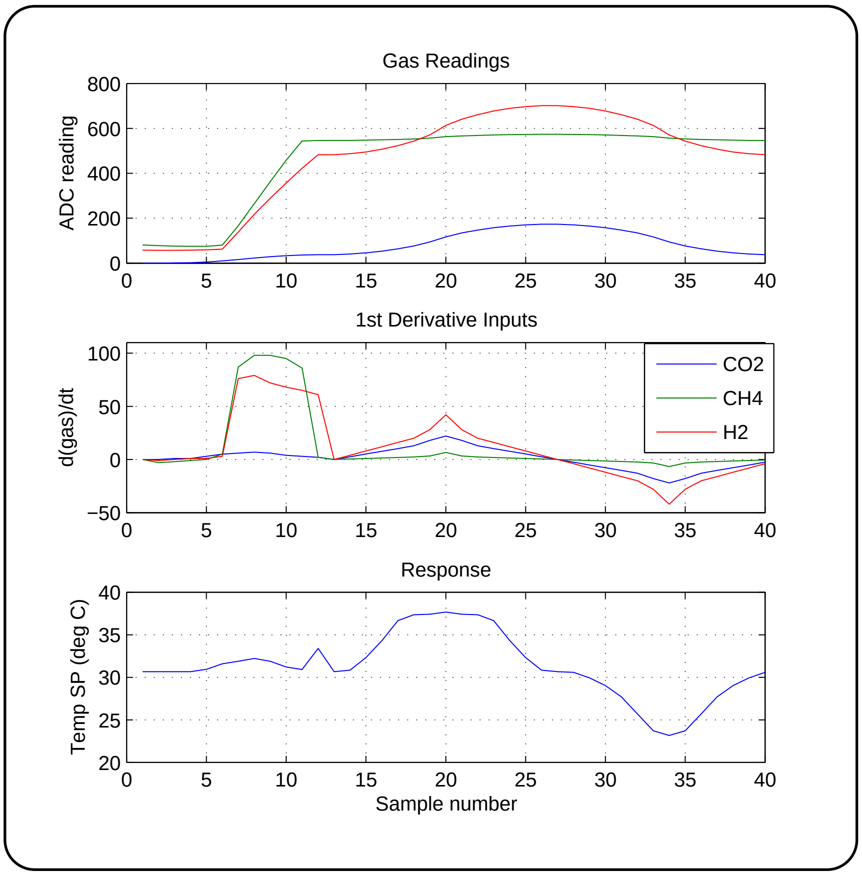

Figure 6.7: Gas inputs against temperature setpoint; case 3.

Figure 6.7 from case 3 shows the scenario in which increases in CO_2 and H_2 result in a perceived increase in CH_4 levels. This is estimated to be the worst case. The response closely resembles that of case 1.

It appears that the controller may be tuned to largely ignore an input of questionable accuracy. Further tuning may result in even further improvements, but it must be noted that the less the controller relies on this input, the less reason for having it connected at all. This would have been more the case had it been decided to use the weighting constant during defuzzification as covered in 4.3.2. It does appear that using fuzzy logic, an acceptable compromise may be reached. A large contributer to this result is the tweaking of the fuzzy parameters related to the CH_4 input to ignore CH_4 readings until the first derivative becomes excessive. For the record, the response of the controller to the fuzzy parameters dialled in is shown in Figure 6.8.

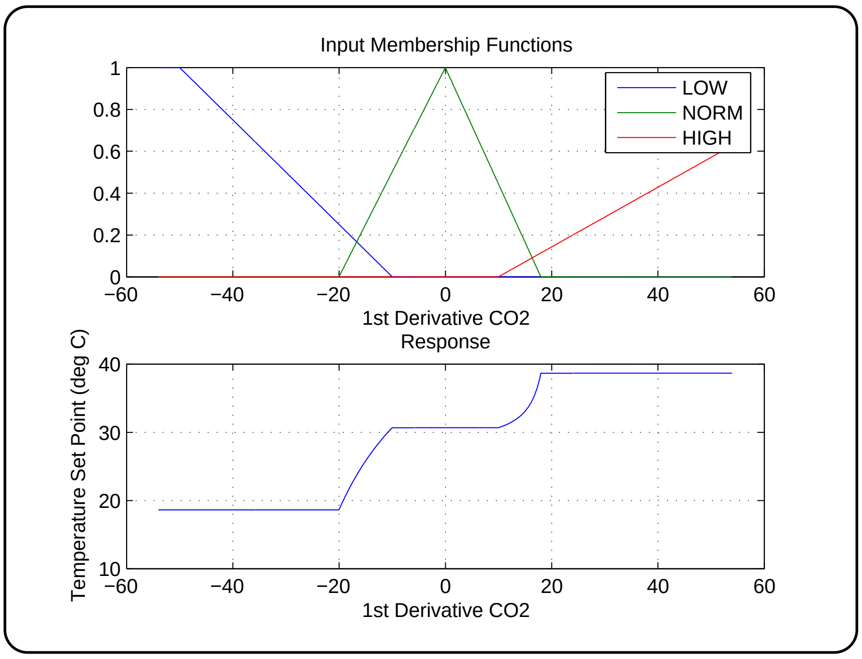

Figure 6.8: Controller response to synthesised d(CH_4) levels

.

6.3 Chapter Conclusion

This chapter has explained the process through which data collected from the micro-controller has been used to derive data that may be used to examine the responses of the controller. The data reflects likely inputs from the chosen gas sensors. While the data may not reflect exactly that which was obtained, using it with the MATLAB simulation scripts does show the potential of this system to be able to perform well under a variety of situations which may be encountered during a real overload.

Chapter 7

Conclusions and Further Work

7.1 Chapter Overview

This Chapter reviews and draws conclusions from the findings of this research project. It additionally looks at further work arising from the conclusions and makes recommendations as to the direction in which such work may prove beneficial.

It may be appreciated from Chapter 6, the developed embedded controller does indeed respond as intended, in that a temperature setpoint is determined from the gas sensor inputs using fuzzy logic algorithms. However, several complications mean that improved performance, in terms of robust control, should be possible with further efforts.

7.2 Conclusions

7.2.1 Biological Aspects

Literature review revealed that the biological aspects of anaerobic decomposition are well researched, and a genuine interest and commitment in this area would be required in order to make an original contribution in this field.

7.2.2 Electronic Aspects

The concept of using the signals from gas sensors as inputs to a fuzzy logic controller is also well established, but implementing such a system requires attention to several details.

The major hurdle in this project arose during analysis of sensor responses. There were three main aspects, each contributing to difficulties using these low cost devices. The first was cross-sensitivity, which resulted in unintentional sensor response to gases other than the target gas. The second was the logarithmic nature of the response of each of the three sensors, and the third was the absence of pure samples of each of the 3 target gases during the testing procedure.

The simplest method to reduce cross-sensitivity is the utilisation of better quality gas sensors with more informative documentation. Although the sensors employed in this project are able to perform tolerably, the selection of gas sensors for a particular application is no trivial task, and significant literature is available on the subject. Points worthy of consideration are target and background gas concentrations, sensor construction and characteristics, such as noise susceptibility, heater or other power requirements, output characteristics such as resistive, emf, logarithmic, linear, or necessary amplifier input impedance. The careful selection of sensors could greatly reduce the necessary signal processing.

Testing procedures, though considered adequate, needed to be more numerous and varied. Further trials using slightly different compositions were necessary in order to better determine the responses of the sensors.

7.2.3 Computational Aspects

It must be noted that development on embedded systems is not without unique challenges of its own. Graphical analysis on a PC based simulator is extremely advantageous, development from scratch being close to impossible on the target system alone. Memory constraints pose another limitation, this project approaching the limits of the

Atmega 8 micro-controller in this respect, even without the use of logarithmic signal processing. Further optimisation of the C code would definitely be possible, but significantly more ambitious projects would require the use of assembly language or a chip with a more generous memory.

Implementation of simple fuzzy logic algorithms to determine a single output setpoint proved to be a relatively straightforward and rewarding exercise. One interesting aspect was the possibility of the tuning of fuzzy parameters to suit uncertainty in the inputs. To be able to do so with multiple inputs is a great strength of fuzzy logic, and is a major reason for its use in so many consumer products and services.

Signal processing algorithms are another area in which considerable simplification was made. The employment of logarithmic signal data using linear methods would result in significant deviations from actual values. There were experiments conducted using logarithms with linear equations to extract individual target gas concentrations. These resulted in definite improvements in the available data, but were hampered by the absence of data from tests with pure CH_4 samples, thus demanding significant trial and error to arrive at acceptable parameters. Unfortunately time did not permit adequate experimentation, the success of which would additionally necessitate adjustment of fuzzy logic parameters, due to the difference in order of the obtained response data. However, extra time spent tuning these parameters is thought to be of major benefit, considering time invested and achievable results.

7.3 Further Work

This research project has encompassed a wide range of disciplines, from electronic and computing fields to biological processes which are to be controlled. While every attempt was made to cover the relevant details from each area in the available time, it is accepted that assumptions and simplifications have been made in the interest of practicality. Suggested further research into specific areas is recommended below.

Several parts of the programs need to be investigated, finished and thoroughly tested. This includes the logarithmic concentration extraction algorithm, fuzzy logic using all

3 inputs, and the semaphore lights. These were left due to the time constraints on this project. A basic working controller has been simulated, but not implemented.

Thorough investigation into gas sensors and their peculiarities are mentioned in the conclusion. The selection of those with more suitable responses as opposed to those of suitable cost, is expected to reduce the necessity of sophisticated signal processing techniques. Better laboratory testing conditions using pure, or at least known target gases and mixtures to investigate their responses during overload conditions would be of great additional assistance.

As mentioned in Chapter 3, E-noses in general have been a recent buzz topic due to their possible application to numerous fields such as process control, air quality monitoring, security and drug detection. Cross-sensitivity and noise in e-noses is therefore not a new area of concern, and again, significant literature is available. A recent area of interest is the use of artificial neural networks (ANN's) to determine accurate gas concentrations from confused signals. Interested readers are referred to the following works listed in the bibliography:

- Electronic noses - a mini review: (Strike, Meijerink & Koudelka-Hep 1999)
- Data analysis for electronic nose systems: (Simon, James & Zulfiquer 2006)
- An artificial olfactory system based on gas sensor array and back propagation neural network: (Huiling, Guangzhong & Yadong n.d.)
- Monitoring growth of the methanogenic archaea methanobacterium formicicum using an electronic nose: (Brandgard, Sundh, Nordburg, Schnurer, Mandenius & Mathisen 2001)

The final test for this controller would be to place it on a working digester and tune the fuzzy parameters to suit. It may also be found that the monitoring of other parameters, such as absolute gas concentrations and second derivatives may be necessary to achieve robust control. Analysis of the response of a digester to fluctuations in temperature and its use as a valid method of control should also be assessed.

7.4 Final Conclusion

It was established through literature review that the largest problem with anaerobic digesters is that with overloading of the digester. It was found that a series of microbial processes produce an increase in CO_2 and H_2, and a reduction in CH_4 concentrations during an overload. These gases and their derivatives with respect to time indicate an impending overload. Using electronic gas sensors, these variables may be monitored to determine an ideal temperature setpoint in an attempt to increase the activity of methanogenic bacteria, thus reducing the possibility of an overload.

In order to implement a possible controller, an embedded system was designed and developed to read gas concentrations from electronic hardware. Fuzzy logic MATLAB scripts were developed, then ported to C for use on the embedded platform. The response of the embedded system was compared to the MATLAB simulator to ascertain the relevance of the simulator. The system was then tested in a laboratory with target gas samples, and the data used to tune the fuzzy parameters, and to derive estimates of overload conditions as perceived by the microcontroller. Data reflecting an overload was then put into MATLAB simulation scripts and then critically analysed. The results were promising, but further research is necessary to ensure that this controller is able to adequately prevent a digester overload.

It is hoped that the results of this project are used to make anaerobic digesters a more competitive, reliable, and safe option to produce renewable fuel gas for households and small communities. The greatest success is desired for any further progress towards this end, and persuits in ethical energy solutions for the future in general.

References

Brandgard, J., Sundh, I., Nordburg, A., Schnurer, A., Mandenius, C. F. & Mathisen, B. (2001), ‘Monitoring growth of the methanogenic archaea methanobacterium formicicum using an electronic nose’, *Biotechnology Letters* **23**, 241–248.

Fry, J. L. (1973), ‘Methane digesters for fuel and fertiliser’.

Hobson, P. N., Bousfield, S. & Summers, R. (1981), *Methane production from agricultural and domestic wastes*, Applied Science Publishers LTD.

Huiling, T., Guangzhong, X. & Yadong, J. (n.d.), ‘An artificial olfactory system based on gas sensor array and back propagation neural network’.

Muller, A., Marsili-Libelli, S., Aivasidis, A., Lloyd, T., Kroner, S. & Wandrey, C. (1997), ‘Fuzzy logic control of disturbances in a wastewater treatment process’.

Murnleitner, E., Becker, T. M. & Delgado, A. (2002), ‘State detection and control of overloads in the anaerobic wastewater treatment using fuzzy logic’, *Water Reasearch* **36**.

Neswold, R. (2006), *Avrlibc Reference Manual 1.4.5.*
`http://savannah.nongnu.org/projects/avr-libc/`
current February 2007.

Reznik, L. (1997), *Fuzzy Controllers*, Newnes.

Roarty, M. (1998), *Natural Gas: Energy for the New Millenium.*
`http://www.aph.gov.au/library/pubs/rp/1998-99/99rp05.htm`
current November 2007.

Simon, S. M., James, D. & Zulfiquer, A. (2006), 'Data analysis for electronic nose systems'.

Smith, P., Bordeux, F., Goto, M., Shiralipour, A., Wilkie, A., Andrews, J., Ide, S. & Barnett, M. (1988), *Biological production of methane from biomass*, Elsevier Science Publishers LTD, p. 291.

Sowell, T. (2005), *Fuzzy Logic for Just Plain Folks*.
`http://www.fuzzy-logic.com`
current August 2005.

Stafford, D., Hawkes, D. & Horton, R. (1981), *Methane Production from Organic Wastes*, CRC Press Inc.

Strike, D. J., Meijerink, M. G. H. & Koudelka-Hep, M. (1999), 'Electronic noses - a mini review'.

Appendix A

Project Specification

University of Southern Queensland

Faculty of Engineering and Surveying

ENG 4111/2 Research Project

Project Specification

FOR:	Terence Michael Sullavan
TOPIC:	Embedded Control System for Biogas Digester
SUPERVISORS:	Selvan Pather
	Mark Phythian
PROJECT AIM:	This project aims to investigate, design and develop an embedded control system to control the temperature of, and to improve the efficiency, reliablity and safety of domestic scale biogas digesters.
PROGRAMME:	**Issue C, 27th April, 2007**

1. Research anaerobic digestion requirements.
2. Select appropriate microcontroller chip and establish development environment.
3. Research and develop electronic circuits and microcontroller program subroutines.
4. Critically evaluate controller performance in laboratory.
5. Tune program parameters to improve performance.

AGREED:

________________ (student), ________________ , ________________ (supervisors)

(dated) _____ / _____/ _____

Appendix B

Source Code Listings

A large part of this research project has been developing MATLAB source code to assist with the development of C code, and to provide visual assistance with, and insight into the behaviour of the controller. These scripts were used to plot many of the figures throughout this dissertation.

There were two main types of script developed, one which simply reads data files captured by Minicom from the micro-controller and plots it, and the second which uses the same fuzzy algorithms as the controller. The latter is able to assign temperature setpoints to the data provided.

The second section contains C source code suitable for use with recent versions of avr-gcc, version 4.1.2 used in this project. The fuzzy C code development followed a similar evolution to the MATLAB scripts, but with hardware subroutines, based heavily on those from the avrlibc examples, developed first. All have been included.

B.1 MATLAB Functions

B.1.1 The `minicomPlot.m` Script

The MATLAB script file `minicomPlot.m` is a simple read and plot type script. It is able to calculate averages and CH_4 and H_2 derivatives, necessitated by bugs in the averaging buffers which arose when the controller was activated without having its memory erased. These mimic the behaviour of the micro-controller on startup and were uncommented when appropriate. It also includes 2 attempts at logarithmic gas concentration extraction, the first used to produce the plots in section 6.1.1.1. This script is shown in Listing B.1.

Listing B.1: minicomplot1m.

```
%Just reads the file 'Trial_n.cap' in current dir and plots it out.
clear; clc;
[ temp, temp_avg, CO2, CO2_avg, d_CO2, CH4, CH4_avg, d_CH4, H2,
 →H2_avg, d_H2] = ...
   textread('Trial_4.cap', 'Temp: %s Temp_avg: %s CO2: %s CO2_avg: %
    →s d_CO2: %s CH4: %s CH4_avg: %s d_CH4: %s H2: %s H2_avg: %s
    →d_H2: %s');
%Trial 1: 10 000 ppm CO2
%Trial 2 : 1000 ppm H2; gain = 3
%Trial 3 : 1000 ppm H2; gain = 2
%Trial 3_1: 1000ppm H2; gain = 1
%Trial 4: 10 000 ppm town gas
CH4 = hex2dec(CH4);
CH4_avg = hex2dec(CH4_avg);
d_CH4 = hex2dec(d_CH4);
CO2 = hex2dec(CO2);
CO2_avg = hex2dec(CO2_avg);
d_CO2 = hex2dec(d_CO2);
H2 = hex2dec(H2);
H2_avg = hex2dec(H2_avg);
d_H2 = hex2dec(d_H2);
for i = 1:length(CO2)
     if CH4(i) > 32667
         CH4(i) = CH4(i) - 65536;
     end
     if CH4_avg(i) > 32667
         CH4_avg(i) = CH4_avg(i) - 65536;
     end
     if d_CH4(i) > 32667
         d_CH4(i) = d_CH4(i) - 65536;
     end
     if CO2(i) > 32667
         CO2(i) = CO2(i) - 65536;
     end
     if CO2_avg(i) > 32667
         CO2_avg(i) = CO2_avg(i) - 65536;
     end
     if d_CO2(i) > 32667
         d_CO2(i) = d_CO2(i) - 65536;
     end
     if H2(i) > 32667
         H2(i) = H2(i) - 65536;
     end
     if H2_avg(i) > 32667
         H2_avg(i) = H2_avg(i) - 65536;
     end
     if d_H2(i) > 32667
         d_H2(i) = d_H2(i) - 65536;
     end
end

%Work around for dodgy buffers in Trial_3_1
%Read raw sensor data and calculate averages and derivatives in
 →MATLAB
%averages:
for n = 1:4
         CH4_avg(n) = (sum(CH4(1:n))/5);
end
for n=5:length(CH4)
      CH4_avg(n) = ((sum(CH4((n-4):n))) / 5);
end
for n = 1:5
     H2_avg(n) = sum(H2(1:n))/6;
end
for n=6:length(H2)
     H2_avg(n) = sum(H2((n-5):n));
     H2_avg(n) = H2_avg(n) / 6;
end

%Logarithmic compensation for cross sensitivity 1
%for n = 1:length(CH4_avg)
%     H2_avg(n) = 4*(log10((H2_avg(n))))% - log10(CO2_avg(n))/20;
%   CO2_avg(n) = 50*log10(CO2_avg(n)) - 6*H2_avg(n);
%    CH4_avg(n) = 180*log10(log10(CH4_avg(n)))% - CO2_avg(n)/20 -
 →45.5 ;
%end

%Logarithmic compensation for cross sensitivity 2
%for n = 1:length(CH4_avg)
%     H2_avg(n) = 4*log10((log10((H2(n))))) - 1;
%   CO2_avg(n) = 50*log10(CO2(n)) - H2_avg(n);
%    CH4_avg(n) = 180*log10(log10(CH4(n))) - CO2(n)/50 - 15*H2_avg(n)
 →  -48;
%end

d_CO2(1)=0;
d_CH4(1)=0;
d_H2(1)=0;
%derivatives:
for n=1:(length(CH4_avg)-1);
     d_CO2(n+1) = (CO2_avg(n+1) - CO2_avg(n));
      d_CH4(n+1) = (CH4_avg(n+1) - CH4_avg(n));
     d_H2(n+1) = (H2_avg(n+1) - H2_avg(n));
end

%Plot averaged reading
```

```
%Plot raw and averaged readings with derivatives on 3 separate
→figures; 1
%for each sensor.
subplot(3, 1, 1), plot(1:i, CO2);
title('CO2');
grid on;

subplot(3, 1, 2), plot(1:i, CO2_avg);
title('CO2 averaged');
grid on;
%axis(subplot(312), [0 i 0 250]);

subplot(3, 1, 3), plot(1:i, d_CO2);
title('CO2: 1st Derivative');
grid on;
%axis(subplot(313), [0 i -15 25]);

figure;

subplot(3, 1, 1), plot(1:i, CH4);
title('CH4');
grid on;

subplot(3, 1, 2), plot(1:i, CH4_avg);
title('CH4 averaged');
grid on;
%axis(subplot(312), [0 i 0 250]);

subplot(3, 1, 3), plot(1:i, d_CH4);
title('CH4: 1st Derivative');
grid on;
%axis(subplot(313), [0 i -15 25]);

figure;

subplot(3, 1, 1), plot(1:i, H2);
title('H2');
grid on;

subplot(3, 1, 2), plot(1:i, H2_avg);
title('H2 averaged');
grid on;
%axis(subplot(312), [0 i 0 250]);

subplot(3, 1, 3), plot(1:i, d_H2);
title('H2: 1st Derivative');
grid on;
%axis(subplot(313), [0 i -15 25]);
```

B.1.2 The Simulator Scripts

The MATLAB based simulator was created in stages.

B.1.2.1 Simulator Stage 1

The first stage were 3 files consisting of a main file and fuzzify and de-fuzzify function files. These scripts plotted input and output membership functions, with the input (d(CO2)/dt) and output response (temperature setpoint) values shown as small circles. The main script, `FL_CO2_temp.m` is able to produce several different plots, the relevant lines commented or uncommented as necessary. These scripts are shown in Listings B.2 to B.4. The main function calls the other two. Scripts are listed in the order in which they are called.

Listing B.2: FLCO2temp.m

```
% MAIN function
%A first attmept at producing a fuzzy logic simulation using d_CO2
 →in and
% temperature out.
% The lower and upper membership and response functions must be
 →trapezoidal
% and the centre membership and response functions must be
 →triangular.
%
% mem1  mem2
%    ******     *     *******
%          *   * *   *
%           *     * *
%            *     *
%             *   * *
%          *   *     *
%    *********  *************
%               mem3
clear; clc;
%Globalvars for membership parameters
global LOW_membs NORM_membs HIGH_membs;
%Responses
global DN_TEMP_membs AB_RITE_membs UP_TEMP_membs

%fuzzylims = [ -2 2];

%Define membership functions

%These values seem about right for the 1st derivative of C02
%Input
LOW_membs = [-16 -16 -2];
NORM_membs = [-16 0 10];
HIGH_membs = [2 10 16];

%Output
DN_TEMP_membs = [15 15 35];
AB_RITE_membs = [25 30 37];
```

```
UP_TEMP_membs = [29 39 45];

%These values seem about right for the 1st derivative of C02
%Input
%LOW_membs = [-16 -16 -2];
%NORM_membs = [-16 0 10];
%HIGH_membs = [2 10 45];

%Output
%DN_TEMP_membs = [15 15 35];
%AB_RITE_membs = [10 30 52];
%UP_TEMP_membs = [29 39 45];

%d_CO2 reading from sensor read function
d_CO2 = [LOW_membs(1)-5: .1: HIGH_membs(3) ]; %CO2 array used to
 →plot

d_CO2(length(d_CO2)+1) = 16;    % *** Append user defined reading of
 →interest here ***

temp = [DN_TEMP_membs(1): .1: UP_TEMP_membs(3)];     %temperature
 →array used to plot

%d_CO2 reading from sensor read function
%d_CO2 = [LOW_membs(1) - 5: .1: HIGH_membs(3) - 29]; %CO2 array used
 → to plot

%d_CO2(length(d_CO2)+1) = 20;    % *** Append user defined reading of
 → interest here ***

%temp = [DN_TEMP_membs(1): .1: UP_TEMP_membs(3)];     %temperature
 →array used to plot

%*************End of user defined variables***************

%Obtain membership to fuzzy rules
for i=1:length(d_CO2)
    [LOW(i), NORM(i), HIGH(i)] = fuzzify(d_CO2(i));
end

figure;
%plot membership functions
subplot(2,1,1), plot(d_CO2(1:length(d_CO2)-1), LOW(1:length(d_CO2)
 →-1),...
    d_CO2(1:length(d_CO2)-1), NORM(1:length(d_CO2)-1),...
    d_CO2(1:length(d_CO2)-1), HIGH(1:length(d_CO2)-1)...
    ); grid on; hold on;

%Plot circles around membership functions
subplot(2,1,1), plot(d_CO2(length(d_CO2)), LOW(length(d_CO2)), 'o'
 → ,...
    d_CO2(length(d_CO2)), NORM(length(d_CO2)), 'o',...
    d_CO2(length(d_CO2)), HIGH(length(d_CO2)), 'o');

title('Input Membership Functions');
%legend('LOW','NORM','HIGH','d GAS/ dt')

%Required response;
%if CO2 LOW, down temp,
%if CO2 NORM, temp ~=temp,
%if CO2 HIGH, up temp.

%Obtain crisp temperature setpoint (response)
temp_SP = Dfuzzify(LOW(length(LOW)), NORM(length(NORM)), HIGH(length
 →(HIGH)));

%Obtain crisp temperature setpoint array(response)
%temp_SP=Dfuzzify(LOW, NORM, HIGH);

%Do a plot of response
for i=1:length(temp)
```

```
    %Membership rules DN_TEMP
    if (temp(i) < DN_TEMP_membs(2))
        DN_TEMP(i) = LOW(length(LOW));
    end
    if (temp(i) >= DN_TEMP_membs(2) & temp(i) < DN_TEMP_membs(3))
        DN_TEMP(i) = LOW(length(LOW)) + (DN_TEMP_membs(2) - temp(i))
         → * LOW(length(LOW))/(DN_TEMP_membs(3) - DN_TEMP_membs(2));
    end
    if (temp(i) >= DN_TEMP_membs(3))
        DN_TEMP(i) = 0;
    end
    %Membership rules AB_RITE
    if (temp(i) < AB_RITE_membs(1) | temp(i) >= AB_RITE_membs(3))
        AB_RITE(i) = 0;
    end
    if (temp(i) >= AB_RITE_membs(1) & temp(i) < AB_RITE_membs(2))
        AB_RITE(i) = (temp(i) - AB_RITE_membs(1)) * NORM(length(NORM
         →)) / (AB_RITE_membs(2) - AB_RITE_membs(1));
    end
    if (temp(i) >= AB_RITE_membs(2) & temp(i) < AB_RITE_membs(3))
        AB_RITE(i) = NORM(length(NORM)) + (AB_RITE_membs(2) - temp(
         →i)) * NORM(length(NORM)) / (AB_RITE_membs(3) -
         →AB_RITE_membs(2));
    end
    %Membership rules UP_TEMP
    if (temp(i) < UP_TEMP_membs(1))
        UP_TEMP(i) = 0;
    end

    if (temp(i) >= UP_TEMP_membs(1)) & (temp(i) < UP_TEMP_membs(2))
        UP_TEMP(i) = (temp(i) - UP_TEMP_membs(1)) * HIGH(length(HIGH
         →)) / (UP_TEMP_membs(2) - UP_TEMP_membs(1));
    end
    if temp(i) >= UP_TEMP_membs(2)
        UP_TEMP(i) = HIGH(length(HIGH));
    end
end

%plot response functions
subplot(2,1,2), plot(temp, DN_TEMP, temp, AB_RITE, temp, UP_TEMP);
 →grid on; hold on;

%figure;
%Plot input (d_CO2)
%subplot(2,1,1), plot(1:length(d_CO2), d_CO2, 1:length(d_CO2),
 →fuzzylims(1), 'r', 1:length(d_CO2), fuzzylims(2), 'r');
%grid on; hold on;
%title('Input d(CO2)/dt');
%ylabel('d(CO2)/dt');

%Plot setpoint on x axis (temp_SP array)
subplot(2,1,2), plot(temp_SP,0, 'o');grid on; hold on;
axis(subplot(212), [DN_TEMP_membs(1) UP_TEMP_membs(3) 0 1]);

title('Response');
%xlabel('1st Derivative CO2');
xlabel('Temperature Set Point (deg C)');

%EOF
```

Listing B.3: fuzzifyV1.m

```
function [LOW, NORM, HIGH] = fuzzify(d_CO2)
% A function to fuzzify.
% The lower and upper membership and response functions must be
 →trapezoidal
% and the centre membership and response functions must be
 →triangular for
% the membership calculations.
global LOW_membs NORM_membs HIGH_membs;
%Membership rules LOW
      if (d_CO2 < LOW_membs(2))
           LOW = 1;
      end
      if (d_CO2 >= LOW_membs(2)) & (d_CO2 < LOW_membs(3))
           LOW = 1 - (d_CO2 -  LOW_membs(2))/(LOW_membs(3)-LOW_membs(2)
             →);
      end
      if (d_CO2 >= LOW_membs(3))
           LOW = 0;
      end
      %Membership rules NORM
      if (d_CO2 < NORM_membs(1) | d_CO2 >= NORM_membs(3))
           NORM = 0;
      end
      if (d_CO2 >= NORM_membs(1) & d_CO2 < NORM_membs(2))
           NORM = (d_CO2 - NORM_membs(1)) / (NORM_membs(2) - NORM_membs
             →(1));
      end
      if (d_CO2 >=  NORM_membs(2) & d_CO2 < NORM_membs(3))
           NORM = 1 - (d_CO2 - NORM_membs(2)) / (NORM_membs(3) -
             →NORM_membs(2));
      end
      %Membership rules HIGH
      if (d_CO2 < HIGH_membs(1))
           HIGH = 0;
      end
      if (d_CO2 >= HIGH_membs(1) & d_CO2 < HIGH_membs(2) )
           HIGH = (d_CO2 - HIGH_membs(1)) / (HIGH_membs(2) - HIGH_membs
             →(1));
      end
      if d_CO2 >= HIGH_membs(2)
           HIGH = 1;
      end
%EOF
```

Listing B.4: DfuzzifyV1.m

```
function temp_SP = Dfuzzify(LOW, NORM, HIGH)
% A function to de-fuzzify.
% The lower and upper membership and response functions are
→trapezoidal,
% and the centre membership and response functions triangular for
% the area and centroidal calculations.
global DN_TEMP_membs AB_RITE_membs UP_TEMP_membs
%Calculate areas
Area_DN_TEMP = LOW(length(LOW)) * ((DN_TEMP_membs(2) - DN_TEMP_membs
→(1)) + (DN_TEMP_membs(3) - DN_TEMP_membs(2))/2 )
Area_AB_RITE = NORM(length(NORM)) * (AB_RITE_membs(3) -
→AB_RITE_membs(1))/2
Area_UP_TEMP = HIGH(length(HIGH)) * ((UP_TEMP_membs(3) -
→UP_TEMP_membs(2)) + (UP_TEMP_membs(2) - UP_TEMP_membs(1))/2 )

%Work out response areas horizontal centroids
%z_DN_TEMP = ((DN_TEMP_membs(2)-DN_TEMP_membs(1))^2 +...
%     ((DN_TEMP_membs(3)-DN_TEMP_membs(2))^2)/3 +...
%     (DN_TEMP_membs(2)-DN_TEMP_membs(1)) * (DN_TEMP_membs(3)-
→DN_TEMP_membs(2)))  /...
%     (2*(DN_TEMP_membs(2)-DN_TEMP_membs(1)) + (DN_TEMP_membs(3) -
→DN_TEMP_membs(2)))

z_DN_TEMP = ((DN_TEMP_membs(2)-DN_TEMP_membs(1))^2 + 2*DN_TEMP_membs
→(1)*(DN_TEMP_membs(2)-DN_TEMP_membs(1)) + ...
     ((DN_TEMP_membs(3)-DN_TEMP_membs(2))^2)/3 +...
     DN_TEMP_membs(2) * (DN_TEMP_membs(3)-DN_TEMP_membs(2)))  /...
     (2*(DN_TEMP_membs(2)-DN_TEMP_membs(1)) + (DN_TEMP_membs(3) -
     →DN_TEMP_membs(2)))

z_AB_RITE = (2*((AB_RITE_membs(2)-AB_RITE_membs(1))^2)/3 +
→AB_RITE_membs(1) * (AB_RITE_membs(2)-AB_RITE_membs(1)) + ...
     ((AB_RITE_membs(3)-AB_RITE_membs(2))^2)/3 + AB_RITE_membs(2)*(
     →AB_RITE_membs(3)-AB_RITE_membs(2))) / ...
     (AB_RITE_membs(2)-AB_RITE_membs(1) + AB_RITE_membs(3)-
     →AB_RITE_membs(2))

z_UP_TEMP = (4*((UP_TEMP_membs(2)-UP_TEMP_membs(1))^2)/3 + 2*
→UP_TEMP_membs(1)*(UP_TEMP_membs(2)-UP_TEMP_membs(1)) +...
     3*((UP_TEMP_membs(3)- UP_TEMP_membs(2))^2)/2 + 3*UP_TEMP_membs
     →(2)*(UP_TEMP_membs(3)-UP_TEMP_membs(2))) /...
     (2*(UP_TEMP_membs(2)-UP_TEMP_membs(1)) + 3*(UP_TEMP_membs(3)-
     →UP_TEMP_membs(2)))

%final temperature setpoint is centroid of entire response area
temp_SP = (Area_DN_TEMP*z_DN_TEMP + Area_AB_RITE*z_AB_RITE +
→Area_UP_TEMP*z_UP_TEMP) ...
     / (Area_DN_TEMP + Area_AB_RITE + Area_UP_TEMP)
```

B.1.2.2 Simulator Stage 2

The second stage involved re-writing the scripts to handle 3 different inputs, one for each gas, plus membership function geometry parameters as necessary. The second stage contains the data used to simulate an overload. Included are two sets of data; one mimicking the captured data as closely as possible, and the second with a range of derivatives used to explore the performance of the controller. These scripts are shown in Listings B.5 to B.7. Again, the functions are listed in the order in which they are called.

Listing B.5: FL3SensTemp.m

```
% MAIN function
%A first attmept at producing a fuzzy logic simulation using d_CO2
→in and
% temperature out.
% The lower and upper membership and response functions must be
→trapezoidal
% and the centre membership and response functions must be
→triangular.
%
% mem1   mem2
%    ******     *     *******
%          *   * *   *
%           *     * *
%            *     *
%             *   * *
%          *   *     *
%    *********  *************
%               mem3
clear; clc;

fuzzylims = [ -2 2];

%Define membership functions

%These values seem about right for the 1st derivative of C02
%+d(CO2)/dt require increases in temp
%Input
CO2_LOW_membs = [-16 -16 -2];
CO2_NORM_membs = [-16 0 10];
CO2_HIGH_membs = [2 10 16];

%Output
CO2_DN_TEMP_membs = [15 15 35];
CO2_AB_RITE_membs = [10 30 52];
CO2_UP_TEMP_membs = [29 39 45];

%These values seem about right for the 1st derivative of CH4
%-d(CH4)/dt require increases in temp
%Input
CH4_LOW_membs = [-50 -50 -10];
CH4_NORM_membs = [-20 0 18];
CH4_HIGH_membs = [10 80 80];

%Output
CH4_DN_TEMP_membs = [15 15 35];
```

```
CH4_AB_RITE_membs = [26 30 36];
CH4_UP_TEMP_membs = [29 39 45];

%These values seem about right for the 1st derivative of H2
%+D(H2)/dt require increases in temp
%Input
H2_LOW_membs = [-60 -60 -5];
H2_NORM_membs = [-20 0 18];
H2_HIGH_membs = [5 18 50];

%Output
H2_DN_TEMP_membs = [15 15 35];
H2_AB_RITE_membs = [10 30 52];
H2_UP_TEMP_membs = [29 39 45];


%Readings from micro sensor read function. First 12 represent
 →startup. Next
%14 represent o/load. Final 14 represent opposite condition.
%CO2_avg=[0  0  1  2  5  10 16  23  29  33  36  38   38  46  54  62
 →  70  78  86  94  102 110 118 126 134 142   142 134 126 118 110
 →102  94  86  78  70  62  54  46  38];
%CH4_avg=[81 78 76 75 75 80 167 265 363 458 544 546   546 546 546
 →546 546 546 546 546 546 546 546 546 546 546   546 546 546 546 546
 →546 546 546 546 546 546 546 546 546];
%H2_avg= [58 57 57 58 59 62 138 217 289 357 422 483   483 500 517
 →534 551 568 585 602 619 636 653 670 687 701   701 687 670 653 636
 →619 602 585 568 551 534 517 500 483];

CO2_avg=[0  0  1  2  5  10 16  23  29  33  36  38   38  40  44  50
 → 58  68  82   98 112 122 130 136 140 142   142 140 136 130 122 112
 →  98  82  68  58  50  44  40  38];
CH4_avg=[81 78 76 75 75 80 167 265 363 458 544 546   546 546 546 546
 → 546 546 546 546 546 546 546 546 546 546   546 546 546 546 546 546
 → 546 546 546 546 546 546 546 546];
H2_avg= [58 57 57 58 59 62 138 217 289 357 422 483   483 487 495 507
 → 523 543 571 613 641 661 677 689 697 701   701 697 689 677 661 641
 → 613 571 543 523 507 495 487 483];

%Modify readings to reflect cross-sensitivity
for n=13:length(CO2_avg)
    CO2_avg(n)=CO2_avg(n) + (H2_avg(n)-483)/7;   %remove constant
     →offset and add sensitivity to H2
    CH4_avg(n)=CH4_avg(n) - ((CO2_avg(n)-38)-(H2_avg(n)-483))/3;
    %CH4_avg(n)=0;
end

d_CO2(1)=0;
d_CH4(1)=0;
d_H2(1)=0;

%derivatives:
for n=1:(length(CO2_avg)-1);
    d_CO2(n+1) = (CO2_avg(n+1) - CO2_avg(n));
     d_CH4(n+1) = (CH4_avg(n+1) - CH4_avg(n));
    d_H2(n+1) = (H2_avg(n+1) - H2_avg(n));
end
%d_H2=[-150:150];  %Just for plotting tempsp vs d_H2
%d_CH4=[-80:120];

%************End of user defined variables**************

%Obtain membership to fuzzy rules
for i=1:length(d_CH4)
    [CO2_LOW(i), CO2_NORM(i), CO2_HIGH(i)] = fuzzify(d_CO2(i),
     →CO2_LOW_membs, CO2_NORM_membs, CO2_HIGH_membs);
    [CH4_LOW(i), CH4_NORM(i), CH4_HIGH(i)] = fuzzify(d_CH4(i),
    →CH4_LOW_membs, CH4_NORM_membs, CH4_HIGH_membs);
    [H2_LOW(i), H2_NORM(i), H2_HIGH(i)] = fuzzify(d_H2(i),
    →H2_LOW_membs, H2_NORM_membs, H2_HIGH_membs);
end
%Obtain crisp temperature setpoint array(response)
for i=1:length(d_CH4)
    [CO2_A_DN_TEMP(i) CO2_z_DN_TEMP(i) CO2_A_AB_RITE(i)
    →CO2_z_AB_RITE(i) CO2_A_UP_TEMP(i) CO2_z_UP_TEMP(i)] = ...
        Dfuzzify(CO2_DN_TEMP_membs, CO2_AB_RITE_membs,
        →CO2_UP_TEMP_membs, CO2_LOW(i), CO2_NORM(i), CO2_HIGH(i));
    [CH4_A_DN_TEMP(i) CH4_z_DN_TEMP(i) CH4_A_AB_RITE(i)
    →CH4_z_AB_RITE(i) CH4_A_UP_TEMP(i) CH4_z_UP_TEMP(i)] = ...
        Dfuzzify(CH4_DN_TEMP_membs, CH4_AB_RITE_membs,
        →CH4_UP_TEMP_membs, CH4_HIGH(i), CH4_NORM(i), CH4_LOW(i));
    [H2_A_DN_TEMP(i) H2_z_DN_TEMP(i) H2_A_AB_RITE(i) H2_z_AB_RITE(i)
    → H2_A_UP_TEMP(i) H2_z_UP_TEMP(i)] = ...
        Dfuzzify(H2_DN_TEMP_membs, H2_AB_RITE_membs,
        →H2_UP_TEMP_membs, H2_LOW(i), H2_NORM(i), H2_HIGH(i));
end
%final temperature setpoint is centroid of entire response area: All
→ gases
for i=1:length(d_CO2)
    temp_SP(i) = ((CO2_A_DN_TEMP(i)*CO2_z_DN_TEMP(i) + CO2_A_AB_RITE
    →(i)*CO2_z_AB_RITE(i) + CO2_A_UP_TEMP(i)*CO2_z_UP_TEMP(i)) ...
        + (CH4_A_DN_TEMP(i)*CH4_z_DN_TEMP(i) + CH4_A_AB_RITE(i)*
        →CH4_z_AB_RITE(i) + CH4_A_UP_TEMP(i)*CH4_z_UP_TEMP(i))...
        + (H2_A_DN_TEMP(i)*H2_z_DN_TEMP(i) + H2_A_AB_RITE(i)*
        →H2_z_AB_RITE(i) + H2_A_UP_TEMP(i)*H2_z_UP_TEMP(i)))...
        / ((CO2_A_DN_TEMP(i) + CO2_A_AB_RITE(i) + CO2_A_UP_TEMP(i))
        →...
        + (CH4_A_DN_TEMP(i) + CH4_A_AB_RITE(i) + CH4_A_UP_TEMP(i))
        →...
        + (H2_A_DN_TEMP(i) + H2_A_AB_RITE(i) + H2_A_UP_TEMP(i)));
end
%final temperature setpoint is centroid of entire response area: CO2
→ & H2
%for i=1:length(d_H2)
%    temp_SP(i) = (CO2_A_DN_TEMP(i)*CO2_z_DN_TEMP(i) + CO2_A_AB_RITE
→(i)*CO2_z_AB_RITE(i) + CO2_A_UP_TEMP(i)*CO2_z_UP_TEMP(i) ...
%     + H2_A_DN_TEMP(i)*H2_z_DN_TEMP(i) + H2_A_AB_RITE(i)*
→H2_z_AB_RITE(i) + H2_A_UP_TEMP(i)*H2_z_UP_TEMP(i))...
%        / ( CO2_A_DN_TEMP(i) + CO2_A_AB_RITE(i) + CO2_A_UP_TEMP(i)
→...
%        + H2_A_DN_TEMP(i) + H2_A_AB_RITE(i) + H2_A_UP_TEMP(i));
%end
%final temperature setpoint is centroid of entire response area: CO2
%for i=1:length(d_CO2)
%    temp_SP(i) = (CO2_A_DN_TEMP(i)*CO2_z_DN_TEMP(i) + CO2_A_AB_RITE
→(i)*CO2_z_AB_RITE(i) + CO2_A_UP_TEMP(i)*CO2_z_UP_TEMP(i))...
%        / (CO2_A_DN_TEMP(i) + CO2_A_AB_RITE(i) + CO2_A_UP_TEMP(i));
%end
%final temperature setpoint is centroid of entire response area: H2
%for i=1:length(d_H2)
%    temp_SP(i) = (H2_A_DN_TEMP(i)*H2_z_DN_TEMP(i) + H2_A_AB_RITE(i)
→*H2_z_AB_RITE(i) + H2_A_UP_TEMP(i)*H2_z_UP_TEMP(i))...
%        / (H2_A_DN_TEMP(i) + H2_A_AB_RITE(i) + H2_A_UP_TEMP(i));
%end
%final temperature setpoint is centroid of entire response area: CH4
%for i=1:length(d_CH4)
```

```
%     temp_SP(i) = (CH4_A_DN_TEMP(i)*CH4_z_DN_TEMP(i) + CH4_A_AB_RITE
→(i)*CH4_z_AB_RITE(i) + CH4_A_UP_TEMP(i)*CH4_z_UP_TEMP(i))...
%          / (CH4_A_DN_TEMP(i) + CH4_A_AB_RITE(i) + CH4_A_UP_TEMP(i));
%end
figure;
%Plot inputs
subplot(3,1,1), plot(1:length(CO2_avg), CO2_avg, 1:length(CH4_avg),
→CH4_avg, 1:length(H2_avg), H2_avg);
grid on; hold on;
title('Gas Readings');
ylabel('ADC reading');
axis(subplot(311), [0 (length(CO2_avg)) 0 800]);

%Plot derivatives
subplot(3,1,2), plot(1:length(d_CO2), d_CO2, 1:length(d_CH4), d_CH4,
→ 1:length(d_H2), d_H2);
grid on; hold on;
title('1st Derivative Inputs');
ylabel('d(gas)/dt');
legend('CO2','CH4','H2');
axis(subplot(312), [0 (length(CO2_avg)) -50 110]);

%Plot controller response
subplot(3,1,3), plot(1:length(d_CO2),temp_SP);grid on; hold on;
grid on; hold on;
%title('Input H2');
%ylabel('H2');
title('Response');
ylabel('Temp SP (deg C)');
xlabel('Sample number');

%Plot range of values for tuning
%plot(d_H2, temp_SP);
%grid on; hold on;
%title('Input d(H2)/dt');
%xlabel('d(H2)/dt');
%ylabel('Temperature Set Point (deg C)');

%figure;
%Plot range of values for tuning
%plot(d_CH4, temp_SP);
%grid on; hold on;
%title('Input d(CH4)/dt');
%xlabel('d(CH4)/dt');
%ylabel('Temperature Set Point (deg C)');

%EOF
```

Listing B.6: fuzzifyV2.m

```
function [LOW, NORM, HIGH] = fuzzify(d_GAS, LOW_membs, NORM_membs,
→HIGH_membs)
% A function to fuzzify.
% The lower and upper membership and response functions must be
→trapezoidal
% and the centre membership and response functions must be
→triangular for
```

```
% the membership calculations.
%Membership rules LOW
    if (d_GAS < LOW_membs(2))
        LOW = 1;
    end
    if (d_GAS >= LOW_membs(2)) & (d_GAS < LOW_membs(3))
        LOW = 1 - (d_GAS -  LOW_membs(2))/(LOW_membs(3)-LOW_membs(2)
         →);
    end
    if (d_GAS >= LOW_membs(3))
        LOW = 0;
    end
    %Membership rules NORM
    if (d_GAS < NORM_membs(1) | d_GAS >= NORM_membs(3))
        NORM = 0;
    end
    if (d_GAS >= NORM_membs(1) & d_GAS < NORM_membs(2))
        NORM = (d_GAS - NORM_membs(1)) / (NORM_membs(2) - NORM_membs
         →(1));
    end
    if (d_GAS >=  NORM_membs(2) & d_GAS < NORM_membs(3))
        NORM = 1 - (d_GAS - NORM_membs(2)) / (NORM_membs(3) -
         →NORM_membs(2));
    end
    %Membership rules HIGH
    if (d_GAS < HIGH_membs(1))
        HIGH = 0;
    end
    if (d_GAS >= HIGH_membs(1) & d_GAS < HIGH_membs(2) )
        HIGH = (d_GAS - HIGH_membs(1)) / (HIGH_membs(2) - HIGH_membs
         →(1));
    end
    if d_GAS >= HIGH_membs(2)
        HIGH = 1;
    end
%EOF
```

Listing B.7: DfuzzifyV2.m

```
function [ Area_DN_TEMP z_DN_TEMP Area_AB_RITE z_AB_RITE
 →Area_UP_TEMP z_UP_TEMP ] = ...
    Dfuzzify(DN_TEMP_membs, AB_RITE_membs, UP_TEMP_membs, LOW, NORM,
     → HIGH)
% A function to de-fuzzify.
% The lower and upper membership and response functions are
 →trapezoidal,
% and the centre membership and response functions triangular for
% the area and centroidal calculations.

%Calculate areas
Area_DN_TEMP = LOW(length(LOW)) * ((DN_TEMP_membs(2) - DN_TEMP_membs
 →(1)) + (DN_TEMP_membs(3) - DN_TEMP_membs(2))/2 );
Area_AB_RITE = NORM(length(NORM)) * (AB_RITE_membs(3) -
 →AB_RITE_membs(1))/2;
```

```
Area_UP_TEMP = HIGH(length(HIGH)) * ((UP_TEMP_membs(3) -
→UP_TEMP_membs(2)) + (UP_TEMP_membs(2) - UP_TEMP_membs(1))/2 );

%Work out response areas horizontal centroids
z_DN_TEMP = ((DN_TEMP_membs(2)-DN_TEMP_membs(1))^2 + 2*DN_TEMP_membs
→(1)*(DN_TEMP_membs(2)-DN_TEMP_membs(1)) + ...
    ((DN_TEMP_membs(3)-DN_TEMP_membs(2))^2)/3 +...
   DN_TEMP_membs(2) * (DN_TEMP_membs(3)-DN_TEMP_membs(2)))   /...
    (2*(DN_TEMP_membs(2)-DN_TEMP_membs(1)) + (DN_TEMP_membs(3) -
     →DN_TEMP_membs(2)));

z_AB_RITE = (2*((AB_RITE_membs(2)-AB_RITE_membs(1))^2)/3 +
→AB_RITE_membs(1) * (AB_RITE_membs(2)-AB_RITE_membs(1)) + ...
    ((AB_RITE_membs(3)-AB_RITE_membs(2))^2)/3 + AB_RITE_membs(2)*(
     →AB_RITE_membs(3)-AB_RITE_membs(2))) / ...
    (AB_RITE_membs(2)-AB_RITE_membs(1) + AB_RITE_membs(3)-
     →AB_RITE_membs(2));

z_UP_TEMP = (4*((UP_TEMP_membs(2)-UP_TEMP_membs(1))^2)/3 + 2*
→UP_TEMP_membs(1)*(UP_TEMP_membs(2)-UP_TEMP_membs(1)) +...
    3*((UP_TEMP_membs(3)- UP_TEMP_membs(2))^2)/2 + 3*UP_TEMP_membs
     →(2)*(UP_TEMP_membs(3)-UP_TEMP_membs(2))) /...
    (2*(UP_TEMP_membs(2)-UP_TEMP_membs(1)) + 3*(UP_TEMP_membs(3)-
     →UP_TEMP_membs(2)));
```

B.2 C Code for the Atmega8

One version of each file is presented in Listings B.8 to B.18, as later versions of the code contain roughly the same code as earlier versions, just with fewer bugs and extra signal processing and fuzzy code. The semaphore LED code has not yet been included due to the fact that the system is still rather too immature to be concerned about user interfaces just yet.

Listing B.8: Main File

```c
// Fuzzy Logic app. for the ATMega8
// Takes a value from the ADC, fuzzyfies & de-fuzzifies and returns
 →areas and centroids.
// Includes
#include <inttypes.h>
#include <avr/interrupt.h>
#include <uart.h>
#include <adc.h>
#include <timer.h>
#include <avr/sleep.h>
#include <avr/io.h>
#include <avr/pgmspace.h>        // included to enable the writing of
 → strings from rom
#include <fuzzy_funcs.h>
#include <adc.inc>
// constants
#define FCPU          1000000       /* CPU speed */
#define temp_buf_sz 5
#define co2_buf_sz 6             //These can be increased as
 →necessary to smooth signals. min 3
#define ch4_buf_sz 5
#define h2_buf_sz 6
#define avgs_buf_sz 3    //min 2
//
 →----------------------------------------------------------------------------
//a structure of interrupt flags
volatile struct
{
        uint8_t tmr0_int: 1;     //assign a 1 bit flag
        uint8_t adc_int: 1;
}int_flags;
float data;      //A general purpose var to fetch stuff, often ADC
 →readings.
//------------------------------------------------------------
// Main - a simple loop program
int main(void)
{
        //3 floats to store the output from fuzzify() and input to
         →defuzzify()
        //[LOW, NORM, HIGH]
        float CO2_membership[3];
```

```
//An array of 3 areas from defuzzify
        float CO2_areas[3];
//An array of 3 centroids from defuzzify
        float CO2_centrds[3];
//CO2 input membership function data
        static float CO2_LOW_membs[3] = {-16, -16, -2};
        static float CO2_NORM_membs[3] = {-16, 0 ,10};
        static float CO2_HIGH_membs[3] = {2, 10, 45};
//responses; used to defuzzify
        static float CO2_DN_TEMP_membs[3] = {15, 15, 35};
        static float CO2_AB_RITE_membs[3] = {10, 30, 52};
        static float CO2_UP_TEMP_membs[3] = {29, 39, 45};
        //Gas reading buffers
        float temp[temp_buf_sz];
        float temp_avg;
        float temp_SP;
        float CO2[co2_buf_sz];// [0->6]
        float CO2_avgs[avgs_buf_sz];    //[0->2]; keep 3 averages
        float d_CO2[(avgs_buf_sz - 1)]; //[0->1]; keep 2 1st
         →derivatives
        float CH4[ch4_buf_sz];
        float CH4_avgs[avgs_buf_sz];
        float d_CH4[(avgs_buf_sz - 1)];
        float H2[h2_buf_sz];
        float H2_avgs[avgs_buf_sz];
        float d_H2[(avgs_buf_sz - 1)];
        //Initialise buffers
        CO2_avgs[(avgs_buf_sz-1)] = 0;
        d_CO2[(avgs_buf_sz-2)] = 0;
        CH4_avgs[(avgs_buf_sz-1)] = 0;
        d_CO2[(avgs_buf_sz-2)] = 0;
        H2_avgs[(avgs_buf_sz-1)] = 0;
        d_H2[(avgs_buf_sz-2)] = 0;
        uint8_t i;   //indexing variable
        USART_Init(UART_BAUD_SELECT(2400,FCPU),USART_SET_8_1_N);
         →//2400 bps, 8 data bits, 1 stop bit, no parity
        sei(); // enable interrupts
        uart_puts_p(PSTR("* * * * * CO2, CH4 & H2 Test App * * * * *
         →")); //Welcome message
        USART_Transmit(0x0d); USART_Transmit(0x0a); // new line
        timer_init();
        MCUCR |= _BV(SE);          //Enable sleep mode (idle)
        while(1)// forever
        {
                if (int_flags.tmr0_int)
                {
                        int_flags.tmr0_int = 0;
                        temp_avg = 0;
//------------------------------------------------------------------
                        //left shift array of [temp_buf_sz] temp
                         →values
                        for (i=0; i<(temp_buf_sz - 1); i++)
                        {
```

```
                temp[i] = temp[i+1];
                temp_avg += temp[i];
        }
        //Left shift arrays of averages
        for (i=0; i<(avgs_buf_sz-1); i++)          //
         →max i = (avgs_buf_sz - 2) (1)
        {
                CO2_avgs[i] = CO2_avgs[i+1];       //
                 →CO2_avgs[0] = CO2_avgs[1] ->
                 →CO2_avgs[2]
                CH4_avgs[i] = CH4_avgs[i+1];
                H2_avgs[i] = H2_avgs[i+1];
        }
        //Left shift array of CO2 samples
        for (i=0; i<(co2_buf_sz - 1); i++)         //
         →max i = (co2_buf_sz - 2) (5)
        {
                CO2[i] = CO2[i+1];          // 1) CO2[0]
                 → = CO2[1]  ->  CO2[5] = CO2[6]
                CO2_avgs[(avgs_buf_sz-1)] += CO2[i];
        }
        //Left shift array of CH4 samples
        for (i=0; i<(ch4_buf_sz - 1); i++)         //
         →max i = (co2_buf_sz - 2) (5)
        {
                CH4[i] = CH4[i+1];          // 1) CO2[0]
                 → = CO2[1]  ->  CO2[5] = CO2[6]
                CH4_avgs[(avgs_buf_sz-1)] += CH4[i];
        }
        //Left shift array of H2 samples
        for (i=0; i<(h2_buf_sz - 1); i++)          //
         →max i = (co2_buf_sz - 2) (5)
        {
          H2[i] = H2[i+1];        // 1) CO2[0] = CO2
           →[1]  ->  CO2[5] = CO2[6]
          H2_avgs[(avgs_buf_sz-1)] += H2[i];
        }
        //Left shift elements of 1st derivative
         →arrays
        for (i=0; i<(avgs_buf_sz-2); i++)          //i=
         → 0->1
        {
                d_CO2[i] = d_CO2[i+1];
                d_CH4[i] = d_CH4[i+1];
                d_H2[i] = d_H2[i+1];
        }
//------------------------------------------------------------------
        //get readings all at once to reduce noise
        adc_init_ch1(); //Temp actually on CH5
        temp[(temp_buf_sz - 1)] = (0x3ff - 0x070 -
         →ADC_read()); //Append last ADC reading

        adc_init_ch2();   //CO2 actually on CH1
        CO2[(co2_buf_sz - 1)] = 0x3ff - ADC_read();

        adc_init_ch3(); //CH4 actually on ch2
        CH4[(ch4_buf_sz - 1)] = ADC_read();

        adc_init_ch5(); //H2 actually on CH3
        H2[(h2_buf_sz - 1)] = ADC_read();
```

```
//----------------------------------------------------------------------
                    //Finish averaging
                    temp_avg += temp[(temp_buf_sz-1)];
                    temp_avg = temp_avg / (temp_buf_sz);

                    CO2_avgs[(avgs_buf_sz-1)] += CO2[(co2_buf_sz
                     → -1)];        //CO2[0->5] + CO2[6]; add most
                     → recent reading
                    CO2_avgs[(avgs_buf_sz-1)] = CO2_avgs[(
                     →avgs_buf_sz-1)] / (co2_buf_sz+1);

                    CH4_avgs[(avgs_buf_sz-1)] += CH4[(ch4_buf_sz
                     → -1)];
                    CH4_avgs[(avgs_buf_sz-1)] = CH4_avgs[(
                     →avgs_buf_sz-1)] / (ch4_buf_sz+1);

                    H2_avgs[(avgs_buf_sz-1)] += H2[(h2_buf_sz
                     →-1)];
                    H2_avgs[(avgs_buf_sz-1)] = H2_avgs[(
                     →avgs_buf_sz-1)] / (h2_buf_sz+1);
//----------------------------------------------------------------------
                    //get arrays of 1st derivatives
                    for (i=0; i<(avgs_buf_sz-1); i++)        //i
                     →=0->1
                    {
                            d_CO2[i] = CO2_avgs[i+1] - CO2_avgs[
                             →i]; //d_CO2[0->1] = CO2_avgs[1->2]
                             → - CO2_avgs[0->1]
                            d_CH4[i] = CH4_avgs[i+1] - CH4_avgs[
                             →i];
                            d_H2[i] = H2_avgs[i+1] - H2_avgs[i];
                    }
//----------------------------------------------------------------------
                    //Output to screen
                    uart_puts_p(PSTR("Temp: "));
                    for(i=(temp_buf_sz-1); i<(temp_buf_sz); i++)
                     →      //i= 0->6
                    {
                            uart_put_num(temp[i]);
                            uart_puts_p(PSTR(" "));
                    }
                    //USART_Transmit(0x0d); USART_Transmit(0x0a)
                     →; //new line

                    uart_puts_p(PSTR("Temp_avg: "));
                    uart_put_num(temp_avg);
                    //USART_Transmit(0x0d); USART_Transmit(0x0a)
                     →;
                    uart_puts_p(PSTR(" "));

                    uart_puts_p(PSTR("CO2: "));
                    for(i=(co2_buf_sz-1); i<(co2_buf_sz); i++)
                     →        //i= 0->6
                    {
                            uart_put_num(CO2[i]);
                            uart_puts_p(PSTR(" "));
                    }
                    //USART_Transmit(0x0d); USART_Transmit(0x0a)
                     →; //new line

                    uart_puts_p(PSTR("CO2_avg: "));
```

```
for (i=(avgs_buf_sz -1); i<(avgs_buf_sz); i
 →++)    //i= 0->2
{
        uart_put_num(CO2_avgs[i]);
        uart_puts_p(PSTR(" "));
}
//USART_Transmit(0x0d); USART_Transmit(0x0a)
 →; //new line
uart_puts_p(PSTR("d_CO2: "));
for (i=(avgs_buf_sz -2); i<(avgs_buf_sz -1); i
 →++) //i= 0->1
{
        uart_put_num(d_CO2[i]);
        uart_puts_p(PSTR(" "));
}
//USART_Transmit(0x0d); USART_Transmit(0x0a)
 →; //new line

uart_puts_p(PSTR("CH4: "));
for (i=(ch4_buf_sz -1); i<(ch4_buf_sz); i++)
 →       //i= 0->1
{
        uart_put_num(CH4[i]);
        uart_puts_p(PSTR(" "));
}
//USART_Transmit(0x0d); USART_Transmit(0x0a)
 →;
uart_puts_p(PSTR("CH4_avg: "));
for (i=(avgs_buf_sz -1); i<(avgs_buf_sz); i
 →++)    //i= 0->1
{
        uart_put_num(CH4_avgs[i]);
        uart_puts_p(PSTR(" "));
}
//USART_Transmit(0x0d); USART_Transmit(0x0a)
 →;

uart_puts_p(PSTR("d_CH4: "));
for (i=(avgs_buf_sz -2); i<(avgs_buf_sz -1); i
 →++) //i= 0->1
{
        uart_put_num(d_CH4[i]);
        uart_puts_p(PSTR(" "));
}
//USART_Transmit(0x0d); USART_Transmit(0x0a)
 →;

uart_puts_p(PSTR("H2: "));
for (i=(h2_buf_sz -1); i<(h2_buf_sz); i++)
 →         //i= 0->1
{
   uart_put_num(H2[i]);
   uart_puts_p(PSTR(" "));
}
//USART_Transmit(0x0d); USART_Transmit(0x0a)
 →;
uart_puts_p(PSTR("H2_avg: "));
for (i=(avgs_buf_sz -1); i<(avgs_buf_sz); i
 →++)    //i= 0->1
{
   uart_put_num(H2_avgs[i]);
```

```
                        uart_puts_p(PSTR(" "));
                    }
                    //USART_Transmit(0x0d); USART_Transmit(0x0a)
                     →;
                    uart_puts_p(PSTR("d_H2: "));
                    for (i=(avgs_buf_sz -2); i<(avgs_buf_sz -1); i
                     →++) //i= 0->1
                    {
                        uart_put_num(d_H2[i]);
                        uart_puts_p(PSTR(" "));
                    }
                    //USART_Transmit(0x0d); USART_Transmit(0x0a)
                     →;
                    USART_Transmit(0x0d); USART_Transmit(0x0a);
                    //fuzzify(CO2_membership, d_CO2_avg,
                     →CO2_LOW_membs, CO2_NORM_membs,
                     →CO2_HIGH_membs);
                    //determine m/ship vals

                    /*
                    USART_Transmit(0x0d); USART_Transmit(0x0a);
                     →// new line
                    uart_puts_p(PSTR("main(): LOW: "));     //
                     →fuzzify debugging output
                    uart_put_num(CO2_membership[0]);
                    uart_puts_p(PSTR("  NORM: "));
                    uart_put_num(CO2_membership[1]);
                    uart_puts_p(PSTR("  HIGH: "));
                    uart_put_num(CO2_membership[2]);
                    USART_Transmit(0x0d); USART_Transmit(0x0a);
                     →// new line
                    */
                    //defuzzify(CO2_areas, CO2_centrds,
                     →CO2_membership, CO2_DN_TEMP_membs,
                     →CO2_AB_RITE_membs, CO2_UP_TEMP_membs);
                    //final temperature setpoint is centroid of
                     →entire response area
                    //temp_SP = (CO2_areas[0]*CO2_centrds[0] +
                     →CO2_areas[1]*CO2_centrds[1] + CO2_areas
                     →[2]*CO2_centrds[2]) / (CO2_areas[0] +
                     →CO2_areas[1] + CO2_areas[2]);

                    //USART_Transmit(0x0d); USART_Transmit(0x0a)
                     →; // new line

                    //uart_puts_p(PSTR("  Temp_SP: "));
                    //uart_put_num(temp_SP);
                    USART_Transmit(0x0d);   USART_Transmit(0x0a)
                     →; // new line
                }
                sleep_mode();
        }
}
```

Listing B.9: Fuzzy Logic Functions

```c
/*Fuzzify and de-fuzzify functions live in here.
 Written by Terry Sullavan for USQ ENG4111/4112
*/
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <math.h>
#include <avr/io.h>
#include <uart.h>
#include <adc.h>
#include <adc.inc>                // setup the tx and rx que size here
#include <avr/sleep.h>
void fuzzify(float membership[], float gas_reading, float LOW_membs
 →[], float NORM_membs[], float HIGH_membs[])
{
        /*
        USART_Transmit(0x0d); USART_Transmit(0x0a); // new line
        uart_puts_p(PSTR("fuzzify(): LOW_membs: "));   //fuzzify
         →debugging output
        uart_put_num(LOW_membs[1]);
        uart_puts_p(PSTR("  NORM_membs: "));
        uart_put_num(NORM_membs[1]);
        uart_puts_p(PSTR("  HIGH_membs: "));
        uart_put_num(HIGH_membs[1]);
        */
        //Membership rules LOW
        if (gas_reading < LOW_membs[1])
        {
                membership[0] = 1;
        }
        if ((gas_reading >= LOW_membs[1]) && (gas_reading <
         →LOW_membs[2]))
        {
                membership[0] = 1 - (gas_reading -  LOW_membs[1])/(
                 →LOW_membs[2]-LOW_membs[1]);
        }
        if (gas_reading >= LOW_membs[2])
        {
                membership[0] = 0;
        }
        //Membership rules NORM
        if ((gas_reading < NORM_membs[0]) || (gas_reading >=
         →NORM_membs[2]))
        {
                membership[1] = 0;
        }
        if ((gas_reading >= NORM_membs[0]) && (gas_reading <
         →NORM_membs[1]))
        {
                membership[1] = (gas_reading - NORM_membs[0]) / (
                 →NORM_membs[1] - NORM_membs[0]);
        }
        if ((gas_reading >=  NORM_membs[1]) && (gas_reading <
         →NORM_membs[2]))
        {
                membership[1] = 1 - (gas_reading - NORM_membs[1]) /
                 →(NORM_membs[2] - NORM_membs[1]);
        }
        //Membership rules HIGH
        if (gas_reading < HIGH_membs[0])
        {
                membership[2] = 0;
        }
        if ((gas_reading >= HIGH_membs[0]) && (gas_reading <
         →HIGH_membs[1]))
        {
                membership[2] = (gas_reading - HIGH_membs[0]) / (
                 →HIGH_membs[1] - HIGH_membs[0]);
        }
        if (gas_reading >= HIGH_membs[1])
        {
                membership[2] = 1;
        }
        /*
        USART_Transmit(0x0d); USART_Transmit(0x0a); // new line
        uart_puts_p(PSTR("fuzzify():  LOW: ")); //fuzzify debugging
         →output
        uart_put_num(membership[0]);
        uart_puts_p(PSTR("  NORM: "));
        uart_put_num(membership[1]);
        uart_puts_p(PSTR("  HIGH: "));
        uart_put_num(membership[2]);
        */
}
void defuzzify(float areas[], float centrds[], float membership[],
 →float DN_TEMP_membs[], float AB_RITE_membs[], float UP_TEMP_membs
 →[])
{
//Calculate areas
//Area_DN_TEMP
        areas[0] = membership[0] * ((DN_TEMP_membs[1] -
         →DN_TEMP_membs[0]) + (DN_TEMP_membs[2] - DN_TEMP_membs[1])
         →/2);
//Area_AB_RITE
areas[1] = membership[1] * (AB_RITE_membs[2] - AB_RITE_membs[0])/2;
//Area_UP_TEMP
areas[2] = membership[2] * ((UP_TEMP_membs[2] - UP_TEMP_membs[1]) +
 →(UP_TEMP_membs[1] - UP_TEMP_membs[0])/2);

        /*uart_puts_p(PSTR("defuzzify():  A_DN_TEMP: "));          //
         →debugging output
        uart_put_num(areas[0]);

        uart_puts_p(PSTR("  A_AB_RITE: "));
        uart_put_num(areas[1]);

        uart_puts_p(PSTR("  A_UP_TEMP: "));
        uart_put_num(areas[2]);

        USART_Transmit(0x0d); USART_Transmit(0x0a); // new line
        */
//Work out response areas horizontal centroids ********This doesn't
 →have to be done every time!!!*******
//z_DN_TEMP
        centrds[0] = ((DN_TEMP_membs[1]-DN_TEMP_membs[0])*(
         →DN_TEMP_membs[1]-DN_TEMP_membs[0]) + 2*DN_TEMP_membs[0]*(
           →DN_TEMP_membs[1] - DN_TEMP_membs[0]) + ((DN_TEMP_membs[2]-
           →DN_TEMP_membs[1])*(DN_TEMP_membs[2]-DN_TEMP_membs[1]))/3 +
           → (DN_TEMP_membs[1])*(DN_TEMP_membs[2]-DN_TEMP_membs[1])) /
           → (2*(DN_TEMP_membs[1]-DN_TEMP_membs[0]) + (DN_TEMP_membs
           →[2] - DN_TEMP_membs[1]));
//z_AB_RITE
          centrds[1] = (2*((AB_RITE_membs[1]-AB_RITE_membs[0])*(
           →AB_RITE_membs[1]-AB_RITE_membs[0]))/3 +
                   AB_RITE_membs[0] * (AB_RITE_membs[1]-AB_RITE_membs
                    →[0]) + ((AB_RITE_membs[2]-AB_RITE_membs[1])*(
                    →AB_RITE_membs[2]-AB_RITE_membs[1]))/3 +
                    →AB_RITE_membs[1]*(AB_RITE_membs[2]-AB_RITE_membs
                    →[1])) / (AB_RITE_membs[1]-AB_RITE_membs[0] +
                    →AB_RITE_membs[2]-AB_RITE_membs[1]);
//z_UP_TEMP
          centrds[2] = (4*((UP_TEMP_membs[1]-UP_TEMP_membs[0])*(
           →UP_TEMP_membs[1]-UP_TEMP_membs[0]))/3 + 2*UP_TEMP_membs
           →[0]*(UP_TEMP_membs[1]-UP_TEMP_membs[0]) + 3*((
           →UP_TEMP_membs[2]- UP_TEMP_membs[1])*(UP_TEMP_membs[2]-
           →UP_TEMP_membs[1]))/2 + 3*UP_TEMP_membs[1]*(UP_TEMP_membs
           →[2]-UP_TEMP_membs[1])) / (2*(UP_TEMP_membs[1]-
           →UP_TEMP_membs[0]) + 3*(UP_TEMP_membs[2]-UP_TEMP_membs[1]))
           →;

          /*USART_Transmit(0x0d); USART_Transmit(0x0a); // new line

          uart_puts_p(PSTR("defuzzify():  z_DN_TEMP: ")); //debugging
           →output
          uart_put_num(centrds[0]);

          uart_puts_p(PSTR("   z_AB_RITE: "));
          uart_put_num(centrds[1]);

          uart_puts_p(PSTR("   z_UP_TEMP: "));
          uart_put_num(centrds[2]);

          USART_Transmit(0x0d); USART_Transmit(0x0a); // new line*/
}
```

Listing B.10: Fuzzy Logic Header

```
/*Fuzzy functions header file*/

void fuzzify(float membership[], float gas_reading, float LOW_membs
 →[], float NORM_membs[], float HIGH_membs[]);

void defuzzify(float CO2_areas[], float CO2_centrds[], float
 →CO2_membership[], float CO2_DN_TEMP_membs[], float
 →CO2_AB_RITE_membs[], float CO2_UP_TEMP_membs[]);
```

Listing B.11: ADC Subroutines

```
//
 →***********************************************************************/
 →
//Using the ADC.
// This file contains the ADC initiation and interrupt routines.
//
// Last modified: 26 October 2006
// Changed to work with WinAVR 20060421
// Written 3 Nov 2005 By Murray Horn.
//
 →***********************************************************************/
 →
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <avr/io.h>
#include <uart.h>
#include <adc.h>
#include <adc.inc>              // setup the tx and rx que size here
#include <avr/sleep.h>

//------------------------------------------------------------
extern float data;
extern volatile struct
{
        uint8_t tmr0_int: 1;    //assign a 1 bit flag
        uint8_t adc_int: 1;
}int_flags;

#define circular_t      uint8_t
//------------------------------------------------------------
// Static Variables
static uint16_t ADC_Buf[ADC_BUFSIZE];
static volatile circular_t ADC_head;
static volatile circular_t ADC_tail;
//------------------------------------------------------------
//------------------------------------------------------------
SIGNAL(SIG_ADC)
{
//      Read the received data
        data = ADCL; // always read the lower byte first
        data = data + (ADCH << 8);
        int_flags.adc_int = 1;
}
//------------------------------------------------------------
//ADC channel initialisation routines
void adc_init_ch0(void)
{
        int_flags.adc_int = 0;
        ADMUX = 0;       //select ch0
        ADMUX = (1 << REFS0); // select vcc as a ref with right
         →justification
        //enable the adc, set interrupts, set the prescaler 1MHz/8
        ADCSRA = (1 << ADEN) | (1 << ADIF) | (1 << ADIE) | (1 <<
         →ADPS1) | (1 << ADPS0);
//      uart_puts_p(PSTR("Init_ch0")); // into string
//      USART_Transmit(0x0d);   USART_Transmit(0x0a); // new line
}
```

```
void adc_init_ch1(void)
{
        int_flags.adc_int = 0;
        ADMUX = 0;       //select ch1 for CO2
        ADMUX = (1 << REFS0) | (1 << MUX0); // select vcc as a ref
         →with right justification
        ADCSRA = (1 << ADEN) | (1 << ADIF) | (1 << ADIE) | (1 <<
         →ADPS1) | (1 << ADPS0);
//      uart_puts_p(PSTR("Init_ch1")); // into string
//      USART_Transmit(0x0d);   USART_Transmit(0x0a); // new line
}
void adc_init_ch2(void)
{
        int_flags.adc_int = 0;
        ADMUX = 0;       //select ch2
        ADMUX = (1 << REFS0) | (1 << MUX1) ; // select vcc as a ref
         →with right justification
        //enable the adc, set interrupts, set the prescaler 1MHz/8
        ADCSRA = (1 << ADEN) | (1 << ADIF) | (1 << ADIE) | (1 <<
         →ADPS1) | (1 << ADPS0);
//      uart_puts_p(PSTR("Init_ch2")); // into string
//      USART_Transmit(0x0d);   USART_Transmit(0x0a); // new line
}
void adc_init_ch3(void)
{
        int_flags.adc_int = 0;
        ADMUX = 0;       //select ch3
        ADMUX = (1 << REFS0) | (1 << MUX0) | (1 << MUX1) ; // select
         → vcc as a ref with right justification
        //enable the adc, set interrupts, set the prescaler 1MHz/8
        ADCSRA = (1 << ADEN) | (1 << ADIF) | (1 << ADIE) | (1 <<
         →ADPS1) | (1 << ADPS0);
//      uart_puts_p(PSTR("Init_ch2")); // into string
//      USART_Transmit(0x0d);   USART_Transmit(0x0a); // new line
}

void adc_init_ch5(void)
{
        int_flags.adc_int = 0;
        ADMUX = 0;       //select ch5 for temperature
        ADMUX = (1 << REFS0) | (1 << MUX0) | (1 << MUX2); // select
         →vcc as a ref with right justification
        //enable the adc, set interrupts, set the prescaler 1MHz/8
        ADCSRA = (1 << ADEN) | (1 << ADIF) | (1 << ADIE) | (1 <<
         →ADPS1) | (1 << ADPS0);
//      uart_puts_p(PSTR("Init_ch5")); // into string
//      USART_Transmit(0x0d);   USART_Transmit(0x0a); // new line
}

//———————————————————————————————————————————
//Obtains the average of 64 samples from the ADC and returns
//the average of 32 of these readings
float ADC_read(void)
{
        uint16_t i;
        uint16_t val_1;
        val_1 = 0;
        ADCSRA |= _BV(ADSC); //start the adc
        for (i=0; i<64; i++)
        {
                ADCSRA |= _BV(ADSC); //start the adc
                        //MCUCR |= _BV(SM0);
                        //sleep_mode();
                        while(int_flags.adc_int != 1)
                        {
//                              uart_puts_p(PSTR("ADC read wait loop
→; ")); // into string
//                              USART_Transmit(0x0d);
→USART_Transmit(0x0a); // new line
                        } //wait for adc
                        int_flags.adc_int = 0;
                        cli(); //atomic read
                        val_1 += data;
                        sei();
        }
                cli();
                val_1 = val_1 >> 6;
                //ADC_val += val_1;
                sei();

                ADCSRA = 0;
                ADMUX = 0;      //Disable ADC
                sei();
//              uart_puts_p(PSTR("ADC read returns: ")); // into
→string
//              uart_put_num(val_1);
//              USART_Transmit(0x0d);   USART_Transmit(0x0a); // new
→ line
        return val_1;
}
//——————————————————————————————————————
```

Listing B.12: ADC Subroutine Header

```
/*Header file for adc subroutines
*/

void adc_init_ch0(void);
void adc_init_ch1(void);
void adc_init_ch2(void);
void adc_init_ch3(void);
void adc_init_ch4(void);
void adc_init_ch5(void);

uint16_t ADC_get( void );
uint8_t ADC_DataInBuffer( void );
float ADC_read(void);
uint16_t buff_pop(void);
void buff_push(uint16_t data);
```

Listing B.13: ADC Include File

```
// Created: 2 Nov 2005
//==============================================================
//       UART Buffer Defines
//==============================================================
//       Val                Buffer Size
//
//       0                  2
//       1                  4
//       2                  8
//       3                  16
//       4                  32
//       5                  64
//       6                  128
//       7                  256
//
#define ADC_BUFFER_BITS 3       /* 2,4,8,16,32,64,128 or 256 bytes */
//
//==============================================================
// do not change !!!     do not change    !!!do not change
// the software expects the buffer size to be a multiple of 2
#define ADC_BUFSIZE (1 << (7 & (1+ADC_BUFFER_BITS)))
#define ADC_BUFFER_MASK ( ADC_BUFSIZE - 1 )
// do not change upto here!!!    do not change upto here!!!
//==============================================================
```

Listing B.14: Timer Subroutines

```
/*timer.c Routines and functions for the timer*/
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>        // included to enable the writing of
  → strings from rom

extern volatile struct
{
        uint8_t tmr0_int: 1;     //assign a 1 bit flag
        uint8_t adc_int: 1;      //not needed here
}int_flags;

enum { HEAT_H, HEAT_L, HEAT_L_READ, SLEEPING };
//H2 heater states, VH = 5v, VL = 1.4V (preheat for 90secs & read),
  →heater off.
ISR (TIMER0_OVF_vect)
{
   static uint16_t int_count;    //counter for timer; 305 is about 20
     → seconds
   static uint8_t sens_heat_st; //sensor heater state
```

```
    //uart_puts_p(PSTR("Entered timer ISR....  ")); // into string
    switch (sens_heat_st)
    {
        case HEAT_H:
            if (++int_count == 2046) //No. is time of the last state
            {
                sens_heat_st = HEAT_L;
                OCR1A = 880;
                int_count = 0;
            }
            break;
        case HEAT_L:
            if (++int_count == 29950)  //60 secs
            {
                sens_heat_st = HEAT_L_READ;
                OCR1A = 250;
                int_count = 0;
            }
            break;
        case HEAT_L_READ:
            if (++int_count == 44900)  //90secs
            {
                sens_heat_st = SLEEPING;
                int_count = 0;
                int_flags.tmr0_int = 1; //enable the main program to
                 →resume
            }
            break;
        case SLEEPING:
            if (++int_count == 5000)   //time to read & do fuzzy stuff
            {
                sens_heat_st = HEAT_H;
                OCR1A = 0;
                int_count = 0;
            }
    }
}
/*End of ISR*/

void timer_init (void)
{
    int_flags.tmr0_int = 0;
        //enable timer 0 to set sampling period
    TCCR0 |= _BV(CS01);
    //set up timer 1 for pwm (H2 sensor)
    TCCR1A |= _BV(WGM10) | _BV(WGM11) | _BV(WGM12) | _BV(COM1A1);
    TCCR1B |= _BV(CS10);
    /* Set PB3 value to 0. */
    OCR1A = 0;
    /* Enable PB3 as output. */
    DDRB = _BV(DDB1);
    TIMSK = _BV (TOIE0); /*Timer Interrupt Mask Bit0*/
    sei ();
}
```

Listing B.15: Timer Subroutine Header

```
/*timer.h function prototypes for the timer*/
void timer_init (void);
uint8_t timer_ovf( void );
```

Listing B.16: Serial USART Subroutines

```
// Based on ap note AVR306 by ATMEL
// Routines for interrupt controlled USART
// Last modified: 26 October 2006
// Changed to work with WinAVR 20060421
// Changed : 3 Nov 2005
// changed rx buffer access, added checks for non-atomic interrupt
 →pointer modification
// changed function name DataInReceiveBuffer to USART_DataRx
// modified: 10 APR 2005
// added double speed baud support;
// Modified by: Murray Horn
// this file includes the tx and rx routines for a single uart mega
 →avr.
// seperate tx and rx ques are available and can be sized in the .
 →inc file
// a string tx feature from rom has been added (uart_puts_p).
//      Includes
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>        // used for the string printing
 →feature
#include <uart.h>                // Prototypes
#define circular_t      uint8_t
// Static Variables
static uint8_t USART_RxBuf[USART_RX_BUFS];
static volatile circular_t USART_RxHead;
static volatile circular_t USART_RxTail;
static uint8_t USART_TxBuf[USART_TX_BUFS];
static volatile circular_t USART_TxHead;
static volatile circular_t USART_TxTail;

//————————————————————————————————————————————————————————————————
// Initialize USART
void USART_Init( uint16_t baudrate,uint8_t setup)
{
//————————————————————————————————————————
// the doublespeed selector
        if (baudrate > 0x7ff)
        {
                baudrate += 1;
                baudrate = baudrate >> 1;
                baudrate -= 1;
                UCSRA = 0;
        }
        else
                UCSRA = (1<<U2X);
//-----------------------------------
//              Set the baud rate
    UBRRH = (uint8_t)(0x0f &(baudrate >> 8));
        UBRRL = (uint8_t) baudrate;

//              Enable UART receiver and transmmitter and receive
 →int
    UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);
//              Set frame format: data bits, stop bits,parity etc
        UCSRC = (1<<URSEL)| setup ; // set the uart bits,baudrate,
         →stop bits
//              Flush buffers
        USART_RxTail = 0;
        USART_RxHead = 0;
        USART_TxTail = 0;
        USART_TxHead = 0;

}
//---------------------------------------------------------

//---------------------------------------------------------
//      RX Interrupt handler
// the only function that is allowed to change rx_head
SIGNAL(SIG_UART_RECV)
{
        uint8_t data;
        circular_t tmp_head,nxt_head,tmp_tail;
//      Read the received data
        data = UDR;
//      Calculate buffer index
        tmp_head = USART_RxHead;
        nxt_head = ( tmp_head + 1 ) & USART_RX_BUFFER_MASK;

// non atomic protection
        do
                tmp_tail = USART_RxTail;
        while (tmp_tail != USART_RxTail);

        if ( nxt_head == tmp_tail)
        {
// ERROR! Receive buffer overflow
        }
        else
        {
                USART_RxBuf[tmp_head] = data; // Store received data
                 → in buffer
                USART_RxHead = nxt_head;      // Store new index
        }
}
//---------------------------------------------------------

//---------------------------------------------------------
//      TX Interrupt handler
// the only function that is allowed to change tx_tail
SIGNAL(SIG_UART_DATA)
{
        circular_t tmp_tail,tmp_head;

// non atomic protection
        do
                tmp_head = USART_TxHead;
        while (tmp_head != USART_TxHead);
        tmp_tail = USART_TxTail;
//      Check if all data is transmitted
        if ( tmp_head != tmp_tail )
        {
//      Calculate buffer index
                UDR = USART_TxBuf[tmp_tail];      // Start transmition
                tmp_tail = ( tmp_tail + 1 ) & USART_TX_BUFFER_MASK;
                USART_TxTail = tmp_tail;                          //
                 →Store new index
        }
        else
        {
                UCSRB &= ~(1<<UDRIE);                         // Disable
                 →UDRE interrupt
        }
}
//--------------------------------------------------------------------

//--------------------------------------------------------------------
//      RX function
// the only function that is allowed to change rx_tail
uint8_t USART_Receive( void )
{
        circular_t tmp_tail,tmp_head;
        uint8_t b;
        do
                {
                tmp_tail = USART_RxTail;
                 →                        //
// double get the head to ensure no interrupt based corruption of
 →the pointer
                do
                        tmp_head = USART_RxHead;
                while (tmp_head != USART_RxHead);                 //
                 →non atomic protection
                }
        while (tmp_head == tmp_tail);             // Wait for
         →incomming data

        b = USART_RxBuf[tmp_tail];

        tmp_tail = ( tmp_tail + 1 ) & USART_RX_BUFFER_MASK; //
         →Calculate buffer index

        USART_RxTail = tmp_tail;
         →                    // Store new index

        return b;                                 // Return data
}
//--------------------------------------------------------------------

//--------------------------------------------------------------------
//      TX function
// the only function that is allowed to change USART_TxHead
void USART_Transmit( uint8_t data )
{
        circular_t tmp_head,nxt_head,tmp_tail;

//      Calculate buffer index
        tmp_head = USART_TxHead;
        nxt_head = ( tmp_head + 1 ) & USART_TX_BUFFER_MASK;       //
         →Next point in buffer

// wait for free space
        do
        {
// non atomic protection
                do      tmp_tail = USART_TxTail;
                while (tmp_tail != USART_TxTail);
        }
        while (nxt_head == tmp_tail);
        USART_TxBuf[tmp_head] = data;                   // Store
         →data in buffer
        USART_TxHead = nxt_head;
         →         // Store new index
        UCSRB |= (1<<UDRIE);                                    //
         →       Enable UDRE interrupt
}
//————————————————————————————————————————

//————————————————————————————————————————
uint8_t USART_DataRx( void )
{
        uint8_t again;
        circular_t tmp_tail,tmp_head;
// protection against interrupt interference (non atomic access)
        do      {
                again = 0;
                tmp_head = USART_RxHead;
                tmp_tail = USART_RxTail;
                if (tmp_head != USART_RxHead) again = 1;
                if (tmp_tail != USART_RxTail) again = 1;
                }
        while (again);
        if ( tmp_head == tmp_tail )
                return(0);
        else
                return(1);
}
//————————————————————————————————————————

//————————————————————————————————————————
// write a string to the usart from the program memory
void uart_puts_p(const char *progmem_s )
{
register char c;
        while ( (c = pgm_read_byte(progmem_s++)) )
                USART_Transmit(c);
}
//————————————————————————————————————————
//Converts a 16 bit number and puts it to the UART
void uart_put_num(float number) //Changed from uint_16t. Change back
 → if no good
{
        uint16_t w;
        uint16_t i;
        float b;
        w = number;
//a cheap word to ascii conversion
        for (i=0;i<4;i++)
        {
                b = (w >> 12) & 0x0f;
                if (b <= 0x09)
                        b = b + '0';
                else
                        b = b - 10 + 'A';
                USART_Transmit(b);
                w = w << 4;
        }
}
```

Listing B.17: Serial USART Subroutine Header

```
//A header file for functions of the uart
#include "uart.inc"
//      Prototypes
void USART_Init( uint16_t baudrate,uint8_t setup);
uint8_t USART_Receive( void );
void USART_Transmit(uint8_t data );
void uart_puts_p(const char *progmem_s );
uint8_t USART_DataRx( void );
void uart_put_num(float number);
```

Listing B.18: Serial USART Include File

```
// Last modified: 10 APR 2005
// added double speed baud support;
//===========================================================
//      UART Buffer Defines
//===========================================================
//      Val             Buffer Size
//
//      0               2
//      1               4
//      2               8
//      3               16
//      4               32
//      5               64
//      6               128
//      7               256
//
#define USART_RX_BUFFER_BITS 4      /* 2,4,8,16,32,64,128 or 256
 →bytes */
#define USART_TX_BUFFER_BITS 4      /* 2,4,8,16,32,64,128 or 256
 →bytes */
//===========================================================

//===========================================================
// do not change !!!    do not change   !!!do not change
// the software expects the buffer size to be a multiple of 2
#define USART_RX_BUFS (1 << (7 & (1+USART_RX_BUFFER_BITS)))
#define USART_TX_BUFS (1 << (7 & (1+USART_RX_BUFFER_BITS)))
```

```
#define USART_RX_BUFFER_MASK ( USART_RX_BUFS - 1 )
#define USART_TX_BUFFER_MASK ( USART_TX_BUFS - 1 )

#define UART_BAUD_SELECT(baudRate,xtalCpu) ((xtalCpu/baudRate/8)-1)

#define USART_SET_8_1_N 0x006
#define USART_SET_7_1_N 0x004
#define USART_SET_6_1_N 0x002
#define USART_SET_5_1_N 0x000

#define USART_SET_8_1_E 0x026
#define USART_SET_7_1_E 0x024
#define USART_SET_6_1_E 0x022
#define USART_SET_5_1_E 0x020

#define USART_SET_8_1_O 0x036
#define USART_SET_7_1_O 0x034
#define USART_SET_6_1_O 0x032
#define USART_SET_5_1_O 0x030

#define USART_SET_8_2_N 0x00e
#define USART_SET_7_2_N 0x00c
#define USART_SET_6_2_N 0x00a
#define USART_SET_5_2_N 0x008

#define USART_SET_8_2_E 0x02e
#define USART_SET_7_2_E 0x02c
#define USART_SET_6_2_E 0x02a
#define USART_SET_5_2_E 0x028

#define USART_SET_8_2_O 0x03e
#define USART_SET_7_2_O 0x03c
#define USART_SET_6_2_O 0x03a
#define USART_SET_5_2_O 0x038

// do not change upto here!!!   do not change upto here!!!
//=======================================================
```

Appendix C

Raw Captured Data

C.1 Raw Captured Data

This appendix contains raw data captured during laboratory trials. This data was used to obtain plots, and values of significance were used as a basis for synthesised data for simulations.

Listing C.1: Data for Trial 1; Pure CO_2

```
Temp: 00D9 Temp_avg: 00D9 CO2: 0000 CO2_avg: 0001 d_CO2: FFFC CH4: 0047 CH4_avg: 0049 d_CH4:
 →0000 H2: 00E0 H2_avg: 00D2 d_H2: 0009
Temp: 00DB Temp_avg: 00D9 CO2: 0000 CO2_avg: 0000 d_CO2: 0000 CH4: 004A CH4_avg: 004A d_CH4:
 →0000 H2: 00E2 H2_avg: 00D8 d_H2: 0006
Temp: 00DE Temp_avg: 00DA CO2: 0005 CO2_avg: 0001 d_CO2: 0000 CH4: 0048 CH4_avg: 004A d_CH4:
 →0000 H2: 00E1 H2_avg: 00DC d_H2: 0004
Temp: 00E3 Temp_avg: 00DD CO2: 0004 CO2_avg: 0001 d_CO2: 0000 CH4: 004A CH4_avg: 004A d_CH4:
 →0000 H2: 00EA H2_avg: 00E0 d_H2: 0003
Temp: 00E5 Temp_avg: 00DE CO2: 009D CO2_avg: 0015 d_CO2: 0013 CH4: 005E CH4_avg: 004C d_CH4:
 →0002 H2: 00DD H2_avg: 00E1 d_H2: 0000
Temp: 00E3 Temp_avg: 00E0 CO2: 0096 CO2_avg: 002A d_CO2: 0014 CH4: 0059 CH4_avg: 004F d_CH4:
 →0003 H2: 00C9 H2_avg: 00DE d_H2: FFFE
Temp: 00DF Temp_avg: 00E1 CO2: 0098 CO2_avg: 003F d_CO2: 0015 CH4: 0053 CH4_avg: 0051 d_CH4:
 →0002 H2: 00BD H2_avg: 00D9 d_H2: FFFC
Temp: 00DA Temp_avg: 00E0 CO2: 0095 CO2_avg: 0055 d_CO2: 0015 CH4: 0051 CH4_avg: 0053 d_CH4:
 →0001 H2: 00BA H2_avg: 00D4 d_H2: FFFB
Temp: 00E5 Temp_avg: 00E1 CO2: 008E CO2_avg: 0069 d_CO2: 0014 CH4: 0052 CH4_avg: 0055 d_CH4:
 →0001 H2: 00C6 H2_avg: 00D0 d_H2: FFFC
Temp: 00E6 Temp_avg: 00E1 CO2: 008B CO2_avg: 007C d_CO2: 0013 CH4: 0051 CH4_avg: 0053 d_CH4:
 →FFFF H2: 00D6 H2_avg: 00CE d_H2: FFFF
Temp: 00E3 Temp_avg: 00E1 CO2: 0086 CO2_avg: 008F d_CO2: 0012 CH4: 0053 CH4_avg: 0052 d_CH4:
 →FFFF H2: 00D5 H2_avg: 00CB d_H2: FFFE
Temp: 00E1 Temp_avg: 00E1 CO2: 007D CO2_avg: 008D d_CO2: FFFF CH4: 0050 CH4_avg: 0051 d_CH4:
 →0000 H2: 00D0 H2_avg: 00C9 d_H2: FFFF
Temp: 00DF Temp_avg: 00E2 CO2: 007A CO2_avg: 008A d_CO2: FFFD CH4: 0052 CH4_avg: 0051 d_CH4:
 →0000 H2: 00C5 H2_avg: 00C8 d_H2: 0000
Temp: 00E9 Temp_avg: 00E3 CO2: 0076 CO2_avg: 0085 d_CO2: FFFC CH4: 0053 CH4_avg: 0051 d_CH4:
 →0000 H2: 00C5 H2_avg: 00C9 d_H2: 0000
Temp: 00E9 Temp_avg: 00E4 CO2: 0075 CO2_avg: 0080 d_CO2: FFFC CH4: 004D CH4_avg: 0051 d_CH4:
 →0000 H2: 00C7 H2_avg: 00CB d_H2: 0001
Temp: 00ED Temp_avg: 00E6 CO2: 006B CO2_avg: 007B d_CO2: FFFC CH4: 0050 CH4_avg: 0050 d_CH4:
 →0000 H2: 00C5 H2_avg: 00CB d_H2: 0000
Temp: 00EB Temp_avg: 00E8 CO2: 006A CO2_avg: 0077 d_CO2: FFFC CH4: 0052 CH4_avg: 0050 d_CH4:
 →0000 H2: 00C9 H2_avg: 00C9 d_H2: FFFF
Temp: 00EB Temp_avg: 00EA CO2: 0062 CO2_avg: 0072 d_CO2: FFFB CH4: 0052 CH4_avg: 0050 d_CH4:
 →0000 H2: 00CD H2_avg: 00C8 d_H2: FFFF
Temp: 00EE Temp_avg: 00EB CO2: 0065 CO2_avg: 006E d_CO2: FFFD CH4: 0052 CH4_avg: 0050 d_CH4:
 →0000 H2: 00CF H2_avg: 00C8 d_H2: 0000
Temp: 00ED Temp_avg: 00EC CO2: 0061 CO2_avg: 006A d_CO2: FFFD CH4: 0052 CH4_avg: 0051 d_CH4:
 →0000 H2: 00CF H2_avg: 00C9 d_H2: 0001
Temp: 00E8 Temp_avg: 00EB CO2: 0052 CO2_avg: 0065 d_CO2: FFFC CH4: 004D CH4_avg: 0051 d_CH4:
 →0000 H2: 00D1 H2_avg: 00CB d_H2: 0001
Temp: 00DE Temp_avg: 00E8 CO2: 004A CO2_avg: 005F d_CO2: FFFB CH4: 0053 CH4_avg: 0051 d_CH4:
 →0000 H2: 00D0 H2_avg: 00CC d_H2: 0001
Temp: 00E0 Temp_avg: 00E6 CO2: 0050 CO2_avg: 005B d_CO2: FFFC CH4: 0053 CH4_avg: 0051 d_CH4:
 →0000 H2: 00D2 H2_avg: 00CE d_H2: 0001
Temp: 00E0 Temp_avg: 00E3 CO2: 004F CO2_avg: 0057 d_CO2: FFFD CH4: 0056 CH4_avg: 0052 d_CH4:
 →0000 H2: 00CA H2_avg: 00CE d_H2: 0000
Temp: 00E9 Temp_avg: 00E3 CO2: 0042 CO2_avg: 0053 d_CO2: FFFC CH4: 004B CH4_avg: 0051 d_CH4:
 →FFFF H2: 00C5 H2_avg: 00CD d_H2: 0000
Temp: 00E9 Temp_avg: 00E3 CO2: 0043 CO2_avg: 004E d_CO2: FFFC CH4: 004D CH4_avg: 0050 d_CH4:
 →0000 H2: 00C1 H2_avg: 00CB d_H2: FFFF
Temp: 00E2 Temp_avg: 00E4 CO2: 003F CO2_avg: 0049 d_CO2: FFFC CH4: 0050 CH4_avg: 0050 d_CH4:
 →0000 H2: 00BB H2_avg: 00C9 d_H2: FFFE
Temp: 00E8 Temp_avg: 00E5 CO2: 0022 CO2_avg: 0043 d_CO2: FFFA CH4: 004B CH4_avg: 004E d_CH4:
 →FFFF H2: 00C9 H2_avg: 00C7 d_H2: FFFF
Temp: 00E8 Temp_avg: 00E7 CO2: 0008 CO2_avg: 003A d_CO2: FFF7 CH4: 0052 CH4_avg: 004D d_CH4:
 →0000 H2: 00DE H2_avg: 00C9 d_H2: 0001
Temp: 00EE Temp_avg: 00E8 CO2: 0000 CO2_avg: 002E d_CO2: FFF5 CH4: 004A CH4_avg: 004D d_CH4:
 →0000 H2: 00DA H2_avg: 00CA d_H2: 0001
Temp: 00EF Temp_avg: 00E9 CO2: 0000 CO2_avg: 0023 d_CO2: FFF5 CH4: 0049 CH4_avg: 004C d_CH4:
 →0000 H2: 00BD H2_avg: 00C9 d_H2: FFFF
Temp: 00F4 Temp_avg: 00ED CO2: 0000 CO2_avg: 0019 d_CO2: FFF7 CH4: 004A CH4_avg: 004B d_CH4:
 →FFFF H2: 00D0 H2_avg: 00CA d_H2: 0001
Temp: 00F5 Temp_avg: 00EF CO2: 0000 CO2_avg: 0010 d_CO2: FFF7 CH4: 0048 CH4_avg: 004B d_CH4:
 →0000 H2: 00E2 H2_avg: 00CE d_H2: 0004
Temp: 00EA Temp_avg: 00F0 CO2: 0001 CO2_avg: 0007 d_CO2: FFF8 CH4: 0045 CH4_avg: 0048 d_CH4:
 →FFFE H2: 00E2 H2_avg: 00D4 d_H2: 0005
Temp: 00F1 Temp_avg: 00F0 CO2: 0000 CO2_avg: 0002 d_CO2: FFFB CH4: 0047 CH4_avg: 0047 d_CH4:
 →0000 H2: 00DF H2_avg: 00D7 d_H2: 0003
Temp: 00F0 Temp_avg: 00F0 CO2: 0000 CO2_avg: 0000 d_CO2: FFFF CH4: 004C CH4_avg: 0048 d_CH4:
 →0000 H2: 00CF H2_avg: 00D6 d_H2: FFFF
Temp: 00EE Temp_avg: 00EF CO2: 0000 CO2_avg: 0000 d_CO2: 0000 CH4: 0050 CH4_avg: 0049 d_CH4:
 →0001 H2: 00CA H2_avg: 00D5 d_H2: FFFF
```

Listing C.2: Data for Trial 3; Pure H_2

```
Temp: 038F Temp_avg: 00B6 CO2: 0014 CO2_avg: 0003 d_CO2: 0003 CH4: 0050 CH4_avg: 4ED8 d_CH4:
 →4ED8 H2: 0032 H2_avg: 0000 d_H2: 0000
Temp: 00E3 Temp_avg: 00E3 CO2: 001D CO2_avg: 0008 d_CO2: 0004 CH4: 0050 CH4_avg: E294 d_CH4:
 →93BC H2: 0043 H2_avg: 0000 d_H2: 0000
Temp: 00DE Temp_avg: 0110 CO2: 001C CO2_avg: 000C d_CO2: 0004 CH4: 004F CH4_avg: FB41 d_CH4:
 →18AC H2: 004D H2_avg: 0000 d_H2: 0000
Temp: 00DB Temp_avg: 013B CO2: 0019 CO2_avg: 0010 d_CO2: 0004 CH4: 0050 CH4_avg: FF6B d_CH4:
 →042A H2: 004B H2_avg: 0000 d_H2: 0000
Temp: 00DF Temp_avg: 0168 CO2: 0025 CO2_avg: 0016 d_CO2: 0005 CH4: 0079 CH4_avg: 0030 d_CH4:
 →00C5 H2: 0271 H2_avg: 0000 d_H2: 0000
Temp: 00E1 Temp_avg: 00DF CO2: 0049 CO2_avg: 0021 d_CO2: 000B CH4: 011D CH4_avg: 0073 d_CH4:
 →0043 H2: 0376 H2_avg: 0000 d_H2: 0000
Temp: 00E9 Temp_avg: 00E0 CO2: 0034 CO2_avg: 0027 d_CO2: 0006 CH4: 00E1 CH4_avg: 0096 d_CH4:
 →0023 H2: 033E H2_avg: 0000 d_H2: 0000
Temp: 00E6 Temp_avg: 00E2 CO2: 001A CO2_avg: 0028 d_CO2: 0000 CH4: 00AD CH4_avg: 00AC d_CH4:
 →0015 H2: 02EE H2_avg: 0000 d_H2: 0000
Temp: 00E2 Temp_avg: 00E3 CO2: 0018 CO2_avg: 0027 d_CO2: 0000 CH4: 0096 CH4_avg: 00BB d_CH4:
 →000F H2: 0276 H2_avg: 0000 d_H2: 0000
Temp: 00E9 Temp_avg: 00E5 CO2: 0015 CO2_avg: 0026 d_CO2: 0000 CH4: 0074 CH4_avg: 00BD d_CH4:
 →0001 H2: 01E7 H2_avg: 0000 d_H2: 0000
Temp: 00E9 Temp_avg: 00E7 CO2: 000B CO2_avg: 0023 d_CO2: FFFD CH4: 006A CH4_avg: 009F d_CH4:
 →FFE3 H2: 0137 H2_avg: 0000 d_H2: 0000
Temp: 00E9 Temp_avg: 00E7 CO2: 000E CO2_avg: 001A d_CO2: FFF8 CH4: 005D CH4_avg: 0084 d_CH4:
 →FFE6 H2: 00AD H2_avg: 0000 d_H2: 0000
Temp: 00E4 Temp_avg: 00E6 CO2: 000D CO2_avg: 0013 d_CO2: FFFA CH4: 0055 CH4_avg: 0071 d_CH4:
 →FFED H2: 006D H2_avg: 0000 d_H2: 0000
Temp: 00E9 Temp_avg: 00E8 CO2: 0002 CO2_avg: 000E d_CO2: FFFC CH4: 0059 CH4_avg: 0064 d_CH4:
 →FFF3 H2: 004A H2_avg: 0000 d_H2: 0000
Temp: 00E7 Temp_avg: 00E7 CO2: 000A CO2_avg: 000C d_CO2: FFFE CH4: 005A CH4_avg: 005D d_CH4:
 →FFFA H2: 0047 H2_avg: 0000 d_H2: 0000
Temp: 00E6 Temp_avg: 00E7 CO2: 0003 CO2_avg: 0009 d_CO2: FFFE CH4: 005C CH4_avg: 005A d_CH4:
 →FFFD H2: 003E H2_avg: F700 d_H2: 0000
Temp: 00EB Temp_avg: 00E7 CO2: 0000 CO2_avg: 0007 d_CO2: FFFF CH4: 005A CH4_avg: 0059 d_CH4:
 →FFFF H2: 0036 H2_avg: 9120 d_H2: 9A00
Temp: 00E8 Temp_avg: 00E8 CO2: 0001 CO2_avg: 0005 d_CO2: FFFE CH4: 005C CH4_avg: 005A d_CH4:
 →0000 H2: 0031 H2_avg: F068 d_H2: 5F40
Temp: 00EA Temp_avg: 00E8 CO2: 0006 CO2_avg: 0003 d_CO2: FFFF CH4: 0053 CH4_avg: 0059 d_CH4:
 →0000 H2: 003D H2_avg: 4720 d_H2: 56B8
Temp: 00E3 Temp_avg: 00E7 CO2: 000B CO2_avg: 0004 d_CO2: 0001 CH4: 0054 CH4_avg: 0058 d_CH4:
 →FFFF H2: 0048 H2_avg: C139 d_H2: 7A19
Temp: 00ED Temp_avg: 00E9 CO2: 000B CO2_avg: 0005 d_CO2: 0000 CH4: 0058 CH4_avg: 0057 d_CH4:
 →0000 H2: 0042 H2_avg: 4060 d_H2: 7F28
Temp: 00EF Temp_avg: 00E9 CO2: 000C CO2_avg: 0006 d_CO2: 0001 CH4: 0054 CH4_avg: 0056 d_CH4:
 →FFFF H2: 003D H2_avg: 771D d_H2: 36BD
Temp: 00EC Temp_avg: 00EA CO2: 0006 CO2_avg: 0007 d_CO2: 0001 CH4: 0053 CH4_avg: 0054 d_CH4:
 →FFFF H2: 0039 H2_avg: 1138 d_H2: 9A1C
Temp: 00EC Temp_avg: 00EB CO2: 0002 CO2_avg: 0007 d_CO2: 0000 CH4: 0054 CH4_avg: 0054 d_CH4:
 →0000 H2: 0039 H2_avg: 02AB d_H2: F173
Temp: 00F1 Temp_avg: 00ED CO2: 0006 CO2_avg: 0007 d_CO2: 0000 CH4: 0053 CH4_avg: 0054 d_CH4:
 →0000 H2: 003D H2_avg: 0097 d_H2: FDEC
```

Listing C.3: Data for Trial 4; Town Gas

```
Temp: 00F8 Temp_avg: 00F5 CO2: 0000 CO2_avg: 0001 d_CO2: FFFF CH4: 0049 CH4_avg: 0051 d_CH4:
 →FFFE H2: 003E H2_avg: 003A d_H2: 0000
Temp: 00FA Temp_avg: 00F7 CO2: 0000 CO2_avg: 0001 d_CO2: 0000 CH4: 0049 CH4_avg: 004E d_CH4:
 →FFFE H2: 0037 H2_avg: 0039 d_H2: 0000
Temp: 00FF Temp_avg: 00F8 CO2: 0001 CO2_avg: 0000 d_CO2: 0000 CH4: 004D CH4_avg: 004C d_CH4:
 →FFFF H2: 0042 H2_avg: 0039 d_H2: 0000
Temp: 00F6 Temp_avg: 00F8 CO2: 0000 CO2_avg: 0000 d_CO2: 0000 CH4: 0049 CH4_avg: 004B d_CH4:
 →FFFF H2: 003D H2_avg: 003A d_H2: 0000
Temp: 00FA Temp_avg: 00F9 CO2: 0000 CO2_avg: 0000 d_CO2: 0000 CH4: 0052 CH4_avg: 004B d_CH4:
 →0000 H2: 0044 H2_avg: 003B d_H2: 0000
Temp: 00F6 Temp_avg: 00F9 CO2: 0000 CO2_avg: 0000 d_CO2: 0000 CH4: 0069 CH4_avg: 0050 d_CH4:
 →0005 H2: 0044 H2_avg: 003E d_H2: 0003
Temp: 0103 Temp_avg: 00FB CO2: 0006 CO2_avg: 0001 d_CO2: 0000 CH4: 024C CH4_avg: 00A7 d_CH4:
 →0056 H2: 024A H2_avg: 008A d_H2: 004B
Temp: 0106 Temp_avg: 00FC CO2: 000A CO2_avg: 0002 d_CO2: 0001 CH4: 0243 CH4_avg: 0109 d_CH4:
 →0062 H2: 0215 H2_avg: 00D9 d_H2: 004F
Temp: 0108 Temp_avg: 0100 CO2: 0016 CO2_avg: 0005 d_CO2: 0003 CH4: 0234 CH4_avg: 016B d_CH4:
 →0062 H2: 01EB H2_avg: 0121 d_H2: 0048
Temp: 00F9 Temp_avg: 0100 CO2: 0020 CO2_avg: 000A d_CO2: 0005 CH4: 0226 CH4_avg: 01CA d_CH4:
 →005E H2: 01D5 H2_avg: 0165 d_H2: 0044
Temp: 00FC Temp_avg: 0101 CO2: 0026 CO2_avg: 0010 d_CO2: 0006 CH4: 0212 CH4_avg: 0220 d_CH4:
 →0056 H2: 01C5 H2_avg: 01A6 d_H2: 0040
Temp: 00FF Temp_avg: 0100 CO2: 002A CO2_avg: 0017 d_CO2: 0006 CH4: 0200 CH4_avg: 0222 d_CH4:
 →0001 H2: 01B1 H2_avg: 01E3 d_H2: 003D
Temp: 00FB Temp_avg: 00FE CO2: 0026 CO2_avg: 001D d_CO2: 0005 CH4: 01F1 CH4_avg: 0215 d_CH4:
 →FFF3 H2: 01A4 H2_avg: 01D4 d_H2: FFF2
Temp: 0103 Temp_avg: 00FD CO2: 0022 CO2_avg: 0021 d_CO2: 0004 CH4: 01D9 CH4_avg: 0203 d_CH4:
 →FFEF H2: 0192 H2_avg: 01C0 d_H2: FFEC
Temp: 0105 Temp_avg: 00FF CO2: 0029 CO2_avg: 0024 d_CO2: 0003 CH4: 01C5 CH4_avg: 01F0 d_CH4:
 →FFED H2: 018A H2_avg: 01AF d_H2: FFF0
Temp: 00FE Temp_avg: 0100 CO2: 0025 CO2_avg: 0026 d_CO2: 0001 CH4: 01AD CH4_avg: 01DC d_CH4:
 →FFEC H2: 016E H2_avg: 019E d_H2: FFEF
Temp: 0104 Temp_avg: 0101 CO2: 0022 CO2_avg: 0025 d_CO2: 0000 CH4: 019F CH4_avg: 01C9 d_CH4:
 →FFED H2: 015C H2_avg: 018C d_H2: FFEF
Temp: 0105 Temp_avg: 0103 CO2: 0025 CO2_avg: 0024 d_CO2: 0000 CH4: 0190 CH4_avg: 01B5 d_CH4:
 →FFED H2: 0151 H2_avg: 017C d_H2: FFF0
```

```
Temp: 0106 Temp_avg: 0103 CO2: 0024 CO2_avg: 0024 d_CO2: 0000 CH4: 017C CH4_avg: 01A3 d_CH4:
 →FFEE H2: 0144 H2_avg: 016C d_H2: FFF0
Temp: 0105 Temp_avg: 0103 CO2: 0025 CO2_avg: 0024 d_CO2: 0000 CH4: 0162 CH4_avg: 018F d_CH4:
 →FFED H2: 0139 H2_avg: 015D d_H2: FFF1
Temp: 0102 Temp_avg: 0104 CO2: 001E CO2_avg: 0023 d_CO2: FFFF CH4: 0157 CH4_avg: 017D d_CH4:
 →FFEF H2: 0127 H2_avg: 014D d_H2: FFF0
Temp: 0100 Temp_avg: 0103 CO2: 0014 CO2_avg: 0020 d_CO2: FFFE CH4: 0140 CH4_avg: 016B d_CH4:
 →FFEE H2: 0114 H2_avg: 013E d_H2: FFF1
Temp: 0106 Temp_avg: 0103 CO2: 0013 CO2_avg: 001E d_CO2: FFFE CH4: 012F CH4_avg: 0157 d_CH4:
 →FFED H2: 0101 H2_avg: 012E d_H2: FFF1
Temp: 0106 Temp_avg: 0103 CO2: 0016 CO2_avg: 001B d_CO2: FFFE CH4: 012A CH4_avg: 0146 d_CH4:
 →FFF0 H2: 0106 H2_avg: 0121 d_H2: FFF4
Temp: 0109 Temp_avg: 0104 CO2: 001D CO2_avg: 001A d_CO2: FFFF CH4: 0111 CH4_avg: 0136 d_CH4:
 →FFF0 H2: 00FA H2_avg: 0115 d_H2: FFF4
Temp: 010B Temp_avg: 0106 CO2: 0015 CO2_avg: 0017 d_CO2: FFFE CH4: 0102 CH4_avg: 0125 d_CH4:
 →FFF0 H2: 00E2 H2_avg: 0107 d_H2: FFF2
Temp: 010A Temp_avg: 0108 CO2: 0010 CO2_avg: 0015 d_CO2: FFFE CH4: 00F0 CH4_avg: 0115 d_CH4:
 →FFF0 H2: 00CA H2_avg: 00F8 d_H2: FFF1
Temp: 0108 Temp_avg: 0108 CO2: 0007 CO2_avg: 0013 d_CO2: FFFE CH4: 00E1 CH4_avg: 0105 d_CH4:
 →FFF1 H2: 00B5 H2_avg: 00E8 d_H2: FFF1
Temp: 010B Temp_avg: 0109 CO2: 0009 CO2_avg: 0011 d_CO2: FFFF CH4: 00D5 CH4_avg: 00F5 d_CH4:
 →FFF0 H2: 00A5 H2_avg: 00D8 d_H2: FFF1
Temp: 010B Temp_avg: 010A CO2: 0003 CO2_avg: 000E d_CO2: FFFE CH4: 00C6 CH4_avg: 00E5 d_CH4:
 →FFF1 H2: 009C H2_avg: 00C7 d_H2: FFEF
Temp: 010A Temp_avg: 010A CO2: 0009 CO2_avg: 000B d_CO2: FFFD CH4: 00BC CH4_avg: 00D7 d_CH4:
 →FFF2 H2: 0094 H2_avg: 00B6 d_H2: FFEF
Temp: 010C Temp_avg: 010A CO2: 0010 CO2_avg: 000A d_CO2: FFFF CH4: 00B5 CH4_avg: 00CB d_CH4:
 →FFF4 H2: 0092 H2_avg: 00A8 d_H2: FFF3
Temp: 0105 Temp_avg: 0109 CO2: 0000 CO2_avg: 0007 d_CO2: FFFE CH4: 005B CH4_avg: 00B3 d_CH4:
 →FFE8 H2: 0094 H2_avg: 009E d_H2: FFF7
Temp: 010F Temp_avg: 010A CO2: 0000 CO2_avg: 0006 d_CO2: FFFF CH4: 0045 CH4_avg: 0097 d_CH4:
 →FFE4 H2: 005B H2_avg: 0090 d_H2: FFF2
Temp: 0108 Temp_avg: 010A CO2: 0000 CO2_avg: 0004 d_CO2: FFFF CH4: 0045 CH4_avg: 007C d_CH4:
 →FFE6 H2: 0057 H2_avg: 0083 d_H2: FFF3
Temp: 010E Temp_avg: 010A CO2: 0000 CO2_avg: 0004 d_CO2: 0000 CH4: 0045 CH4_avg: 0064 d_CH4:
 →FFE8 H2: 0054 H2_avg: 0077 d_H2: FFF4
Temp: 0111 Temp_avg: 010B CO2: 0000 CO2_avg: 0002 d_CO2: FFFF CH4: 0047 CH4_avg: 004E d_CH4:
 →FFEA H2: 0055 H2_avg: 006C d_H2: FFF6
```