Heated stage system for small rodents – Zeiss inverted microscope

1. System description:

This system is intended to be used with ‘single-port’ window chambers, imaged using a Zeiss inverted microscope and is intended to maintain a suitably warm environment around the window chamber and the lower part of the animal, while the rest of the animal’s environment is controlled by an enclosure around the microscope. The system described here consists of the following:

A stage insert which fits an existing motorised Märzhäuser inverted stage and which contains two plate heaters and a temperature sensor, as well as a clamping arrangement to hold a coaxial anaesthetic gas delivery and waste tube.

An electronic temperature controller which delivers an appropriate quantity of electrical power to the plate heaters so as to maintain the user-set surface temperature; this unit is fitted with a USB interface which may be used in conjunction with an optional software package.

A wall plug power supply, delivering +12V at a maximum current of ~1 A, supplying raw power to the system.

An optional software application which may be used to display the time course of temperature settling and variations, as well as setting relevant control parameters

The hardware parts of the system are shown in Figure 1.

The use of the system is (hopefully!) fairly intuitive. The user should first adjust the four levelling screws at the corners of the insert so that the insert is level (minimising focus variations with stage movement) and so that it fits snugly in the stage. The wall plug power supply is then connected and the appropriate target temperature set; this is done by pressing down the switch next to the temperature indicator and turning the ‘set’ control (clockwise = higher temperature) until the appropriate reading is obtained. The range of that control is limited to around 38 degrees so as to prevent possible overheating. The temperature display is capable of displaying temperatures up to around 43 degrees. Details of the controls are shown in Figure 2. The system is able to operate independently of any software, but an optional USB interface is available should the user wish to log the performance.
The mechanics of the system are shown in Figure 3. Two heating pads are coupled to a copper plate to ensure an even temperature distribution and good thermal contact to a solid-state temperature sensor placed in contact with the copper plate. An anodised aluminium plate is thermally bonded to the copper plate and is used to prevent inevitable potential corrosion.

Figure 2: Details of controls on the system temperature control unit

Figure 3: Details of stage insert construction
2. Circuit description:

The circuit is based around a DLP245 PB-G module which has a USB interface and a 16F877A microprocessor. The 16F877A has a number of analogue inputs which are used to implement a proportional-integral-derivative (PID) control loop, driving the heaters with a pulse-width modulated (PWM) output. The circuit is shown in Figure 4.

An LM62 temperature sensor provides an output scaled at 15.6 mV/°C\(^{-1}\) and offset by around 480 mV. An LM4040 reference is used, in conjunction with a LT1013 operational amplifier to provide a DC output such that 20 °C = +2V and 40 °C = +4V. This output is attenuated to provide 20mV-40mV signal which is fed to a digital panel meter of 200 mV sensitivity. An indication of actual temperature, with a 0.1 °C resolution is thus obtained. The non-attenuated temperature signal is also fed to the 16F877A’s analogue-to-digital converters (A-Ds). The reference also drives the set-point control potentiometer, the output of which is fed to one of the other A-Ds; these two signals drive the PID loop. The A-Ds full-scale range is set by additional inputs driven from signal ground and the reference voltage, nominally 4.096 V. The loop output is a PWM signal which drives ground-referenced heaters through two MOSFETs, 2N7000 driver (n-channel) and IRF 5210 output switch (p-channel). A maximum heater power of 7.5W is available, though the maximum on-time of the PWM output may be further restricted by an internal signal derived from an additional analogue input, defined by a preset.

![Circuit diagram of the temperature controller.](image)

Figure 4: Circuit diagram of the temperature controller.

The repetition rate of the PWM signal is made slow (~1.22 kHz) to simplify MOSFET drive requirements. The firmware code on the 16F877A microcontroller is based around a PID control feedback loop. The proportional, integral and derivative gains are stored in the microcontroller’s internal EEPROM. After initialising I/O pins connected with the USB communication to the PC and setting up the PWM unit, A/D converter circuits and timer interrupt, the program enters a control loop where the
microcontroller checks for any USB communication until a timed interrupt allows the PID control to start. The PID timer interrupt is set to 20 ms.

Three voltages are read on the A/D converter channels which relate to actual temperature, set point temperature and maximum set power to the heaters. These are read successively 64 times and the readings are added together. The totals are then divided by 64 to give an average reading for each of the channels. The program then enters the PID control where the actual temperature is compared to the set point temperature. The integral, derivative and proportional terms are calculated and with suitable scaling, a number is derived at which can be entered into the PWM control. This sets the amount of power delivered to the heater mats in the stage insert. The maximum power reading limits the upper range of the PWM control.

3. PIC firmware:

The firmware stored in the microcontroller is shown below. We note that we acquire analogue data on three channels (representing the set-point, the actual temperature and the maximum heater power). The analogue values are referenced to an internal reference and the PIC analogue-to-digital converters can operate with respect to this reference and signal ground (supplied to RA3 and RA2 respectively). The analogue-to-digital converters take 20 µs to complete a conversion and require a delay time of 20 µs between conversions when changing channels, so when digitising three inputs, the total time taken is 120 µs. Since the converters operate with 10 bit resolution, we acquire 64 such sets of values and end up with 16 bit numbers with simple addition (going above 16 bits would have required significantly more complex code and a slower overall response time. We thus spend 120 µs x 64 = 7.680 ms acquiring one block of data. We provide an interrupt with a 20 ms period which triggers this 7.68 ms acquisition sequence. We thus end up with a useful reduction of potential AC mains-induced interference.

We also note that we use a software-defined proportional integral-derivative feedback loop (see for example [http://www.jashaw.com/pid/tutorial/](http://www.jashaw.com/pid/tutorial/)). However implementing this digitally requires the application of various scaling factors in order to keep variables with the 16 bit integer dynamic range. The proportional, integral and derivative gains are set from the high level C-code program, described later, so as to be able to adjust them conveniently and obtain minimal overshoot and fast settling. We modelled our code on the AN964A Microchip application note ([http://www.microchip.com/](http://www.microchip.com/)). Heater power is determined by using the PIC PWM routines.

```c
//************************************************************
// File: heater mat control.c 
// Date: 29/10/10 
//************************************************************
#include "heater mat thermostat.h"
/* 16F877A bytes */
/* Change it per chip */
#byte PIC_SSPBUF=0x13
#byte PIC_SSPADD=0x93
#byte PIC_SSPSTAT=0x94
#byte PIC_SSPCON1=0x14
#byte PIC_TMR1L=0x0E
#byte PIC_TMR1H=0x0F
#byte PIC_INTCON=0x0B
#byte PIC_PWM1CON=0x1C
#byte PIC_CCP1CON=0x17
#byte PIC_CCPR1L=0x15
#byte PIC_EECON1=0x18C
#byte PIC_OPTION=0x81
/* Bit defines */
#define PIC_SSPSTAT_BIT_SMP 0x80
#define PIC_SSPSTAT_BIT_CKE 0x40
#define PIC_SSPSTAT_BIT_DA 0x20
#define PIC_SSPSTAT_BIT_P 0x10
#define PIC_SSPSTAT_BIT_I 0x08
#define PIC_SSPSTAT_BIT_RW 0x04
#define PIC_SSPSTAT_BIT_UA 0x02
#define PIC_SSPSTAT_BIT_OE 0x01
#define PIC_SSPCON1_BIT_WCOL 0x80
#define PIC_SSPCON1_BIT_OE 0x80
```

Heated stage system for small rodents.doc 4
Heated stage system for small rodents.doc
derivative_term = 120;

if(derivative_term <= -120)
    derivative_term = -120;

if(derivative_term >= 120)
    derivative_term = 120;

Cn = en0 + integral_term + derivative_term;
    //Sum the terms
Cn = ((Cn * kp)/512);
    //Multiply by kp then scale
Cn = 1023;
    //Used to limit duty cycle
if(Cn <= -1023)
    Cn = -1023;

if(Cn > 0)
    duty_cycle_zero=0;
set_pwm1_duty(duty_cycle_zero);
    //Function needs 16 bit integer
    
if(Cn <= 0)
    Cn = abs(Cn);
    
set_pwm1_duty(Cn);
    
en5 = en4;
en4 = en3;
en3 = en2;
en2 = en1;
en1 = en0;
en0 = 0;

return;

************************************************************************
unsigned int16 ADConversions(chanNum)
{
    set_adc_channel(chanNum);
    delay_us(20);
    return read_adc();
}
************************************************************************
void DisableEEPROMWrite(void)
{
    PIC_EECON1=PIC_EECON1 & 0xfb;
    //EEPROM write enable bit=0 inhibits write
}
************************************************************************
void EnableEEPROMWrite(void)
{
    PIC_EECON1=PIC_EECON1 | 0x04;
    //EEPROM write enable bit=1 enables write
}
************************************************************************
//Set constants
************************************************************************
void Set_Constants()
{
    kp=read_eeprom(1);
    kp_L=read_eeprom(2);
    kp=(kp<<8) | kp_L;
    ki=read_eeprom(3);
    ki_L=read_eeprom(4);
    ki=(ki<<8) | ki_L;
    kd=read_eeprom(5);
    kd_L=read_eeprom(6);
    kd=(kd<<8) | kd_L;

    //Calculate derivative term using (Td/Ts)(E(n) - E(n-1))
    //where Td is Kd/Kp
    //actual equation used is Kd(en0-en3)/(Kp*X*3*Ts)
    derivative_term = derivative_term * kd;
    //where X is an unknown scaling factor
}
************************************************************************
int dataTest()
{
    int state;

    state=input_state(RXF);
    if(state==0 & busy==0)
        return 0;
    }

int getData(void)
int command;
output_bit(RD,0); //RD pin low
command=input_D(); //Read the first byte this is command
output_bit(RD,1); //RD pin high

//go to dig.IO, AD_read to handle the command
switch(command)
{
  case 1: //PID values
    handle_PID_val();
    break;
  case 2:
    break;
  case 3: //Store PID values in memory
    WriteEeprom();
    ack_return();
    break;
  case 4: //Read PID values
    handle_PID_read();
    ack_return();
    break;
  case 5: //Read A/D inputs
    handle_A_D_inputs();
    ack_return();
    break;
  case 6:
    break;
}
return 0;

int handle_PID_val(void)
{
  int state;
  int x;

  output_bit(RD,1); //RD pin high
  state=input_state(RXF); //Tests if data ready
  while(state==1)
  {
    state=input_state(RXF); //Tests if data ready
    output_bit(RD,0); //RD pin low
    numBytes=input_D(); //Read the number of data bytes
    output_bit(RD,1);
    for(x=0;x<=(numBytes-3);x++)
    {
      state=input_state(RXF); //Tests if data ready
      while(state==1)
      {
        state=input_state(RXF); //Tests if data ready
      }
      output_bit(RD,0); //RD pin low
      i_O_inbytes[x]=input_D(); //Store in array
      output_bit(RD,1); //RD pin high
    }
  }
  state=input_state(TXE); //Sends acknowledge at end of sequence
  while(state==1)
  {
    state=input_state(TXE); //Tests if module ready
    output_D(0x04); //Send acknowledge
    output_bit(WR,1); //WR Pin high to low to actually write
    output_bit(WR,0);
  }

  return 0;
}

void WriteEeprom()
{
  kp M = kp>>8;
  kp L = kp & 0xff;
  ki M = ki>>8;
  ki L = ki & 0xff;
  kd M = kd>>8;
  kd L = kd & 0xff;
  SetTempVal M = temp_set_point>>8;
  SetTempVal L = temp_set_point & 0xff;
  EnableEPROMWrite();
  write_eeprom(1,kp M);
  write_eeprom(2,kp L);
  write_eeprom(3,ki M);
  write_eeprom(4,ki L);
  write_eeprom(5,kd M);
  write_eeprom(6,kd L);
  write_eeprom(7,SetTempVal M);
  write_eeprom(8,SetTempVal L);
}
write_eeprom(4,ki_L);
write_eeprom(5,kd_M);
write_eeprom(6,kd_L);
write_eeprom(7,SetTempVal_M);
write_eeprom(8,SetTempVal_L);

DisableEEPROMWrite();

//‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐
int handle_PID_read(void)
{
    kp_M=read_eeprom(1);
    //Read PID values from memory
    kp_L=read_eeprom(2);
    ki_M=read_eeprom(3);
    ki_L=read_eeprom(4);
    kd_M=read_eeprom(5);
    kd_L=read_eeprom(6);

    writeData(Kp_M);
    writeData(Kp_L);
    writeData(Ki_M);
    writeData(Ki_L);
    writeData(Kd_M);
    writeData(Kd_L);

    return 0;
}

//‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐
int handle_A_D_inputs(void)
{
    unsigned int en0_M,en0_L,max_power_M,
    max_power_L,temp_set_point_M,temp_set_point_L;

    en0_M = measured_temp>>8;
    en0_L = measured_temp & 0xff;

    max_power_M = max_power>>8;
    max_power_L = max_power & 0xff;

    temp_set_point_M = temp_set_point>>8;
    temp_set_point_L = temp_set_point & 0xff;

    writeData(en0_M);
    writeData(en0_L);
    writeData(max_power_M);
    writeData(max_power_L);
    writeData(temp_set_point_M);
    writeData(temp_set_point_L);

    return 0;
}

//‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐
void writeData(data)
//Write data back to PC
{
    int state=1;

    state=input_state(TXE);
    //Tests if module ready
    while(state==1){
        state=input_state(TXE);
        //Set TXE input
        output_D(data);
        output_bit(WR,1); //WR Pin high to low to actually write
        output_bit(WR,0); //WR low
    }

    //‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐
    int ask_return(void)
    {
        int state;

        state=input_state(TXE);
        //Sends acknowledge at end of sequence
        while(state==1){
            state=input_state(TXE);
            //Tests if module ready
            output_D(0x04);
            //Send acknowledge byte
            output_bit(WR,1); //WR Pin high to low to actually write
            output_bit(WR,0); //WR low

            return 0;
        }

    }

//*******************************
//Initialisation routine
//*******************************
void Init()
{
    int state;

    output_bit(RD,1);  //Pin B1 high
    output_bit(WR,0);  //Pin B2 low
    state=input_state(RXF);  //Make RXF input
    state=input_state(TXE);  //Make TXE input

    setup_timer_0(T0_INTERNAL | T0_DIV_1);
    //If div_1 need 391 interrupts or 400 with set_timer0(40)
    //setup_timer_0(T0_INTERNAL | T0_DIV_2);
    //Set timer0 prescalar and clock,need 200 interrupts set_timer0(26)
    set_timer0(40); //Use with T0_DIV_1 and 400 interrupts for 20ms
    setup_ccp1(CCP_PWM);  //Set for PWM output

    Heated stage system for small rodents.doc 8
Heated basic user panel. Twenty of these readings are stored in an array and then averaged and plotted on a

4. PIC configuration / internal fuse file

#include <16F877A.h> //Device file
#include adc-10 //10 bit ADC conversion
#include icd=TRUE //For debugging
#include H5 //High speed Osc (>4MHz)
#include NOPROTECT //Code not protected from reading
#include PUT //Power Up Timer
#include delay(clock=2000000) //Define clock speed for delay function

5. High level software description:

The software communicates with the control unit over a USB link. The set point and actual temperatures along with the maximum power setting are read at 50ms. The temperature readings are displayed on the basic user panel. Twenty of these readings are stored in an array and then averaged and plotted on a
A further control panel can be accessed by pressing a hidden button on the user panel. This gives the ability to set the PID parameters used in the controller, read the values stored in the microcontroller memory and read the value set for the maximum power available to the heaters. This panel is not designed to be user accessible, but is shown for completeness in the left part of Figure 5. The basic software panel accessible to the user is shown in the right portion of Figure 5.

We now list the high level code, developed under LabWindows CVI (http://www.ni.com/lwcvi/).

```
#include "cvixml.h"
#include <rs232.h>
#include <ansi_c.h>
#include <cvirte.h>
#include <userint.h>
#include "utility.h"
#include <analysis.h>
#include "DeviceFinder.h"
#include "heater mat ctrl_ui.h"
#include <IO_interface_v2.h>
#include <usbconverter_v2.h>
#include <DeviceManager.h>

static int PORT;
static int HtrMatCtrlPanel,ReadEepromPanel;
static int initI2Cport(void);
static int ManageDevice(char* device, int enable);
static void GCI_init_heater_mat_ctrl(void);
static void SendPIDValues(void);
static void ReadPIDValues(void);

#include "heater mat ctrl_ui.h"
#include <cvirte.h>
#include <userint.h>
#include "utility.h"
#include <analysis.h>
#include "DeviceFinder.h"
#include "heater mat ctrl_ui.h"
#include <IO_interface_v2.h>
#include <usbconverter_v2.h>
#include <DeviceManager.h>

int main (int argc, char *argv[])
{
    if (InitCVIRTE (0, argc, 0) == 0)
        return -1; /* out of memory */
    if ((HtrMatCtrlPanel = LoadPanel (0, "heater mat ctrl.ui.uir", HRTMATPNL)) < 0)
        return -1;
    if ((ReadEepromPanel = LoadPanel (0, "heater mat ctrl.ui.uir", READEEPROM)) < 0)
        return -1;
    if(initI2Cport()==-1)return 0; //Set port number

    DisplayPanel (HtrMatCtrlPanel);
    // RecallSettings();
    GCI_EnableLowLevelErrorReporting(1);
    Delay(1.0);
    GCI_init_heater_mat_ctrl();
    // SetCtrlAttribute (HtrMatCtrlPanel, HRTMATPNL_TIMER, ATTR_ENABLED, 1); //Enable timer
    RunUserInterface ();
    DiscardPanel (HtrMatCtrlPanel);
    GCI_close(2)(
        return 0;
    )

    static int getFTDIport(int *PORT)
    {
        char path[MAX_PATHNAME_LEN],ID[20];
        int id_panel,ui,ctrl;

        //If we are using an FTDI gizmo Device Finder will give us the port number
        GetProjectDir(path);
        strcat(path,"\\\");
        strcat(path, "heater mat ID.txt");
        return selectPortForDevice(path, PORT, "Select Port for heater mat");
    }

    Heated stage system for small rodents.doc
```
static int initI2Cport()
{
    int err,ans;
    char port_string[10];

    if (getFTDIport(PORT) == 0) 
        sprintf(port_string, "COM%d", PORT);
    else //Device not found or not using FTDI. Use some other mechanism //such as Glenn's COM port allocation module.
        while(getFTDIport(PORT) != 0)
        {
            ans=ConfirmPopup ("Comms error", "Try plugging USB cable in or do you want to quit?");
            if(ans==1) //quit
                return -1;
        }
        sprintf(port_string, "COM%d", PORT);
    err = OpenComConfig(PORT, port_string, 9600, 0, 8, 1, 512, 512);
    SetComTime(PORT, 1.0); //Set port time-out to 1 sec
    FlushInQ(PORT);
    FlushOutQ(PORT);
    return 0;
}

static void GCI_init_heater_mat_ctrl(void)
{
    ReadPIDValues();
}

static void SendPIDValues(void)
{
    int prop_val,integral_val,derivitive_val;
    char val1[20],data_ret[10],msg[256];
    int data,stat,cmd,num_bytes;

    while (GetInQLen(PORT)) { //dump bytes which shouldn't be there
        if (rs232err < 0) ;
        ComRdByte(PORT);
    }
    GetCtrlVal(HtrMatCtrlPanel, HRTMATPNL_PROP,&prop_val);
    GetCtrlVal(HtrMatCtrlPanel, HRTMATPNL_INTEGRAL,&integral_val);
    GetCtrlVal(HtrMatCtrlPanel, HRTMATPNL_DERIVITIVE,&derivitive_val);
    num_bytes = 8;
    cmd = 1;
    //Command for PID setting values
    val1[0] = cmd;
    val1[1] = num_bytes;
    //Number of bytes
    val1[2] = prop_val>>8;
    val1[3] = prop_val & 0xFF;
    val1[4] = integral_val>>8;
    val1[5] = integral_val & 0xFF;
    val1[6] = derivitive_val>>8;
    val1[7] = derivitive_val & 0xFF;
    ComWrt(PORT, val1, num_bytes);
    data_ret[6]=0; //Clear array
    // stat = GetInQLen(PORT);
    ComRd(PORT, data_ret,7); //Read ack
    if(data_ret[0]==4){
        sprintf(msg, "ACK-ERROR");
        MessagePopup("USB: ",msg);
    }
}

static void ReadPIDValues(void)
{
    char val1[20],data_ret[10],msg[256];
    unsigned int stat,cmd,num_bytes,prop_val,integral_val,derivitive_val;
    unsigned int prop_val_l,integral_val_l,derivitive_val_l;
    num_bytes = 1;
    cmd = 4;
    ComWrt(PORT, val1, num_bytes);
    data_ret[0]=0; //Clear array
    // stat = GetInQLen(PORT);
    ComRd(PORT, data_ret,7); //Read data
    if(data_ret[0]==4){
        sprintf(msg, "ACK-ERROR");
        MessagePopup("USB: ",msg);
    }
    prop_val = data_ret[0] & 0xff ;

Heated stage system for small rodents.doc
11
prop_val_l = data_ret[1] & 0xff;
integral_val_l = data_ret[3] & 0xff;
derivative_val_l = data_ret[5] & 0xff;

prop_val = prop_val<<8 | prop_val_l;
integral_val = integral_val<<8 | integral_val_l;
derivative_val = derivative_val<<8 | derivative_val_l;

SetCtrlVal(HtrMatCtrlPanel, HRTMATPNL_PROP, prop_val);
SetCtrlVal(HtrMatCtrlPanel, HRTMATPNL_INTERGRAL, integral_val);
SetCtrlVal(HtrMatCtrlPanel, HRTMATPNL_DERIVITIVE, derivative_val);

int CVICALLBACK cb_quit (int panel, int control, int event,
  void *callbackData, int eventData1, int eventData2)
{
  switch (event)
  {
    case EVENT_COMMIT:
      SetCtrlAttribute(HtrMatCtrlPanel, HRTMATPNL_TIMER, ATTR_ENABLED, 0);  //Disable timer
      QuitUserInterface (0);
      break;
  }
  return 0;
}

int CVICALLBACK cbprop (int panel, int control, int event,
  void *callbackData, int eventData1, int eventData2)
{
  switch (event)
  {
    case EVENT_COMMIT:
      SendPIDValues();
      break;
  }
  return 0;
}

int CVICALLBACK cbintegral (int panel, int control, int event,
  void *callbackData, int eventData1, int eventData2)
{
  switch (event)
  {
    case EVENT_COMMIT:
      SendPIDValues();
      break;
  }
  return 0;
}

int CVICALLBACK cbdervative (int panel, int control, int event,
  void *callbackData, int eventData1, int eventData2)
{
  switch (event)
  {
    case EVENT_COMMIT:
      SendPIDValues();
      break;
  }
  return 0;
}

int CVICALLBACK cbstorepid (int panel, int control, int event,
  void *callbackData, int eventData1, int eventData2)
{
  char val1[20], data_ret[10], msg[256];
  int stat, cmd, num_bytes;

  switch (event)
  {
    case EVENT_COMMIT:
      num_bytes = 1;
      cmd = 3;  //Command for writing PID values to EEPROM
      val1[0] = cmd;
      ComWrt(PORT, val1, num_bytes);
      data_ret[0]=0;  //Clear array

      //stat = GetInQLen(PORT);
      ComRd(PORT, data_ret[10], 10);  //Read ack
      if(data_ret[0]==4)
        sprintf(msg, "ACK ERROR");
        MessagePopup("USB: ", msg);
      break;
  }
  return 0;
}

Heated stage system for small rodents.doc
int CVICALLBACK cbtimer (int panel, int control, int event,  
    void *callbackData, int eventData1, int eventData2)  
{
    char val[20],data_ret[10],msg[256];
    unsigned int stat,cmd,num_bytes,temp_val,max_power_val,temp_set_point_val;
    unsigned int temp_set_point_val_l,temp_val_l,max_power_val_l;
    double max_power_val_d,temp_val_d,temp_set_point_val_d;

    switch (event)  
    {  
        case EVENT_TIMER_TICK:  
            while (GetInQLen(PORT))  
            {  
                num_bytes = 1;
                cmd = 5;  
                val[0] = cmd;
                ComWrt(PORT, val,1, num_bytes);  
                data_ret[6]=0;  
                //Command to read PID values
                //stat = GetInQLen(PORT);
                //ComRd(PORT, data_ret,7);  
                if(data_ret[6]!=4){  
                    sprintf(msg, "ACK‐ERROR");
                    MessagePopup("USB: ",msg);
                }
                temp_val = (data_ret[0] <<8);
                temp_val_l = data_ret[1] & 0xFF;
                max_power_val = data_ret[2] <<8;
                max_power_val_l = data_ret[3] & 0xFF;
                temp_set_point_val = data_ret[4] <<8;
                temp_set_point_val_l = data_ret[5] & 0xFF;
                temp_val = temp_val | temp_val_l;
                max_power_val = max_power_val | max_power_val_l;
                temp_set_point_val = temp_set_point_val | temp_set_point_val_l;

                temp_val_d = (temp_val*41.0)/1023;
                max_power_val_d = max_power_val/10.23;
                temp_set_point_val_d = (temp_set_point_val*41.0)/1023;

                SetCtrlVal(ReadEepromPanel, READEEPROM_TEMP,temp_val_d);
                SetCtrlVal(ReadEepromPanel, READEEPROM_MAX_POWER,max_power_val_d);
                SetCtrlVal(ReadEepromPanel, READEEPROM_TEMP_SET_POINT,temp_set_point_val_d);
            }
            break;
    }
    return 0;
}  

int CVICALLBACK cb_read (int panel, int control, int event,  
    void *callbackData, int eventData1, int eventData2)  
{
    switch (event)  
    {  
        case EVENT_COMMIT:
            DisplayPanel(ReadEepromPanel);
            SetCtrlAttribute (HtrMatCtrlPanel, HRTMATPNL_TIMER, ATTR_ENABLED, 1);  
                //Enable timer
            break;
    }
    return 0;
}  

int CVICALLBACK cbreturn (int panel, int control, int event,  
    void *callbackData, int eventData1, int eventData2)  
{
    switch (event)  
    {  
        case EVENT_COMMIT:
            SetCtrlAttribute (HtrMatCtrlPanel, HRTMATPNL_TIMER, ATTR_ENABLED, 0);  
                //Disable timer
            HidePanel(ReadEepromPanel);
            break;
    }
    return 0;
}  

int CVICALLBACK cb_readeeprom (int panel, int control, int event,  
    void *callbackData, int eventData1, int eventData2)  
{
    char val[20],data_ret[10],msg[256];

Heated stage system for small rodents.doc
unsigned int stat,cmd,num_bytes,prop_val,integral_val,derivative_val;
unsigned int prop_val_l,integral_val_l,derivative_val_l;

switch (event) {
    case EVENT_COMMIT:
        num_bytes = 1;
        cmd = 4; //Command for reading PID values
        val1[0] = cmd;

        ComWrt(PORT, val1, num_bytes);
        data_ret[6]=0; //Clear array

        // stat = GetInQLen(PORT); //Read data
        if(data_ret[6]==4){
            sprintf(msg, "ACK ERROR");
            MessagePopup("USB: ",msg);
        }

        prop_val = data_ret[0] & 0xff;
        prop_val_l = data_ret[1] & 0xff;
        integral_val = data_ret[2] & 0xff;
        integral_val_l = data_ret[3] & 0xff;
        derivative_val = data_ret[4] & 0xff;
        derivative_val_l = data_ret[5] & 0xff;

        prop_val = prop_val<<8 | prop_val_l;
        integral_val = integral_val<<8 | integral_val_l;
        derivative_val = derivative_val<<8 | derivative_val_l;

        SetCtrlVal(ReadEepromPanel, READEEPROM_PROP,prop_val);
        SetCtrlVal(ReadEepromPanel, READEEPROM_INTERGRAL,integral_val);
        SetCtrlVal(ReadEepromPanel, READEEPROM_DERIVITIVE,derivative_val);

        break;
    }
    return 0;
}

The ‘Set point’ indicator relays the set-point temperature, as set by the hardware control, while the ‘Temperature’ indicator shows the actual temperature. When the ‘Chart’ button is pressed, a chart recorder-like display is shown, as depicted in Figure 7.

6. Performance

The typical performance of the system is also shown in Figure 7; warm-up time is less than 0.5 hour to ‘typical’ temperatures of ~35 degC and the cooling rate is comparable to heating rate.

In common with all ‘heating-only’ types of thermostats, the temperature control around ambient room temperature (or surround temperature) will be somewhat poor, unless heat losses are particularly high, and the system has been optimised for working temperatures at least 5 degrees above ambient. The response to load changes is reasonably fast (<2 mins). Upon application of a ‘cold’ load, the temperature drops and recovers within 60 secs; when the now-warmed load is removed, the temperature sensor ‘sees’ the ambient temperature (in this instance room temperature) and the temperature once again drops and recovers.

It is pointed out that the performance at a higher ambient temperature will be inevitably different to that shown here and that some adjustment of the PID gains may be required. Please contact the authors should this be required.
All of the components used in this project are readily available from the following suppliers:

Onecall  http://www.farnell.co.uk
RS Components Ltd http://rswww.com/
Future Technology Devices International Ltd http://www.ftdichip.com

The electronic components used in the heated stage insert are listed below. However note that the prices refer to November 2010….since the UK economic situation is currently volatile (!) they should only be used as a guide.
<table>
<thead>
<tr>
<th>Item</th>
<th>Supplier</th>
<th>Stock #</th>
<th>Part description</th>
<th>Qty</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heater mats</td>
<td>RS</td>
<td>245-512</td>
<td>Self-adhesive heatermat, 3.75W 12V 50 x 75mm</td>
<td>2</td>
<td>£ 29.28</td>
</tr>
<tr>
<td>Connecting cable</td>
<td>Onecall</td>
<td>131-003</td>
<td>Tyco Electronics T01-0599-B07-B, 7 way, 3 metres</td>
<td>1</td>
<td>£ 30.12</td>
</tr>
<tr>
<td>Temperature sensor</td>
<td>Onecall</td>
<td>179-8368</td>
<td>National Semiconductor LM62BIM3 - Temp Sensor, 3SOT23</td>
<td>1</td>
<td>£ 0.39</td>
</tr>
<tr>
<td>Enclosure</td>
<td>Onecall</td>
<td>301-243</td>
<td>Multicomp MB1, black</td>
<td>1</td>
<td>£ 1.82</td>
</tr>
<tr>
<td>Output connector</td>
<td>Onecall</td>
<td>130-886</td>
<td>Tyco Electronics, 7 way, panel mount, T01-0580-P07</td>
<td>1</td>
<td>£ 14.34</td>
</tr>
<tr>
<td>DC power input connector</td>
<td>Onecall</td>
<td>124-3248</td>
<td>Lumberg chassis socket, panel mount, 2.1 mm</td>
<td>1</td>
<td>£ 1.65</td>
</tr>
<tr>
<td>Power supply</td>
<td>Onecall</td>
<td>127-9473</td>
<td>Stotronics ROSA-151A-12UK 12V DC 1.25A</td>
<td>1</td>
<td>£ 14.62</td>
</tr>
<tr>
<td>Digital panel meter</td>
<td>Onecall</td>
<td>993-2844</td>
<td>Lascar - EMV 1025S-01 - 3.5 digit LCD Voltmeter 200 mV FSD</td>
<td>1</td>
<td>£ 16.57</td>
</tr>
<tr>
<td>Push button switch</td>
<td>RS</td>
<td>253-942</td>
<td>Honeywell 8N2021-Z. DPDT, Momentary</td>
<td>1</td>
<td>£ 5.95</td>
</tr>
<tr>
<td>Temperature set potentiometer</td>
<td>Onecall</td>
<td>151-7316</td>
<td>4k7 potentiometer with knob</td>
<td>1</td>
<td>£ 3.61</td>
</tr>
<tr>
<td>Power on LED</td>
<td>Onecall</td>
<td>104-5459</td>
<td>Vishay TLUR44HG, green LED 3mm</td>
<td>1</td>
<td>£ 0.192</td>
</tr>
<tr>
<td>Heating on LED</td>
<td>Onecall</td>
<td>104-5372</td>
<td>Vishay TLUR4400, red, LED 3mm</td>
<td>1</td>
<td>£ 0.143</td>
</tr>
<tr>
<td>Voltage regulator</td>
<td>Onecall</td>
<td>101-4073</td>
<td>Fairchild Semiconductors 78L05 5V regulator TO92</td>
<td>1</td>
<td>£ 0.194</td>
</tr>
<tr>
<td>Gain, offset amplifier</td>
<td>Onecall</td>
<td>955-9280</td>
<td>Linear technology, LT1013CN8, op amp, 8 pin DIP</td>
<td>1</td>
<td>£ 3.66</td>
</tr>
<tr>
<td>Voltage reference</td>
<td>Onecall</td>
<td>175-5114</td>
<td>Texas Instruments LM4040C11LP, 4.096 V, TO92</td>
<td>1</td>
<td>£ 0.356</td>
</tr>
<tr>
<td>MOSFET driver</td>
<td>Onecall</td>
<td>146-7958</td>
<td>2N7000 D26Z MOSFET, N, TO-92</td>
<td>1</td>
<td>£ 0.75</td>
</tr>
<tr>
<td>MOSFET power</td>
<td>Onecall</td>
<td>170-4021</td>
<td>IRF5210 MOSFETPBF, P, TO220</td>
<td>1</td>
<td>£ 2.03</td>
</tr>
<tr>
<td>USB controller</td>
<td>FTDI</td>
<td>DLP-245PB-G</td>
<td>USB / Microcontroller module</td>
<td>1</td>
<td>£ 22.20</td>
</tr>
<tr>
<td>Offset and gain presets</td>
<td>Onecall</td>
<td>935-3186</td>
<td>Bourns 3296W-1-103LF trimmer, 25 turn, 10 kΩ</td>
<td>2</td>
<td>£ 4.06</td>
</tr>
<tr>
<td>Set max power preset</td>
<td>Onecall</td>
<td>935-3240</td>
<td>Bourns 3296W-1-203LF trimmer, 25 turn, 20 kΩ</td>
<td>1</td>
<td>£ 2.03</td>
</tr>
<tr>
<td>Resistors</td>
<td>Onecall</td>
<td>various</td>
<td>MRS25 series 0.6W</td>
<td>13</td>
<td>£ 0.65</td>
</tr>
<tr>
<td>Input decoupling capacitor</td>
<td>Onecall</td>
<td>941-1887</td>
<td>Multicomp, MCFYU610426, 50V, 100 nF</td>
<td>1</td>
<td>£ 0.066</td>
</tr>
<tr>
<td>Decoupling capacitors</td>
<td>Onecall</td>
<td>175-3978</td>
<td>Vishay Sprague, tantalum capacitor, 1 µF, 25 V</td>
<td>3</td>
<td>£ 2.94</td>
</tr>
<tr>
<td>Smoothing capacitor</td>
<td>Onecall</td>
<td>183-2428</td>
<td>Multicomp-MCTEA102M1EB , axial, 1000 µF, 25 V</td>
<td>1</td>
<td>£ 0.429</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>£ 158.05</strong></td>
</tr>
</tbody>
</table>

7. Conclusion

A heated stage insert and temperature controller have been described, compatible with modern inverted microscopes. The inspiration for this project was provided by Dr Jacco Van Rheenen from the Hubrecht Institute in Holland ([www.hubrecht.eu](http://www.hubrecht.eu)) who devised a simple and effective window chamber mount which allows microscopic resolution examination of rodent tumours. It is this chamber which fits inside the 14 mm hole in the stage plate (Figure 3 and below). The construction of the heated stage and the PIC-firmware PID controller may be found useful for other applications. Detailed SolidWorks drawings of the assembly are available on request. No printed circuit board design was developed for this as the wiring is extremely simple; the electronics were constructed on a prototype board.
This system was developed during November 2010 by J Prentice and B Vojnovic (mechanics) and by RG Newman and B Vojnovic (electronics, software), in consultation with R Muschel, T Tapmeier (biology). This note was written by B Vojnovic and RG Newman in December 2010 and updated in August 2011.

We acknowledge the financial support of Cancer Research UK, the MRC and EPSRC.

© Gray Institute, Department of Oncology, University of Oxford, 2011.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.