Design and Fabrication of a Microheater Control System

University of Utah Senior Project

Michael William Chambers
Electrical and Computer Engineering
University of Utah
Salt Lake City, USA
www.ece.utah.edu/~mchamber
mike.chambers@utah.edu

***Abstract*—Microsensors are becoming increasingly important to society as the field of nanotechnology advances. One such microsensor is a solid state gas concentration sensor, which has been used for over a decade, especially in the automotive field to control air/fuel ratios in the combustion process. These devices generally utilize an on-board microheater to improve gas sensitivity. Such a gas sensor is currently being developed here at the University of Utah to detect nitrous oxide (NOx) concentrations in diesel exhaust. My contribution was to develop the controller required to regulate the high temperature set points (up to 650°C) of the microheater. Feedback for this control is provided by an on-chip thin film Resistance Temperature Detector (RTD) located near the microheater. Using this control setup, the system is capable of maintaining a constant set temperature for gas sensing, as well as delivering step temperature profiles needed for testing, tuning and diagnostic purposes.**

Keywords-Microheater; Gas Sensor; PID Control; Data Converters

I. INTRODUCTION

Solid state gas concentration sensors have been used extensively in the automotive industry for over 10 years. These sensors provide feedback to the fuel injector controls to constantly maintain the most efficient ratios of air to fuel, thereby increasing the most efficient combustion. Simply put, this type of sensor utilizes semiconductor properties of surface adsorption to detect changes in resistance as a function of varying concentrations of different gases. In order to detect these resistive changes and equate them to changing (and static) gas concentrations, the gas temperature must be held constant. A microheater (or an array of microheaters) near the gas sensor is controlled to maintain this temperature and to account for the convective cooling caused by gas flow. Fig. 1 below shows an example of a microheater. This picture is one of three microheaters in an array, the combination of which reportedly exhibits excellent temperature homogeneity [1].

A disadvantage of most current gas sensors is that they are rated at relatively low temperatures (up to about 300ºC). Thus, they are located downstream from the combustion chambers far enough to allow sufficient cooling of the exhaust gases. However, high operation temperatures are needed for optimum performance and accuracy.

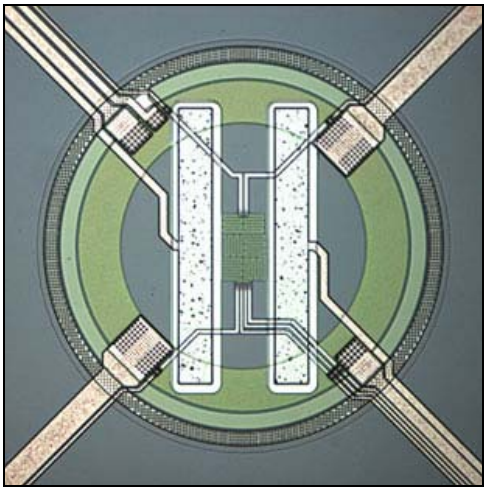

Figure 1. This figure displays an example of a microheater. The green rings are the heaters, the central array is the temperature sensor, and the central speckled bars are the gas sensors. The gas sensors are approximately 50μm wide, or half the width of a human hair.

A disadvantage of most current gas sensors is that they are rated at relatively low temperatures (up to about 300ºC). Thus, they are located downstream from the combustion chambers far enough to allow sufficient cooling of the exhaust gases. However, high operation temperatures are needed for optimum performance and accuracy.

Mike Sorenson, Srinivasan Kannan, and Xiaoxin Chen, under the supervision of Dr. Florian Solzbacher and Dr. Loren Rieth, are currently working on a solid state gas sensor that will be capable of operating at temperatures of over 600ºC. This will allow the sensor to be placed directly outside the exhaust manifold to provide superior accuracy. The design is currently in the last stages of fabrication and consists of interdigitated metal-oxide fingers serving as the gas sensor, with the heater encapsulating these fingers and a platinum RTD temperature sensor located near the heater. The design is shown below in Fig. 2.

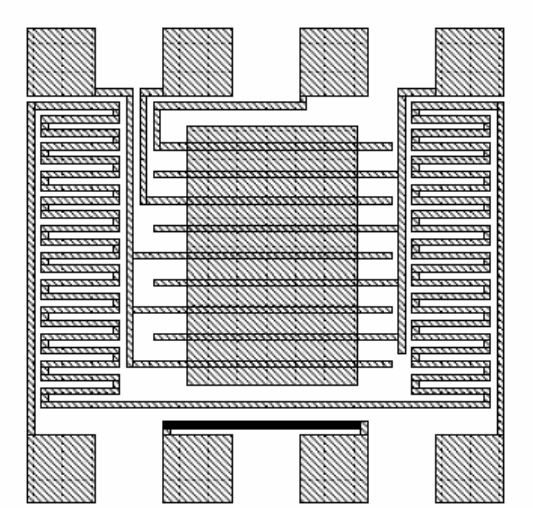

Figure 2. Solid-state gas concentration sensor currently being developed at the University of Utah. The sensor is located in the middle, with the microheater on either side of it. The RTD temperature sensor is located at the bottom center.

II. Specifications

Specifications for controlling the microheater included:

- Max temperature of 700°C ± 2°C accuracy (max temp is 650°C but need to account for overshoot)
- 1W max power delivered to ~150Ω (room temp) microheater
- Response time limited by sensor, not my device
- Estimated as 1ms
- Be able to provide step temperature profiles
- Voltages cannot exceed automotive supply grid (12V)
- Feedback provided by on-board RTD
- Voltage drop across RTD is proportional to temperature
- Needs to be controlled with analog signal
- Digital on/off may decrease life of heater
- Temperature set points communicated by PC
- RS232 interface

III. Control Theory

Rapid, accurate control of the microheater can be accomplished using a *proportional-integral-derivative* (PID) controller. A basic block diagram of this controller is displayed in Fig. 3 below [2].

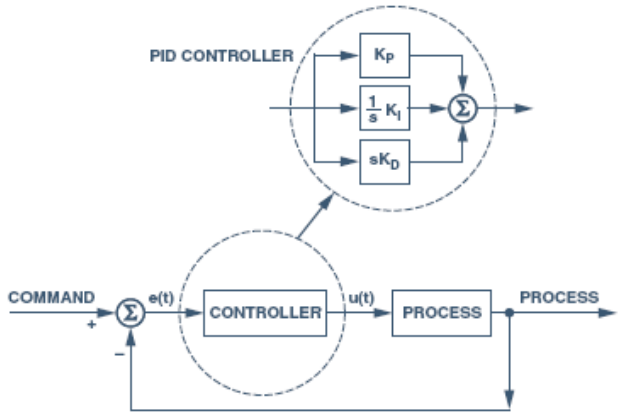

Figure 3. A PID controller sums the 3 terms derived from the error signal e(t) to provide very accurate and stable control of a process.

For a very thorough description of a PID controller, please refer to one of the many articles on the web [2]. For temperature control, the idea of the control is that the temperature is compared with a desired setpoint temperature. The difference, or error signal *e(t),* is applied to the controller, which sums three terms derived from *e(t)* to produce the control signal *u(t).* The proportional term K_p applies a corrective term proportional to the error; the integral term K_i seeks to hold its average input at zero; and the derivative term K_d improves stability, reduces overshot caused by high K_i and K_p terms and improves response time by anticipating changes in error. *u(t)* manipulates a physical input to the process, thereby causing a change in the regulated temperature that will stably reduce the error. To control the temperature of the microheater, the system must accurately measure the current temperature of the microheater and adjust the input power accordingly.

IV. Design

A. PID Realization

The PID controller has traditionally been implemented using discrete analog components. However, in order to obtain response times on the order of 1ms, as well as to perform signal processing on the resistance of the RTD and metal-oxide layer, it was decided to implement the PID control algorithm on a microcontroller (mcu). This also allows simple interfacing with a PC to communicate the desired temperature profiles.

In order to implement the PID controller using a mcu, the continuous time PID equation obtained from Fig. 3 had to first be transformed into a discrete time equation. This process is shown in Fig. 4 below [3]. Notice that starting with the traditional continuous time PID equation, discrete, programmable equations can be obtained.

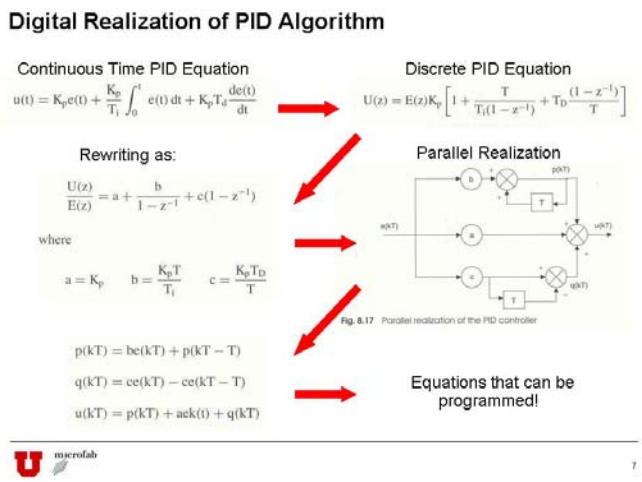

Figure 4. Rewriting the traditional continuous time PID equation in the s-domain and taking the z-transform yields the discrete PID equation. After rewriting this equation and drawing the equivalent parallel realization, equations can be obtained by which to program the PID controller. These equations use previous output variables in current calculations.

The block diagram for the digital implementation of the PID controller is shown in Fig. 5 below [4]. This configuration places the derivative and proportional terms in the feedback of the controller, thereby eliminating the problem known as "derivative kick." With a maximum and minimum output allowed, the problem known as "integral windup."

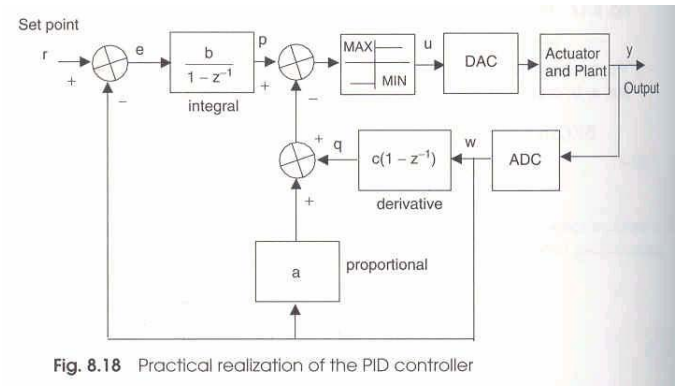

Figure 5. Realizing the PID controller in this configuration minimizes "integral windup" and "derivative kick."

B. *Component Selection*

Now that the PID control setup was taken care of, it was time to design the system. Obtaining the instantaneous temperature is accomplished by measuring the resistance of the RTD. This is done by sampling the voltage across it while forcing a constant current through it. For this, an analog-to-digital converter (ADC) was needed. Measuring the voltage drop across the heater itself is a way of obtaining further information about the temperature of the device. An Analog Devices AD7655 was chosen for this sampling. This 16-bit ADC has dual channels which convert at an astounding 1MSPS (mega sample per second, or 2 channels every 2μs). The known current being forced through the RTD, along with the 16-bit digital word corresponding to the voltage drop across the RTD, is used to calculate the resistance of the RTD. As was mentioned above, this calculation is performed by a mcu.

Two options existed for this mcu. The first was to use a fast (>100MHz) mcu. These fast mcus typically have little or no on-board FLASH, but floating point libraries could have been used to calculate the PID algorithm and the RTD resistance. This option would have made the coding much simpler but would have required a more complex board layout to minimize noise and to interface with FLASH memory.

Using a slower mcu with sufficient integrated FLASH memory seemed like a more viable option for the project. The coding was more complex because the mcu had to be programmed using integer math only, however, the layout and interfacing with the data converters was much simpler. In the end, the Philips LPC2129 32-bit mcu was chosen. This mcu has an operating frequency of 60MHz and 256kB on-board FLASH and 16kB on-board RAM memory, as well as sufficient I/O pins and user peripherals.

Because the heater had to be driven with an analog control, it was obvious that a digital-to-analog converter (DAC) was needed. The Analog Devices AD768 16-bit was chosen because of its rapid conversion time (up to 30MSPS) and the option for current output. The 16-bit data converters both have parallel outputs, which interface nicely with the 32-bit mcu.

A basic block diagram for this project showing the present control setup is shown in Fig. 6 below. Parts used in this design are included in Appendix A of this report.

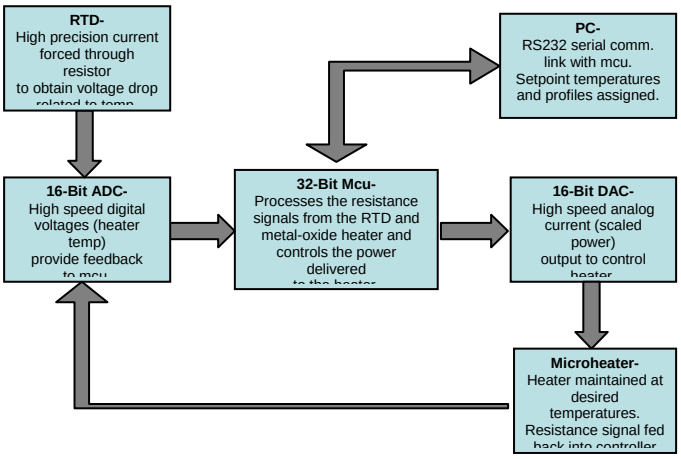

Figure 6. Block diagram of project. The resistance of the RTD corresponds to the temperature of the microheater. The voltage drop across it is converted to a 16-bit word and fed to the mcu to calculate the resistance and perform the PID algorithm based on the difference between the desired setpoint and the actual temperature. The mcu communicates with a computer to relay the desired temperature profile, as well as to relay resistance data. Based upon the setpoint temperature and the error signal, the mcu exports a digital 16-bit word to control the microheater. This signal is converted to an analog signal and

Special thanks to Mike Sorenson, Dr. Florian Solzbacher, and Dr. Ken Stevens for all of their help in this project.

fed to the microheater. The resistance of the heater is calculated using the second channel of the ADC and sent to the mcu for processing.

C. Implementation

The first task to implement the design was to characterize the heater. Because the microheater being developed at the U was not fabricated yet at the time of this publication, a commercial gas sensor was obtained. The sensor of choice was the Figaro TGS 2201 gasoline/diesel gas sensor, which operates at temperatures around 300°C. Because this sensor did not have an on-board RTD, the design had to be slightly modified to obtain the resistance of the heater. This was accomplished by adding a 1Ω precision reference resistor in series with the heater. By sampling the voltage across this resistor, V_{Ref}, the current could be calculated using Ohm's law, which, because the resistance is 1Ω, is equal to V_{Ref}. Knowing the current through the reference resistor (and the heater), the resistance of the heater, R_{Heater}, could be obtained after sampling the voltage across it, V_{Heater}. Thus, by sampling only V_{Ref} and V_{Ref}, the resistance of the heater can be obtained, as shown in (1) below.

$$R_{Heater} = \frac{V_{Heater}}{I_{Heater}} = \frac{V_{Heater}}{\left(\frac{V_{Ref}}{R_{Ref}}\right)} = \frac{V_{Heater}}{V_{Ref}} \quad (1)$$

In order to correlate R_{Heater} with the temperature, set voltages were applied across the heater and reference resistor. Using an infrared (IR) camera, the corresponding temperatures were measured given applied voltages. An example of these measurements is shown in Fig. 7 below.

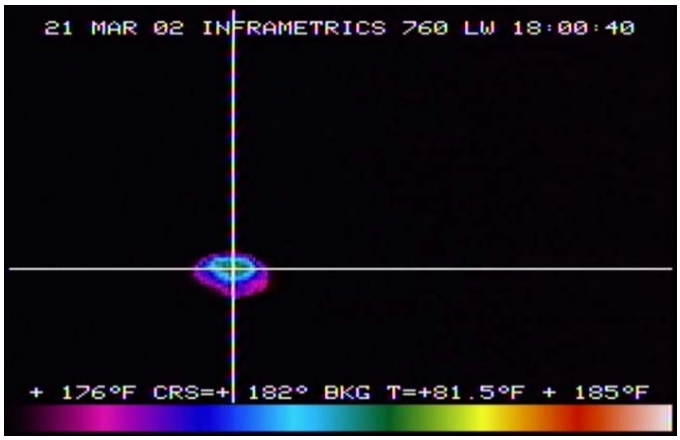

Figure 7. IR picture of temperature of microheater with applied voltage of 5V.

A table was made of the derived R_{Heater} vs. temperature values. These values were fit to a curve in Matlab to obtain the linear equation relating the two variables. The results are graphed in Fig. 8.

The next task was to program the mcu. This was done in the C programming language. Fig. 9 shows the basic structure of the code. Refer to Appendix B for a complete listing of the code.

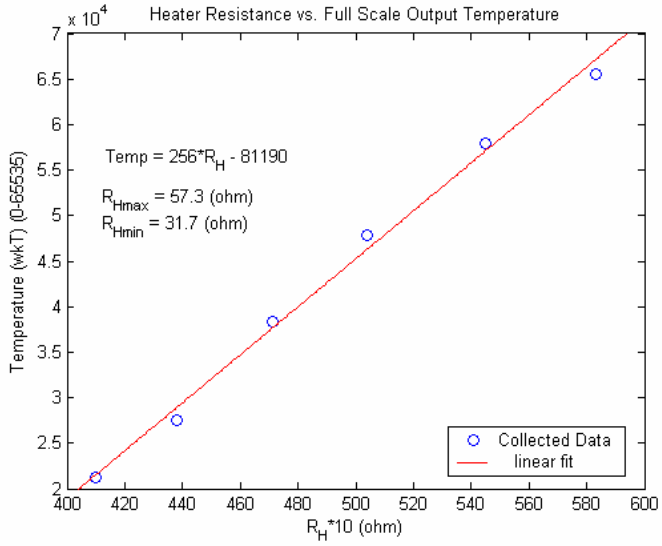

Figure 8. Graph of measured and fit data. The linear equation correlates the resistance of the heater with the temperature. The resistance is multiplied by 10 for ease in using integer math.

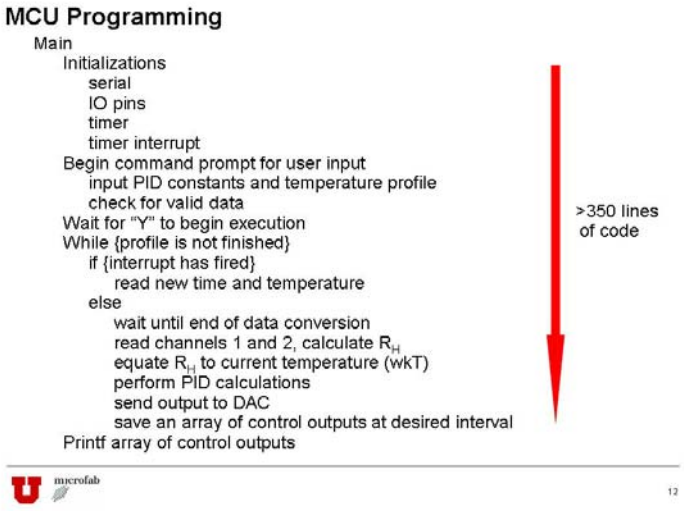

Figure 9. Description of the C code programmed on mcu.

The mcu was programmed using an RS232 serial connection with a PC. Loading the PID coefficients, as well as the temperature profile, was performed using HyperTerminal in Windows. A screenshot of this program is shown in Fig. 10 below. Future plans include designing a GUI in LabVIEW to load these values into the mcu. This GUI is shown in Fig. 11.

The next step was to set up the data converters. This was accomplished incrementally by first hooking the output of the ADC directly to the DAC. Fig. 12 below shows a sine wave input to the ADC and the corresponding output from the DAC.

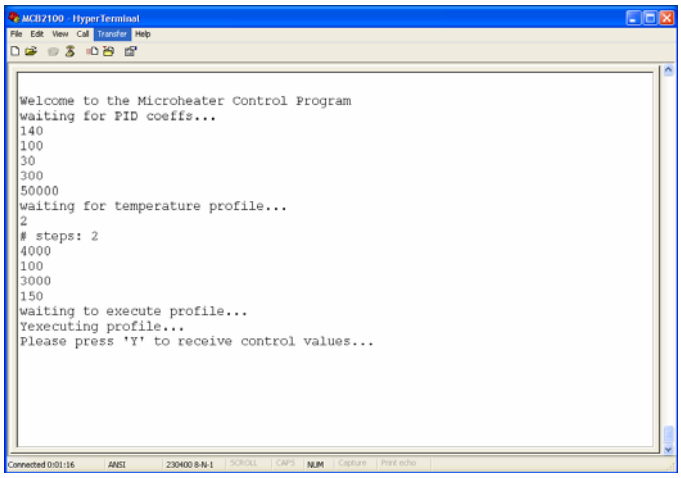

Figure 10. Loading PID coefficients, as well as temperature profile, is done with HyperTerminal.

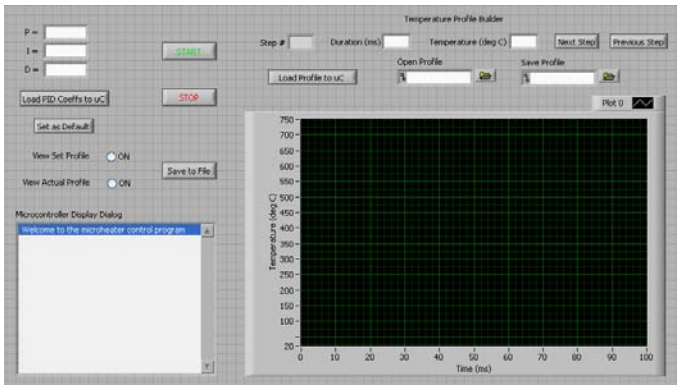

Figure 11. A LabVIEW GUI will be implemented to replace HyperTerminal.

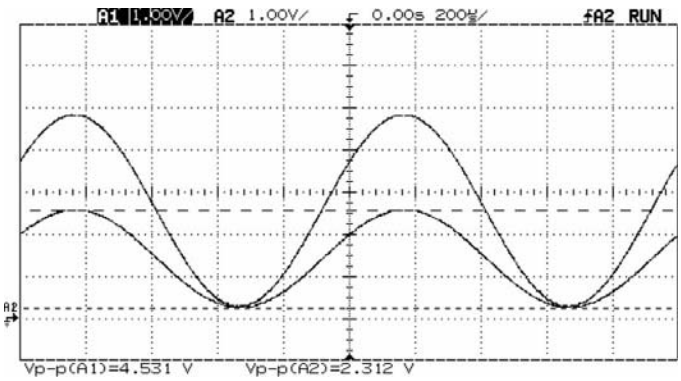

Figure 12. Input wave into ADC and output from DAC.

As can be seen in the above graph, the analog-to-digital conversions, as well as the digital-to-analog conversions, work perfectly. The amplitude of the DAC output can be scaled to the desired output range, determined by resistor R_2 shown in Fig. 14. This provides excellent design flexibility.

The same test was performed again, except this time reading the ouput from the ADC into the mcu and then exporting the corresponding digital word to the DAC. As expected, the results were very similar.

Before hooking everything up, the PID loop was tested to see if it could execute in the time required by the ADC to convert 2 channels. The loop only executes when the BUSY signal from the ADC is low, meaning that the conversions are finished. The bottom graph in Fig. 13 below shows the BUSY signal from the ADC, and the top graph shows the loop iteration frequency. As can be seen, the loop does not execute in less than 2µs. With some efficiency improvements, the execution time can be reduced to below 2µs, thus utilizing the full speed of the ADC.

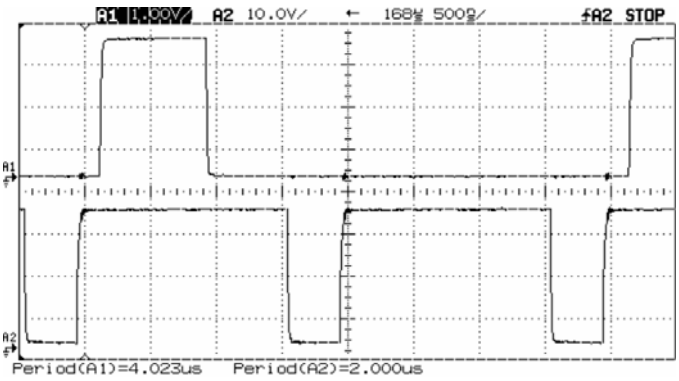

Figure 13. Graphs of PID iteration frequency (top plot) and ADC BUSY signal. Note that conversions are finished when the BUSY signal goes low. Only then can the PID loop be executed, thus ensuring uniform sampling.

Fig. 14 below shows a simplified schematic of the system. Channels 1 and 2 are multiplexed in time in the ADC. The ADC feeds pins 10 through 25 of Port 0 (P0.10..25) on the mcu. The channel read into the mcu is controlled with P0.6. The mcu transmits the computed control output to the DAC from P1.16..31. The 16-bit control word converted to current between 0-20 mA, and the corresponding output voltage is controlled by R_2.

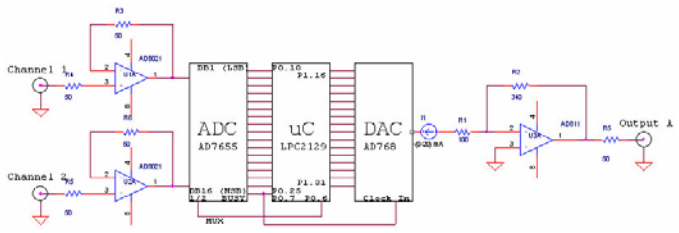

Figure 14. Simplified schematic of control setup.

Fig. 15 below shows a simplified schematic of how the system was integrated with the heater and reference resistor to complete the feedback and control loop. A photograph of the actual setup in the lab is shown in Fig. 16.

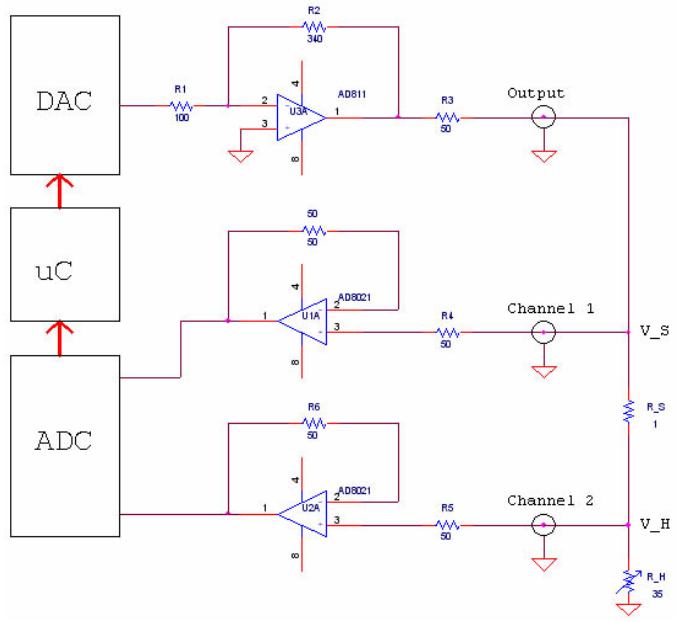

Figure 15. Simplified schematic of system implementation. The voltage read from channel 2 is subtracted from channel 1 to obtain VRS.

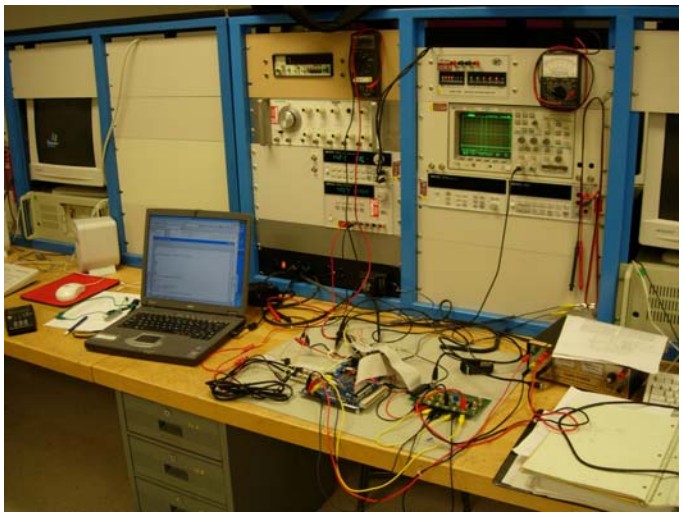

Figure 16. Actual lab control setup.

V. Results

Output current of the op amp from the DAC (AD811) is 100mA. It was thought that if the temperature was regulated at a temperature below 80mA (@4V ~115°C), that this setup would work. Instead the output would swing rail-to-rail. Upon disconnecting the DAC output and applying an external voltage to the heater and reference resistor, the mcu computes the correct R_{Heater}. The conclusion was that the output needed to be buffered.

Using a similar configuration as Fig. 15, except adding the BJT at the output of the op amp provided the schematic shown in Fig. 17. The op amp is a Q2N2222 NPN transistor with a beta value of 180. It has a current output of 500mA, which is sufficient to drive the Figaro sensor.

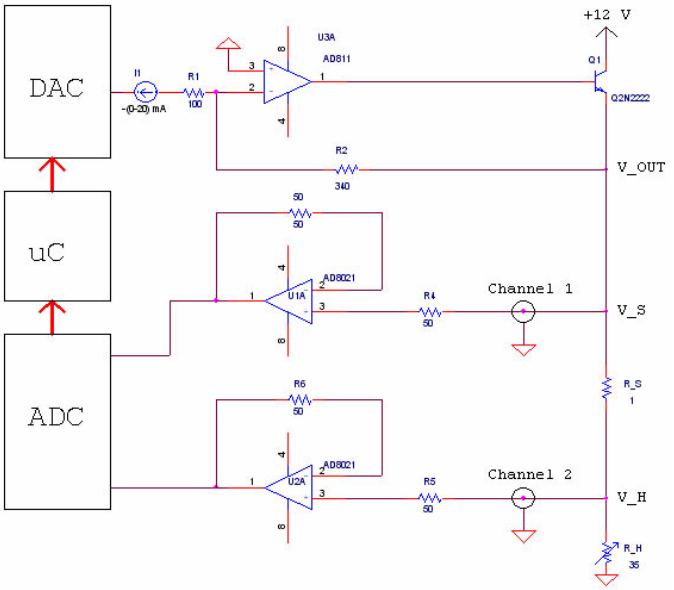

Figure 17. Simplified schematic of system implementation using current buffer.

This setup should have worked correctly; however, the output still rails and R_{Heater} is not computed correctly with the full-scale output (FSO) from the DAC. Upon further tweaking, the problem will be resolved. At the time of this publication, it is assumed that something is wrong with the current buffer. An alternate buffer will be designed and implemented to fix it. Working components of this design include:

- Correct ADC and DAC conversions
- ADC to mcu and mcu to DAC interfacing
- PC to mcu interfacing
- User input
- Error handling on invalid user input
- Sampling using BUSY signal from ADC
- Timer match to time step durations
- Timer interrupt to input new temperature and time duration
- Temperature profile advancing at each time step
- I/O from mcu
- Data logging on mcu
- MUX selection via P0.6
- Correct computation of R_{Heater} (given external voltage applied)
- mcu operation
- JTAG interfacing and debugging
- RS232 communication

VI. Conclusion

PID control is a relatively easy, straightforward approach to use when controlling a process, enabling it to be used in a variety of modern control applications. A microcontroller based approach to this control offered the benefits of easy computer interfacing, fast response time, in-situ data processing and logging, as well as a lot of learning. The list of working components outweighs the list of non-working, but the controller will be operational ASAP. In addition to the LabVIEW GUI still to be written, other future plans include housing a power supply and the evaluation boards into an aluminum project box.

Acknowledgment

M.W.C. thanks his wife, who has been a single mom for the last month, Mike Sorenson for all the times he helped and offered his valuable insights. Also, special thanks to his advisor, Dr. Stevens for his knowledgeable advice and his mentor, Dr. Solzbacher for his patience and scheduling. Dr. Harrison was also very willing to offer his help and advice. Thanks should also be extended to Brian Flint for his help with timer interrupts and providing the author with free lab parts.

M.W.C. also wishes to thank Analog Devices and Keil Software for providing evaluation boards free of charge.

References

[1] M. Graf, D. Barrettino, P. Kaeser, J. Cerda, A. Heirlemann, and H. Baltes, "Smart single-chip CMOS microhotplate array for metal-oxide-based gas sensors", Transducers (2003), 123.

[2] E. Neary, "Mixed-signal control circuits use mcu for flexibility in implementing PID algorithms". Analog Dialogue 38-01, January (2004)

[3] D. Ibrahim, Microcontroller Based Temperature Monitoring and Control, Woburn, MA: Newnes, 2002, pp. 194–195.

[4] D. Ibrahim, Microcontroller Based Temperature Monitoring and Control, Woburn, MA: Newnes, 2002, p. 196.

Appendix A – Supplies Purchased for Project

Date	Item	Vendor	Price
10/10/2005	Keil MCB2100 Eval Board	Keil	$0
10/21/2005	6V Power Supply	UofU Bookstore	$7.45
10/22/2005	Connectors and Header Pins	RaElco Electronics	$10.15
11/10/2005	ADC & DAC Eval Boards	Analog Devices	$0
1/31/2006	BNC (F) to SMB (F) (2)	Ebay	$15.00
1/31/2006	USB to RS232 Connector	Ebay	$8.04
3/15/2006	BNC (F) to Alligator Clips (3)	Ebay	$14.25
3/30/2006	Aluminum Project Box	Ebay	$10.52
4/3/2006	Triple Output Power Supply	Ebay	$21.45
4/3/2006	RS232 Splitter Cable	Ebay	$6.98
3/21/2006	ADC & DAC Eval Boards	Analog Devices	$346.32
4/13/2006	Wire, Switches, Connectors	RaElco Electronics	$19.90
4/17/2006	Connectors and Header Pins	RaElco Electronics	$7.32
4/15/2006	Misc. Electronic Parts	RaElco Electronics	$10.92
4/16/2006	Nuts and Bolts	Home Depot	$3.14
Total			$481.44

APPENDIX B – C CODE OF MCU

```
#include <stdio.h>                              // prototype declarations for I/O functions
#include <math.h>                               // prototype declarations for math functions
#include <stdlib.h>                             // standard library declarations
#include <LPC21xx.H>                            // LPC21xx definitions for LPC2129 mcu
#define MAXBUFFERSIZE    8                      // input cannot exceed this many characters
#define         wait                10000000    // wait before accepting transmissions again
#define         num_steps           25          // max number of profile steps
#define         sample_save_rate    25000       // 250 for 1 ms at 4 us sample rate -> = 5 for 20 us
//#define       max_samples         50000       // at 256 kB of memory, this is number of 4-byte samples available
                                                // AFTER calculating what is required for the program about 50
                                                // seconds for 1 ms samples or 1 second at 20 us sampling

/* Global Variable Declarations */
int interrupt = 0;                      //interrupt=0:interrupt hasn't fired yet--interrupt=1:interrupt fired
int time = 10;                          //initialize time variable

/****************/
/* main program    */
/****************/
int main (void)  {                                      // execution starts here
      //while (1) {                                     //can change this to interrupt and exit w/ INT1
            /* Function Declarations */
            void init_serial (void);                    // initialize the serial interface
            void init_io (void);                        // initialize the GPIO pins
            void init_timer (void);                     // initialize timer0
            void T0isr(void) __fiq;                     // initialize timer0 interrupt

            /* Variable Declarations */
            unsigned int MAX_out,MIN_out,T_Max,rkT,wkT,control[80];
            unsigned int a,b,c,profile[50],r,n,num_samples,loop,icr,V_s,V_h;
            int transmit1,go,delay;
            long ekT,pkT,qkT,pkT_1,ekT_1,cnt,R_h,wkT_l,ukT;
            float temp,profile_f[50],pid[5];
            char d;

            /*Buffer variables*/
            char    ch;                                 //handles user input
            char    buffer[MAXBUFFERSIZE];              // sufficient to handle one line
            int     char_count;                         // number of characters read for this line
            int     valid_choice,exit_flag;

            /* Initializations */
            init_serial();                              //initialize serial interface
            init_io();                                  //initialize GPIO pins
            //IOSET0 = 0x40;                            //turn off (off=high, on=low) ADC (P0.6 hooked to PD
                                                        //pin on ADC)
            IOCLR0 = 0x40;                              //turn on channel 2 first (V_h) (P0.6 hooked to OB pin on
                                                        //ADC)
            /* Local Variable Definitions */
            pkT_1=0;
            ekT_1=0;
            T_Max=200.0;//750.0;                        //max temp of heater being used
            exit_flag=0;                                //exit PID loop
            go=0;                                       //flag to proceed
```

```
icr=0;                                          //counter for profile and printf
loop=0;                                         //counter for number of loops between saves
num_samples=0;                                  //counter for control[] index incramenting

/************************/
/* Start of Program loop */
/************************/
printf("\nWelcome to the Microheater Control Program\n");

/*****************************************************************************/
/* Get Buffered PID parameters when 'Load PID params' is pushed in LabVIEW   */
/*****************************************************************************/
printf("waiting for PID coeffs...\n");
//future need to make an interrupt to bring me here when the "load PID" button pushed
/*Get buffered transmission and ensure validity*/
for (r=0;r<5;r++) {
        valid_choice = 0;

        while( valid_choice == 0 ) {
                ch = getchar();
                char_count = 0;

                while( (ch != '\n')  &&  (char_count < MAXBUFFERSIZE)) {
                        buffer[char_count++] = ch;
                        ch = getchar();
                }

                buffer[char_count] = 0x00;      /* null terminate buffer */
                //printf("buffer: %s\n",buffer);
                pid[r] = atof( buffer );

                if( (pid[r] <= 0) || (pid[r] > 65535) ) {

                        if (buffer[0] == '0' ) valid_choice = 1;     //allow input of '0'

                        else {
                                printf("invalid parameters.  Please reload coeffs.\n");
                                r=0;                            //reset loop counter to load complete new array

                                for (cnt = 0; cnt < wait; cnt++);       // Delay to avoid last transmissions to
                                                                        //count in new array
                        }
                }

                else
                        valid_choice = 1;                               //user entered valid data, proceed
        }
}


//convert this array into individual integers components to use in calculations
a=pid[0];                       /*P Coefficient (can shift if want to)*/
b=pid[1];                       /*I Coefficient (can shift if want to)*/
c=pid[2];                       /*D Coefficient (can shift if want to)*/
MIN_out=pid[3];         /*Min digital word sent to DAC (corresponds to min power output)*/
MAX_out=pid[4];                 /*Max digital word sent to DAC (corresponds to max power output)*/
```

```
/*****************************************************************************/
/* Get Buffered temperature profile when 'Load Profile' is pushed in LabVIEW */
/*****************************************************************************/
valid_choice = 0;

while( valid_choice == 0 ) {
        printf("waiting for temperature profile...\n");
        transmit1=0;
        ch = getchar();
        char_count = 0;

        while( (ch != '\n')  &&  (char_count < MAXBUFFERSIZE)) {
                buffer[char_count++] = ch;
                ch = getchar();
        }

        buffer[char_count] = 0x00;      /* null terminate buffer */
        n = atof( buffer );

        if( (n <= 0) || (n > 25) ) {
                //put in option to have constant temperature rather than steps
                printf("invalid # steps.  Please reload profile\n");
                r=0;                            //reset loop counter to load complete new arry

                for (cnt = 0; cnt < wait; cnt++);       // Delay to avoid last transmissions to count in new
                                                        //array
        }

        else  {
                printf("# steps: %d\n",n);
                /*Import n element array*/

                for (r=0;(r<2*n) && (transmit1 == 0);r++) {
                        valid_choice = 0;

                        while( valid_choice == 0 ) {
                                ch = getchar();
                                char_count = 0;

                                while( (ch != '\n')  &&  (char_count < MAXBUFFERSIZE)) {
                                        buffer[char_count++] = ch;
                                        ch = getchar();
                                }

                                buffer[char_count] = 0x00;      /* null terminate buffer */
                                //printf("buffer: %s\n",buffer);
                                profile_f[r] = atof( buffer );          //to do float math, need all float
                                                                        //elements
                                profile[r] = atof( buffer );            //This is just to create the size of array

                                if( (profile_f[r] <= 0) || (profile_f[r] > 65535) ) {

                                        printf("invalid parameters.  Please reload profile\n");

                                        for (cnt = 0; cnt < wait; cnt++);       //Delay to avoid last
                                                                                //transmissions to count
                                                                                //in new array
```

```
                                        transmit1=1;
                                        break;
                                }

                                else
                                        //check here for correct time & temperature inputs
                                        valid_choice = 1;
                        }
                }
        }
}

/*Convert float Times w/ 1 decimal precision to integers*/
for (r=0; r<n; r++) {
        temp = profile_f[2*r]*10;                       //convert time (Ex 10.2 ms -> 102)
        profile[2*r]=temp;
}

/*Convert float Temperatures to integers*/
for (r=0; r<n; r++) {
        temp = profile_f[2*r+1]/T_Max;                  //convert Temps to between 0-65535
        profile[2*r+1]=temp*65535;
}

/*******************************************************/
/* Begin exectution when 'START' is pushed in LabVIEW */
/*******************************************************/
printf("waiting to execute profile...\n");
/*When START is pressed in LabVIEW, it should transmit a 'Y'*/
while (go == 0) {
        scanf("%c",&d);

        if(d == 'Y')  go = 1;

        else printf("Please press Start in LabVIEW (or 'Y')...\n");
}

/*******************************/
/* Execute Temperature Profile */
/*******************************/
printf("executing profile...\n");

IOCLR1 = 0xFFFF0000;                    //clear all output on P0 so don't get DAC conversion until ready
//IOSET0 = 0x40;                        //turn on ADC (P0.6 hooked to PD pin on ADC)

time  = profile[0]*20;                  //read in first time duration --mult by 20 to get correct ms time
rkT = profile[1];                       //read in first setpoint temp.
init_timer();

while(exit_flag == 0) {                 //blink for element arrays

        if (interrupt != 1) {           //on interrupt, read in new time and Temp

        /*PID Algorithm w/ Ts = 2 us (500 kHz)*/  //-> will take 2 cycles (4 us) to complete algorithm

                while ((IOPIN0 & (1<<7)) != 0);         /* Hold until end of A/D Conversion */
                V_h = (IOPIN0 >> 10);                   //Voltage accross heater, currently between 0-
```

```
                                        //65535
IOSET0 = 0x40;                          //turn on ADC channel 1 (P0.6)

__asm { NOP; }                          //wait at least 40 ns to read channel 2 (4 clock
                                        //cycles at 84 MHz)
__asm { NOP; }
__asm { NOP; }
__asm { NOP; }

V_s = (IOPIN0 >> 10);                   /* Voltage accross R_s & R_h*/

R_h = (V_h*10)/(V_s-V_h);               //V_s not taken differentially --R_s = 1 ohm

//if (R_h<317) R_h=317;
if (R_h>573) R_h=573;

//equate RTD voltage drop to temperature
wkT_l = 256*R_h - 81190;                //linear fit to Temp vs R_h*10 curve
wkT=wkT_l;

/*Calculate Error*/
ekT=rkT-wkT;

/*Calculate I term*/
pkT=b*ekT+pkT_1;

/*Calculate D term*/
qkT=c*(ekT-ekT_1);

IOCLR0 = 0x40;                          //turn ADC channel 2 back on (P0.5)

/*Calculate PID output*/
ukT=pkT+a*ekT+qkT;

/*Protect against integral windup*/
if (ukT > MAX_out) {
        pkT=pkT_1;
        ukT=MAX_out;
}
else if (ukT < MIN_out) {
        pkT=pkT_1;
        ukT=MIN_out;
}

/*Save Variables*/
pkT_1=pkT;
ekT_1=ekT;

/*Send control to DAC*/
IOPIN1 = (ukT << 16);

/*Save array of control values*/
loop++;
if ((loop==sample_save_rate) && (num_samples<80))  {
        control[num_samples] = ukT;
        num_samples++;
        loop=0;
```

```
                        }

                        for (delay=0; delay<5; delay++) {}           //ensure get to 4 us sampling
                }

                else {
                        /*Read in new profile values*/
                        if (icr < n-1) {
                                icr++;
                                time  = (profile[2*icr])*20;
                                rkT = profile[2*icr + 1];            //current temp setpoint
                                interrupt = 0;
                        }

                        /*Exit PID loop*/
                        else {
                                IOCLR1 = 0xFFFF0000;                 //turn off output to DAC
                                exit_flag=1;
                                T0TCR = 0x00000000;                  //disable timer
                        }
                }
        }
        go=0;
        printf("Please press 'Y' to receive control values...\n");

        /*Wait for user input*/
        while (go == 0) {
                scanf("%c",&d);
                if(d == 'Y')  go = 1;

                else printf("\nPlease press 'Y'...\n");
        }

        /*Print out samples*/
        for (icr=0; icr<num_samples; icr++)          {
                printf("%d\n",control[icr]);
        }
    //}
    return 0;
}

/****************/
/* Functions    */
/****************/
void init_serial (void) {                    /* Initialize Serial Interface */
   PINSEL0 = 0x00050000;                     /* Enable RxD1 and TxD1 */
      U1LCR = 0x83;                          /* 8 bits, no Parity, 1 Stop bit */
      U1DLL = 23;                            /* 230400 Baud Rate @ 84MHz VPB Clock */  //(divisor = Pclk/16/baud)
      U1LCR = 0x03;                          /* DLAB = 0 */
}

void init_io (void) {                        /* Initialize IO Pins */
      IODIR1 = 0xFFFF0000;                   /* P1.16..31 defined as DAC Outputs (<< 16)*/
      IODIR0 = ~(0x3FFFC00);                 /* P0.10..25 defined as ADC Inputs  (>> 10)*/
      IODIR0 &= ~(1 << 7);                   /* P0.7 defined as BUSY signal from ADC*/
      IODIR0 |= (1<<6);                      /* P0.6 defined as channel selection output to ADC*/
```

```
	//IODIR0 |= (1<<6);			/*P0.6 defined as PD ADC Outputs*/
	/*Pin Configurations*/
	//P1.16..31 - 16 bit DAC Output
	//P0.15..30 - 16 bit ADC Input
	//P0.5 - MUX selection output signal (1=high channel A (V_S) 0=low channel B (V_H))
	//P0.6 - PD signal (1=high power down 0=low convert)
	//P0.7 - Busy input signal from ADC		(1=high busy 0=low finished converting)
	//pins P0.0,1,8,9 are used for UART and P0.14 is External interrupt1
}

void init_timer (void) {
	PINSEL0 |= 0x00000800;			//Match1 as output
	T0PR = 0x000001A4;			//Load prescaler for 10 usec tick
	T0TCR = 0x00000002;			//Reset counter and prescaler
	T0MCR = 0x00000003;			//On match reset the counter and generate an interrupt
	T0MR0  = time;				//Set the cycle time
	T0MR1  = 0x00000000;			// Set duty cycle to zero
	T0EMR = 0x00000042;			//On match clear MAT1
	T0TCR = 0x00000001;			//enable timer

	VICVectAddr4 = (unsigned)T0isr;		//Set the timer ISR vector address
	VICVectCntl4 = 0x00000024;		//Set channel
	VICIntEnable |= 0x00000010;		//Enable the interrupt
}

void T0isr (void) __fiq {
	interrupt = 1;
	T0EMR |= 0x00000002;			//Set MAT1 high for begining of the cycle
	T0MR1++;				//Increment PWM Duty cycle
	T0MR1  = T0MR1&0x000000FF;		//Limit duty cycle
	T0IR |= 0x00000001;			//Clear match 0 interrupt
	VICVectAddr = 0x00000000;		//Dummy write to signal end of interrupt
}
```