Laboratory 3

ENGG4420: Real-Time Systems Design

Instructor:
Dr.Radu Muresan

Group 17: Wed-8:30 Section

Bilal Ayyache: 0988616
Robert Mackenzie Beggs: 0819747

Lab Start Date: October 23rd, 2020
Lab End Date: November 06th, 2020

Contents

1 Introduction

1.1 Problem Description

Real time systems are by nature multivariate; their reaction to a given environment depends not only on the parameters of the environment and the plant, but also on the time at which a given interaction occurs. Given the amount of parameters involved, specific mechanisms and methodologies are considered during design. Previously, simulations for real time systems, specifically a hot air plant with a PID controller tuned via the Ziegler and Nichols approach, were created through use of LabView Control Design ToolKit Box VIs (LabView). LabView is a graphical programming language that allows complete systems to be efficiently designed and debugged, obfuscating some of the difficulty of implementing real-time systems embedded systems on controllers. This work aims to accommodate the same constraints of previous works through performing PID feedback control of a given hot air plant via the development of an embedded RTOS uC/OS based controller, while overcoming challenges that were antithetical to previous works.

1.2 System Requirements

A given task is delineated into categories labeled hard, firm or soft, dependent on the repercussions of missing a deadline associated with the task. The differentiation of tasks according to these consequences is characteristic of a real time system [4]. These temporal constraints coupled with the requirement of regulating use of system resources any given moment in time makes the developing process challenging. A plant interface, acting as input to the embedded PID controller, must be constructed to operate in accordance with the temporal and resource constraints that exist in a real time paradigm. The PID controller must have a fixed sampling time of 200 ms, communicated to the end user in standard analog form, derived via the Ziegler and Nichols tuning approach. Additionally, the LCD display must be properly configured to display the plot of reference voltage through time, as well as general settings and must be updated at a set frequency. Finally, the software must adhere to the following specifications [5]:

- DDC for temperature control
- Operator display
- Operator input
- Provision of management information
- System start-up and shut-down
- Clock function

2 Background

2.1 PID Control and Tuning Overview

In order to ensure the stability of a given PID controller, the transfer function is tuned pending satisfaction of the Nyquist stability criterion; a graphical representation of the stability of a given system. To do so, integral and derivative gains of the system equated to zero and the proportional gain is gradually increased until the system output reaches steady-state. This is expressed by the following equations, in which K_p, K_d and K_i is represent the proportional, derivative and integral gains respectively:

$$u(t) = K_p\left(e(t) + K_i\int_0^t e(\tau)d\tau + K_d\frac{de(t)}{dt}\right) \tag{1}$$

Furthermore, the error can be expressed by:

$$u(s) = K_p\left(1 + \frac{1}{T_i s}e(s)\right)e(s) = K_p\left(\frac{T_d T_i s^2 + 1}{T_i s}e(s)\right) \tag{2}$$

2.2 uC/OS-III RTOS

A method of coping with the temporal and resource constraints that are inherent in real time environments is to implement priority. Priority refers to the urgency to which a task must be executed relative to another [4]. Micro-Controller Operating Systems Version 3 (uC/OS-III RTOS), is a multitasking operating system designed for the implementation of real time systems on embedded devices. uC/OS-III does not limit the priority range of a given task; this is instead limited by available system memory. Therefore, any number of application tasks, priority levels, and tasks per level can be allocated, allowing for flexibility during development [1].

2.3 Hardware & Software Implementation Tools

The selected IDE for implementation of this project was TrueStudio, and the STM32F429I-DISC1 (STM32) was used as an ST evaluation board. The STM32 evaluation board features the STMSTM32F429 microcontroller. True Studio was conceived to support STM32 devices and development boards, therefore is able to efficiently compile, and analyze developed code. The paradigm of real time operating systems was considered to such a degree that the condition of semaphores and mutexes during debugging can be evaluated [6].

3 Implementation

3.1 Main

Given a pointer to the message queue block, a name, a size limit for the queue, and a pointer to an error code, 'OSQCreate' creates a message queue that grants tasks or interrupt service routines (ISR) the

ability to send messages to tasks [2]. In main, multiple purpose specific queues are created. Additionally, tasks are defined via the use of 'OSTaskCreate'. 'OSTaskCreate' accepts a pointer to a task control block (tcb), a string representing the name or I.D of the task, a pointer to the function that represents the task, a pointer to an agnostic function which passes parameters, an integer representing task priority, a pointer to the base address of the task, the size of the stack, the maximum number of messages the task can receive via a message queue, a bit representing distinct options dependent on task and a pointer to a variable that is responsible for receiving an error code [2].

3.2 Startup Task

The primary purpose of 'StartupTask' is to initialize the hardware and software parameters required. Additionally, 'OS_TRACE_INIT' initializes the trace recorder, allowing for the stack to be traced, at any point following 'BSP_OS_TickEnable'.

3.3 App Task Create

The purpose of 'App Task Create' is to define tasks via 'OsTaskCreate' for application downstream. Tasks for the button task, comm task, GUI task, PID task and plant tsk are defined.

3.4 Button Task

'HAL_GPIO_ReadPin' receives arguments indicating the GPIO peripheral to be selected and desired port bit and returns the input port value; this is compared to the GPIO pin set, and in the case that they are equal a counter for the number of button presses is incremented [3]. If the current control mode is set to auto, it is then switched manual; in the case that it is manual, it is switched to auto. This process executes indefinitely with a 200 ms delay between each cycle, as long as DEF remains true.

3.5 Comm Task

Given plant inputs, set points and operator inputs, 'OSQPend' is utilized to receive messages from the respective queues. Additionally, the buffer is continuously adjusted proportionally to the size indicated by the Comm Queue.

3.6 GUI Task

The GUI task uses 'BSP_LCD' functions to update the appearance of the GUI given that certain conditions are met. Via 'OSQPend', messages from the plant and operator are continuously received. The parameters displayed on the LCD are altered depending on the state of the control mode and the status of system. This process is continued indefinitely, with a delay of 300 clock ticks between cycles.

3.7 Plant Task

A discrete transfer function is continuously evaluated via 'OSQPend'. Given a pointer to a queue, as well as the timeout limit, 'OSQPend' recieves messages from the specified queue. The output is then calculated by considering the transfer function and the output is passed to a specified queue via 'OSQPost'. The process is then paused for 200ms before it is repeated indefinitely.

3.8 PID Task

Given a pointer to the queue created by 'OSQCreate', an int representing the timeout in clock ticks, an option argument signifying blocking or non blocking, the message size in bytes, a pointer to the variable that will track the time the message is received and a pointer to a variable that holds an error code, 'OSQPend' recieves message from the given queue. After the message is received, it is configured as input to the PID. Additionally, the change in output is calculated using the proportional gain as specified by:

$$\Delta output = K_p * (\Delta Error_1 + (CurrentError * \frac{T_s}{T_i}) + (\Delta Error_2 * \frac{T_d}{T_s})) \tag{3}$$

Finally, 'OSQPOST', taking the same arguments as 'OSQCreate' delivers a message via a given queue. This process is repeated while DEF_ON remains true.

3.9 Helper Functions

The system helper functions defined aid in allowing other tasks to be performed. 'createAxis', 'drawLine', 'showText' and 'drawReferenceLine' aid in drawing the GUI via 'BSP_LCD' functions. 'BSP_LCD' functions are predefined library functions created by STM that allow for common processes such as drawing horizontal or vertical lines to be easily conducted [2]. By combining several of these functions, helper functions were created to draw axis, lines, text and reference lines in the desired location on the LCD screen. Additionally, a helper functions were created to convert temperature to voltage via the following:

$$voltage = -0.0015 * temperature^2 + 0.3319 * temperature - 6.9173; \tag{4}$$

Likewise, voltage was converted to temperature via:

$$temperature = 0.3053 * voltage^2 + 2.2602 * voltage + 25.287 \tag{5}$$

Finally, a helper function was defined to configure the system clock. This was accomplished via use of 'RCC_OscInitStruct' functions. 'RCC_OscInitStruct' functions, are also predefined and used to define the function of elements of the operating system.

4 Results

4.1 Manual Settings

In manual settings, the voltage set-point was set to 6V. The manual control is an open loop system in which an input voltage is sent to the tuned PID controller to get the desired set-point. No feedback error is provided to the PID. Figure 1 represents the response of the system. The response has an overshoot and an undershoot. This can be controlled depending on the design requirements set by the programmer using the PID values (The integrator value changes the percentage of the overshoot and undershoot. A higher integrator value would result in a smaller percentage in both but this would make it longer for the controller to get to the steady state). Figure 2 represents the output of the system. It is clear that the manual setting was completely functional as the controller was able to achieve a voltage of approximately six volts. Value is not exactly six volts due to the steady state error. The plant input required to achieve this set-point was 4.732 volts.

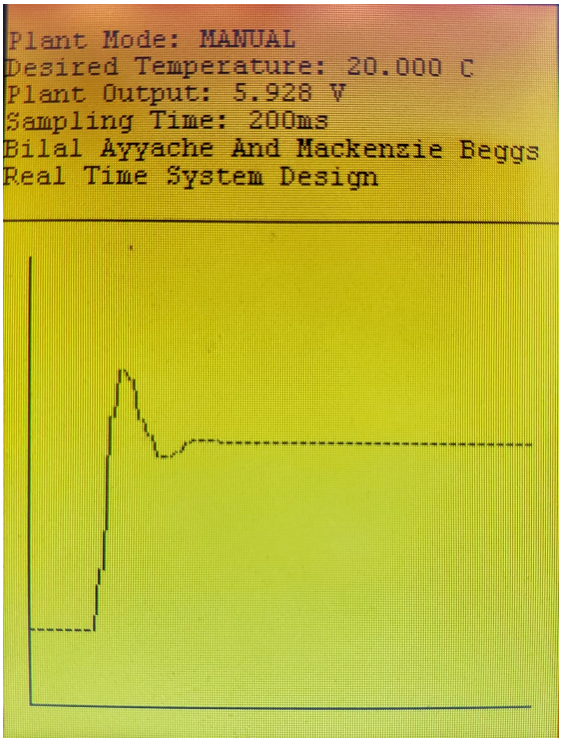

Figure 1: Simulation of Manual Setting

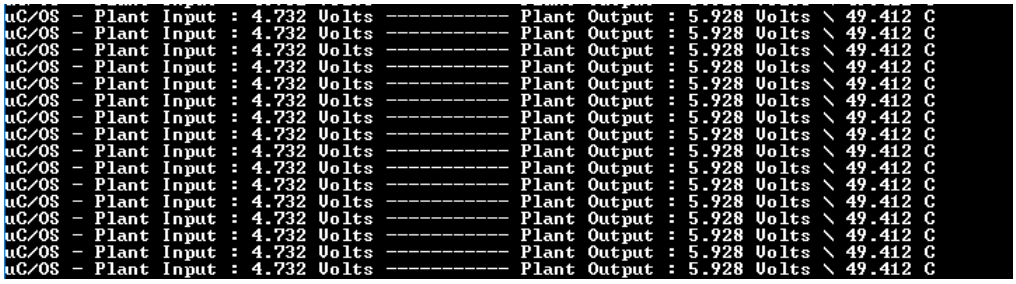

Figure 2: Control System for manual Setting

4.2 Auto Settings

Temperature for auto settings was arbitrarily chosen to be 60 °C. The auto setting is a closed loop system that converts voltage to temperature. In this setting, the performance of the PID controller falls within an acceptable range and is able to match the performance delivered in previous works. In this experimental setting, we aim to achieve a response that is measured in degrees Celsius. To accomplish this, the input to the plant must be set to 6V, rendering an output of 7.6V which is converted to temperature via a helper function as seen in Section 3.9. The overshoot and undershoot of the system is explained in Section 4.1. Figure 4 depicts a constant output due to the fact that it was taken at the end of the simulation; this illustrates that the system was able to achieve the set point selected by the user, the system is in steady state.

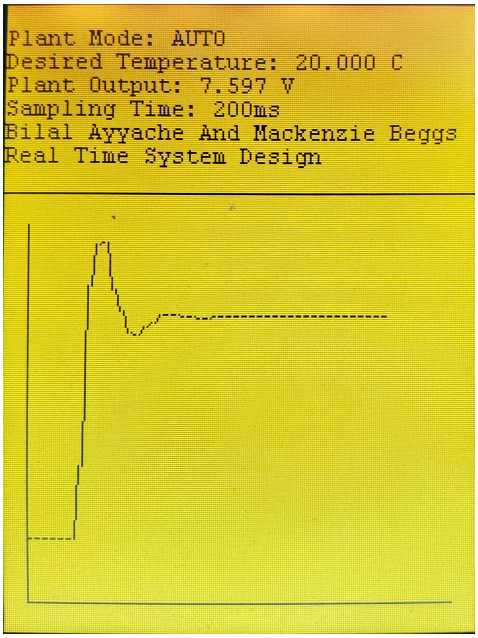

Figure 3: Caption

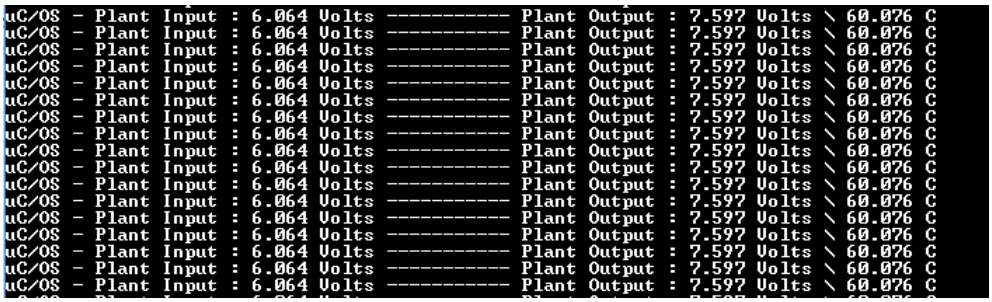

Figure 4: Control System for Auto Setting

5 Conclusion

Simple plants can be represented by real time systems by dividing the tasks of a system into the categories of monitoring, control and actuation. Monitoring involves gathering information about the state of the system via sampling the environment at a given point in time. Control involves relaying the information received through monitoring to a transfer function to continuously actuate required changes to ensure either stability or a required result. A real time system capable of regulating the temperature of air at the output vent of a PT326 process trainer was developed using the Ziegler and Nichols approach to tuning,

and implemented on an embedded device. Implementation of real time systems on embedded devices is challenging as compared to implementation of a comparable system via simulation due to the increased difficulty of debugging. The task of implementing more complex real-time applications is left for future experimentation.

References

[1] 2020. URL: https://www.micrium.com/rtos/.

[2] 2020. URL: https://doc.micrium.com/display/osiiidoc.

[3] 2020. URL: http://www.disca.upv.es/aperles/arm_cortex_m3/llibre/st/STM32F439xx_User_Manual/group__gpio__exported__functions__group2.html.

[4] Giorgio C Buttazzo. *Hard real-time computing systems*. 3rd ed. Springer, 2011.

[5] Radu Muresan and Kevin Dong. *ENGG4420: REAL-TIME SYSTEMS DESIGN - LAB MANUAL*. 2020.

[6] *True Studio Documentation*. 2020. URL: https://atollic.com/truestudio/.

A Appendix

A.0.1 Main

```
#include "main.h"
#include "stm32f4xx_hal.h"
#include "usb_device.h"
#include "gpio.h"

/* user Includes */
#include "includes.h"


/* Private variables ---------------------------------------------------------*/
uint8_t RxData[256];
uint32_t data_received = 0;
uint8_t Str4Display[100];


static  OS_TCB   StartupTaskTCB;
static  CPU_STK  StartupTaskStk[APP_CFG_STARTUP_TASK_STK_SIZE];

static  OS_TCB   CommTaskTCB;
static  CPU_STK  CommTaskStk[APP_CFG_COMM_TASK_STK_SIZE];

static  OS_TCB   BtnTaskTCB;
static  CPU_STK  BtnTaskStk[APP_CFG_BTN_TASK_STK_SIZE];

static  OS_TCB   GuiTaskTCB;
static  CPU_STK  GuiTaskStk[APP_CFG_GUI_TASK_STK_SIZE];

static  OS_TCB         PlantTaskTCB;
static        CPU_STK         PlantTaskStk[APP_CFG_PLANT_TASK_STK_SIZE];
OS_Q          PlantInputQ;

static OS_TCB PIDTaskTCB;
static CPU_STK PIDTaskStack[APP_CFG_PID_TASK_STK_SIZE];
OS_Q PIDQ;

// Automode
OS_Q SetPointQ;
```

```
OS_Q GUIQ;

OS_Q   CommQ;

OS_Q OperatorInputQ;


typedef enum {
  AUTO,
  MANUAL
} Mode ;


static Mode currentControlMode = MANUAL;

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);

/* user private function prototypes */
static void AppTaskCreate(void);
static void StartupTask(void  *p_arg);
static void CommTask(void  *p_arg);
static void BtnTask(void  *p_arg);
static void GuiTask(void  *p_arg);
static void PIDTask(void *p_arg);


/* added functions for Lab 3 */
static void PlantTask(void *p_arg);
static float TempToVoltage(float temp);
static float VoltageToTemp(float volt);
static void createAxis(Point frame, int xAxisSize, int yAxisSize);
static void drawLine(Point a, Point b);
static void showText(uint16_t line, char* text);
static void drawReferenceLine(Point p, int distance);



/**
  * @brief  The application entry point.
  * @retval None
```

```
  */
int main(void)
{

  OS_ERR  os_err;

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */
  BSP_ClkInit();                                                /* Initialize the main clock
  BSP_IntInit();                                                /* Initialize the interrupt vector t
  BSP_OS_TickInit();                                            /* Initialize kernel tick timer

  Mem_Init();                                                   /* Initialize Memory Managment Modul
  CPU_IntDis();                                                 /* Disable all Interrupts
  CPU_Init();                                                   /* Initialize the uC/CPU services
  Math_Init();                                                  /* Initialize Mathematical Module

  OSInit(&os_err);                                              /* Initialize uC/OS-III
         */
  if (os_err != OS_ERR_NONE) {
      while (1);
  }

  App_OS_SetAllHooks();                                         /* Set all applications hooks

  OSQCreate(&CommQ,
            "Comm Queue",
             10,
            &os_err);                                                       /* Create COMM

  // plant input queue
  OSQCreate(&PlantInputQ,
                          "Plant Input Queue",
                        20,
                        &os_err);

  OSQCreate(&PIDQ,
```

```
                "PID Queue",
                20,
                &os_err);

  OSQCreate(&SetPointQ,
                "SetPoint Queue",
                20,
                &os_err);

  OSQCreate(&GUIQ,
                  "GUI Queue",
                  20,
                  &os_err);

  OSQCreate(&OperatorInputQ,
            "OperatorInputQ",
          20,
          &os_err);


  OSTaskCreate(&StartupTaskTCB,                                /* Create the startup task
               "Startup Task",
                StartupTask,
                0u,
                APP_CFG_STARTUP_TASK_PRIO,
               &StartupTaskStk[0u],
                StartupTaskStk[APP_CFG_STARTUP_TASK_STK_SIZE / 10u],
                APP_CFG_STARTUP_TASK_STK_SIZE,
                0u,
                0u,
                0u,
               (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
               &os_err);
  if (os_err != OS_ERR_NONE) {
      while (1);
  }

  OSStart(&os_err);                                            /* Start multitasking (i.e. give con

  while (DEF_ON) {}                                             /* Should Never Get Here.
```

```
}
```

A.0.2 Start Up Task

```
/*********************************************************************************************
*                                          STARTUP TASK
*
* Description : This is an example of a startup task.  As mentioned in the book's text, you MUST
*               initialize the ticker only once multitasking has started.
* Arguments   : p_arg   is the argument passed to 'StartupTask()' by 'OSTaskCreate()'.
* Returns     : none
* Notes       : 1) The first line of code is used to prevent a compiler warning because 'p_arg' is
*                  used.  The compiler should not generate any code for this statement.
*********************************************************************************************
*/

static  void  StartupTask (void *p_arg)
{
  OS_ERR  os_err;
  (void)p_arg;

  OS_TRACE_INIT();                                           /* Initialize the uC/OS-III Trace re

  BSP_OS_TickEnable();                                       /* Enable the tick timer and interru
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USB_DEVICE_Init();

  BSP_LED_Init();

  MX_DMA2D_Init();
  MX_FMC_Init();
  MX_I2C3_Init();
  MX_LTDC_Init();
  MX_SPI5_Init();

  BSP_LCD_Init();
  BSP_LCD_LayerDefaultInit(LCD_BACKGROUND_LAYER, LCD_FRAME_BUFFER);
  BSP_LCD_LayerDefaultInit(LCD_FOREGROUND_LAYER, LCD_FRAME_BUFFER);
  BSP_LCD_SelectLayer(LCD_FOREGROUND_LAYER);
  BSP_LCD_DisplayOn();
  BSP_LCD_Clear(LCD_COLOR_WHITE);
```

```
#if OS_CFG_STAT_TASK_EN > 0u
  OSStatTaskCPUUsageInit(&os_err);                          /* Compute CPU capacity with no task
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN
  CPU_IntDisMeasMaxCurReset();
#endif

  AppTaskCreate();                                          /* Create Application tasks

  while (DEF_TRUE) {                                        /* Task body, always written as an i
      BSP_LED_Toggle(0);
      OSTimeDlyHMSM(0u, 0u, 1u, 0u,
                    OS_OPT_TIME_HMSM_STRICT,
                    &os_err);

  }
}
```

A.0.3 App Task

```
/*********************************************************************************************************
*                                          AppTaskCreate()
*
* Description : Create application tasks.
* Argument(s) : none
* Return(s)   : none
* Caller(s)   : AppTaskStart()
* Note(s)     : none.
*********************************************************************************************************
*/

static  void  AppTaskCreate (void)
{
    OS_ERR  os_err;

    OSTaskCreate(&CommTaskTCB,                                  /* Create the comm task
                 "Comm Task",
                  CommTask,
                  0u,
                  APP_CFG_COMM_TASK_PRIO,
                 &CommTaskStk[0u],
                  CommTaskStk[APP_CFG_COMM_TASK_STK_SIZE / 10u],
                  APP_CFG_COMM_TASK_STK_SIZE,
                  2u,
                  0u,
                  0u,
                 (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 &os_err);
    if (os_err != OS_ERR_NONE) {
        while (1);
    }

    OSTaskCreate(&BtnTaskTCB,                                  /* Create the comm task
                 "Button Task",
                  BtnTask,
                  0u,
                  APP_CFG_BTN_TASK_PRIO,
                 &BtnTaskStk[0u],
                  BtnTaskStk[APP_CFG_BTN_TASK_STK_SIZE / 10u],
```

```
                APP_CFG_BTN_TASK_STK_SIZE,
                0u,
                0u,
                0u,
               (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
               &os_err);
    if (os_err != OS_ERR_NONE) {
        while (1);
    }

    OSTaskCreate(&GuiTaskTCB,                                /* Create the comm task
                 "GUI Task",
                  GuiTask,
                  0u,
                  APP_CFG_GUI_TASK_PRIO,
                 &GuiTaskStk[0u],
                  GuiTaskStk[APP_CFG_GUI_TASK_STK_SIZE / 10u],
                  APP_CFG_GUI_TASK_STK_SIZE,
                  0u,
                  0u,
                  0u,
                 (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 &os_err);

    if (os_err != OS_ERR_NONE) {
        while (1);
    }

    OSTaskCreate(&PlantTaskTCB,
                               "Plant Task",
                                PlantTask,
                                0u,
                                APP_CFG_PLANT_TASK_PRIO,
                               &PlantTaskStk[0u],
                                PlantTaskStk[APP_CFG_PLANT_TASK_STK_SIZE / 10u],
                                APP_CFG_PLANT_TASK_STK_SIZE,
                                0u,
                                0u,
                                0u,
                               (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                               &os_err);
```

```
    OSTaskCreate(&PIDTaskTCB,
                 "PID Task",
                     PIDTask,
                     0u,
                     APP_CFG_PID_TASK_PRIO,
                     &PIDTaskStack[0u],
                     PIDTaskStack[APP_CFG_PID_TASK_STK_SIZE / 10u],
                     APP_CFG_PID_TASK_STK_SIZE,
                     0u,
                     0u,
                     0u,
                     (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                     &os_err);

    if (os_err != OS_ERR_NONE) {
        while (1);
    }

}
```

A.0.4 Button Task

```
/*********************************************************************************************************
*                                                BUTTON TASK
*
* Description : This is an example of a Button task.
* Arguments   : p_arg   is the argument passed to 'BtnTask()' by 'OSTaskCreate()'.
* Returns     : none
* Notes       : 1) The first line of code is used to prevent a compiler warning because 'p_arg' is
*                  used.  The compiler should not generate any code for this statement.
*********************************************************************************************************
*/

static  void  BtnTask (void *p_arg)
{
  OS_ERR  os_err;
  unsigned int count_press = 0;

  (void)p_arg;

#if OS_CFG_STAT_TASK_EN > 0u
  OSStatTaskCPUUsageInit(&os_err);                              /* Compute CPU capacity with no task
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN
  CPU_IntDisMeasMaxCurReset();
#endif

  while (DEF_TRUE) {                                           /* Task body, always written as an i
        OSTimeDlyHMSM(0u, 0u, 0u, 10u,
                                        OS_OPT_TIME_HMSM_STRICT,
                                        &os_err);
    if(HAL_GPIO_ReadPin(KEY_BUTTON_GPIO_PORT, KEY_BUTTON_PIN) == GPIO_PIN_SET){

            count_press++;
            if(currentControlMode == AUTO) {
                    currentControlMode = MANUAL;
            } else {
                    currentControlMode = AUTO;
            }
```

```
            while(HAL_GPIO_ReadPin(KEY_BUTTON_GPIO_PORT, KEY_BUTTON_PIN)==GPIO_PIN_SET);
    }
  }
}
```

A.0.5 Comm Task

```
/*
*********************************************************************************************************
*                                                COMM TASK
*
* Description : This is an example of a Comm task.  As mentioned in the book's text, you MUST
*               initialize the ticker only once multitasking has started.
* Arguments   : p_arg   is the argument passed to 'StartupTask()' by 'OSTaskCreate()'.
* Returns     : none
* Notes       : 1) The first line of code is used to prevent a compiler warning because 'p_arg' is
*                  used.  The compiler should not generate any code for this statement.
*********************************************************************************************************
*/

static  void  CommTask (void *p_arg)
{
  OS_ERR  os_err;
  void        *p_msg;
  OS_MSG_SIZE  msg_size;
  CPU_TS       ts;
  char buffer[10];
  int index = 0;
  float input = 0.0;

  (void)p_arg;

#if OS_CFG_STAT_TASK_EN > 0u
  OSStatTaskCPUUsageInit(&os_err);                              /* Compute CPU capacity with no task
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN
  CPU_IntDisMeasMaxCurReset();
#endif

  while (DEF_TRUE) {/* Task body, always written as an infinite loop.       */


        p_msg = OSQPend(&CommQ,
                        0,
                        OS_OPT_PEND_BLOCKING,
```

```
                    &msg_size,
                    &ts,
                    &os_err);

        char value = *((char *)p_msg);

    if(value == '\r') {
            buffer[index] = '\0';
            index = 0;
            input = strtof(buffer, NULL);


            if (currentControlMode == MANUAL){
                        OSQPost(&PlantInputQ,
                                                (float *) &input,
                                                sizeof((void *)&input),
                                                OS_OPT_POST_FIFO,
                                                &os_err);
            }
            else{
                    OSQPost(&SetPointQ,
                                            (float *) &input,
                                                sizeof((void *)&input),
                                                OS_OPT_POST_FIFO,
                                                &os_err);
            }

            OSQPost(&OperatorInputQ,
                            (float *)&input,
                                sizeof((void *) &input),
                                OS_OPT_POST_FIFO,
                                &os_err);

    } else if(value != '\n') {
            buffer[index] = value;
            index++;
    }

  }
}
```

A.0.6 GUI Task

```
/*********************************************************************************************************
*                                                GUI TASK
*
* Description : This is the Graphical User Interfae task
* Arguments   : p_arg   is the argument passed to 'GuiTask()' by 'OSTaskCreate()'.
* Returns     : none
* Notes       : 1) The first line of code is used to prevent a compiler warning because 'p_arg' is
*                  used.  The compiler should not generate any code for this statement.
*********************************************************************************************************
*/
void GuiTask(void *p_arg)
{

  OS_ERR  os_err;

  CPU_TS       ts;

  int ledXSize = BSP_LCD_GetXSize();
  int startXPosition = ledXSize * 0.05;

  int ledYSize = BSP_LCD_GetYSize();
  int startYPosition = ledYSize * 0.95;

  Point coordinateFrame = {startXPosition, startYPosition};

  int xAxisMax = ledXSize * 0.9;
  int yAxisMax = ledYSize * 0.6;

  int maxVoltage = 10;

  float yScaleFactor = yAxisMax / maxVoltage;

  Point previousPoint = {startXPosition, startYPosition};
  int xSpacing = 2;
  Point currentPoint = {0, 0};

  void *p_plant_msg;
  void *p_operator_input_msg;
  OS_MSG_SIZE msg_size = 0;
```

```
int plantModeLine = 1;
char plantMode[30];

int plantOutputLine = 3;
float plantOutput = 0.0;
char plantOutputStr[30];

int operatorInputLine = 2;
float operatorInput = 0.0;
float voltageInput = 0.0;
float temperatureInput = 20.0;
char operatorInputStr[50];

int samplingTimeLine = 4;
char samplingTimeStr[50];

int nameLine = 5;
char name[50];

int courseLine = 6;
char course[50];
BSP_LCD_Clear(LCD_COLOR_ORANGE);

Point referencePoint = {startXPosition, startYPosition};

while(1)
{
        BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
        BSP_LCD_SetBackColor(LCD_COLOR_ORANGE);
        BSP_LCD_SetFont(&Font12);

        if(previousPoint.X > xAxisMax + startXPosition) {
                        BSP_LCD_Clear(LCD_COLOR_ORANGE);
                        previousPoint.X = startXPosition;
        }

        BSP_LCD_DrawHLine(0, ledYSize * 0.3, ledXSize);

        p_plant_msg = OSQPend(&GUIQ,
                        200,
```

```
                OS_OPT_PEND_BLOCKING,
                &msg_size,
                &ts,
                &os_err);

plantOutput = *((float*)p_plant_msg);

p_operator_input_msg = OSQPend(&OperatorInputQ,
                0,
                OS_OPT_PEND_NON_BLOCKING,
                &msg_size,
                &ts,
                &os_err);

if(msg_size > 0) {
        operatorInput = *((float *)p_operator_input_msg);
        if(currentControlMode == AUTO) {
                //temperatureInput = operatorInput;
                voltageInput = operatorInput;
        } else {
                voltageInput = operatorInput;
        }
}


currentPoint.X = previousPoint.X + xSpacing;
currentPoint.Y = startYPosition - (plantOutput * yScaleFactor) ;


if(currentControlMode == AUTO) {
        strcpy(plantMode, "Plant Mode: AUTO");
        sprintf(operatorInputStr, "Desired Temperature: %.3f C", temperatureInput);
        referencePoint.Y = startYPosition - (TempToVoltage(temperatureInput) * yScaleFac
        drawReferenceLine(referencePoint, xAxisMax);
        BSP_LCD_DisplayStringAt(referencePoint.X, referencePoint.Y, "Vref", CENTER_MODE)
} else {
        strcpy(plantMode, "Plant Mode: MANUAL");
        sprintf(operatorInputStr, "Plant Input Voltage: %.3f V", voltageInput);
}

sprintf(plantOutputStr, "Plant Output: %.3f V", plantOutput);
```

```
            sprintf(samplingTimeStr, "Sampling Time: 200ms");
            sprintf(name, "Bilal Ayyache And Mackenzie Beggs");
            sprintf(course, "Real Time System Design");

            showText(plantModeLine, plantMode);
            showText(operatorInputLine, operatorInputStr);
            showText(plantOutputLine, plantOutputStr);
            showText(samplingTimeLine, samplingTimeStr);
            showText(nameLine,name);
            showText(courseLine, course);

            createAxis(coordinateFrame, xAxisMax, yAxisMax);

            drawLine(previousPoint, currentPoint);

            previousPoint = currentPoint;
            previousPoint.X += xSpacing;

            OSTimeDlyHMSM(0u, 0u, 0u, 300u,
                                                    OS_OPT_TIME_HMSM_STRICT,
                                                    &os_err);
  }
}
```

A.0.7 Plant Task

```
/*********************************************************************************************
*                                          Plant TASK
*
* Description : This task is the discretized transfer function of the Hot Air Plant from Lab 1 & 2
*
*                                              0.119217
*                                  H(z) = ---------------
*                                                z - 0.904837
*
*                                  using settings:
*
*                                   - Sample Time = 200ms
*                                  - Blower opening = 50
*
* Arguments   : p_arg   is the argument passed to 'PlantTask()' by 'OSTaskCreate()'.
* Returns     : none
* Notes       : 1)         You must tune your PID gains in LabVIEW with the above settings (in Des
*                                      so that those gains will work when used within your PID
*
*                                  2)  float plant_intput --> input to descretized transfer functio
*                                      float plant_output --> output of descretized transfer functi
*********************************************************************************************
*/

static float prev_plant_output = 0;
static float plant_input = 0;

static void PlantTask(void *p_arg){

        OS_ERR  os_err;
        void        *p_msg;
        OS_MSG_SIZE  msg_size;
        CPU_TS       ts;

        float plant_output = 0;

        while(1){

                // Get voltage from plant input queue
```

```
p_msg = OSQPend(&PlantInputQ,
                 0,
                 OS_OPT_PEND_NON_BLOCKING, // doesn't wait for queue
                &msg_size,
                &ts,
                &os_err);


if (msg_size > 0){
        plant_input = *((float *) p_msg);
}

// calculate plant output based on discrete transfer function
//plant_output = 0.02*plant_input +0.9995*prev_plant_output;
    plant_output = 0.119217*plant_input +0.904837*prev_plant_output;

    if(currentControlMode == AUTO) {
            OSQPost(&PIDQ,
                            (float *) &plant_output,
                            sizeof((void *)&plant_output),
                        OS_OPT_POST_FIFO,
                        &os_err);
    }

    OSQPost(&GUIQ,
                                    (float *) &plant_output,
                                    sizeof((void *)&plant_output),
                                    OS_OPT_POST_FIFO,
                                    &os_err);


    /*
     * In order to print out floating point numbers,
     * go to Project > Properties > C/C++ Build > Settings and
     * change Runtime Library from 'Newlib-nano' to 'Newlib standard'
     */

    // convert plant output to temperature before printing to teraterm
    printf("uC/OS - Plant Input : %0.3f Volts ---------- Plant Output : %0.3f Volts \

    prev_plant_output = plant_output;
```

```
                // 'sample time' (runs every  200 ms)
                OSTimeDlyHMSM(0u, 0u, 0u, 200u,
                                                OS_OPT_TIME_HMSM_STRICT,
                                                &os_err);
        }
}
```

A.0.8 PID Task

```
/*********************************************************************************************
*                                          PID TASK
*
* Description : This task is the PID controller task. It receives a setpoint value from the user a
* The velocity PID algorithm is implemented to ensure a bumpless transfer between manual and auto
*
*        e[n] = r[n] - c[n]
*        m[n] - m[n - 1] = k_p * (e[n] - e[n - 1]) + K_i * (e[n]) + K_d * (e[n - 1] + e[n - 2])
*
*        Where
*        - e[n] is the error
*        - m[n] is the manipulated variable
*        - r[n] is the set point
*
*
* Arguments   : p_arg   is the argument passed to 'PIDTask()' by 'OSTaskCreate()'.
* Returns     : none
* Notes       : 1)         You must tune your PID gains in LabVIEW with the above settings (in Des
*                                      so that those gains will work when used within your PID
*********************************************************************************************
*/
static void PIDTask(void* p_arg) {
        // PID Coefficients
//        float K_p = 1;
//        float T_i =1;
//        float T_d =0;
        float K_p = 1.321;
        float T_i =0.006;
        float T_d =0.002;
        float T_s = 0.200/60;

        float pidInput = 0.0;
        float pidOutput = 0.0;

        // Desired value in the plant (Temperature)
        float setPoint = 20;

        float currentError= 0.0;
        float errorDelta_1 = 0.0;
```

```
float errorDelta_2 = 0.0;
float changeInOutput = 0.0;
float errors[2] = {0.0, 0.0};

void *p_pid_input_msg;
void *p_setpoint_msg;

OS_ERR  os_err;

OS_MSG_SIZE  msg_size;
CPU_TS       ts;

while(DEF_ON) {

        p_setpoint_msg = OSQPend(&SetPointQ,
                                  0,
                                  OS_OPT_PEND_NON_BLOCKING,
                                 &msg_size,
                                 &ts,
                                 &os_err);

        if(msg_size > 0){
                setPoint = *((float *)p_setpoint_msg);
        }


        // Get voltage from plant input queue
        p_pid_input_msg = OSQPend(&PIDQ,
                    0,
                    OS_OPT_PEND_BLOCKING,
                            &msg_size,
                        &ts,
                        &os_err);


        pidInput = *((float *)p_pid_input_msg);

        currentError = TempToVoltage(setPoint) - pidInput;
        errorDelta_1 = currentError - errors[0];
        errorDelta_2 = errors[0] - errors[1];
```

```
                changeInOutput = K_p * (errorDelta_1 + (currentError * T_s/T_i) + (errorDelta_2 *

                pidOutput += changeInOutput;

                errors[1] = errors[0];
                errors[0] = currentError;

                if(currentControlMode == AUTO) {
                        OSQPost(&PlantInputQ,
                                        (float *) &pidOutput,
                                        sizeof((void *)&pidOutput),
                                        OS_OPT_POST_FIFO,
                                        &os_err);
                }

                OSTimeDlyHMSM(0u, 0u, 0u, 200u,
                                                                OS_OPT_TIME_HMSM_STRICT,
                                                                &os_err);
        }
}
```

A.0.9 Helper Functions

```
/*****************************************************************************************
*
*                                                    HELPER FUNCTIONS
*
*****************************************************************************************
*/


static void createAxis(Point frame, int xAxisSize, int yAxisSize) {
        BSP_LCD_DrawHLine(frame.X, frame.Y, xAxisSize);
        BSP_LCD_DrawVLine(frame.X, frame.Y - yAxisSize, yAxisSize);
}

static void drawLine(Point a, Point b) {
        BSP_LCD_DrawLine(a.X, a.Y, b.X, b.Y);
}

static void showText(uint16_t line, char* text) {
        BSP_LCD_ClearStringLine(line);
        BSP_LCD_DisplayStringAtLine((uint16_t) line, text);
}

static void drawReferenceLine(Point p, int distance) {
        int currentColor =  BSP_LCD_GetBackColor();

        BSP_LCD_SetBackColor(LCD_COLOR_BLUE);
        BSP_LCD_DrawHLine(p.X, p.Y, distance);

        BSP_LCD_SetBackColor(currentColor);
}


/*
 * Lab 1 & 2 temp/voltage relationship
 * using polynomial order 2
 */

static float TempToVoltage(float temp){
        return -0.0015*pow(temp,2) + 0.3319*temp - 6.9173;
```

```
}

static float VoltageToTemp(float volt){
        return 0.3053*pow(volt,2) + 2.2602*volt + 25.287;
}




/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct;
  RCC_ClkInitTypeDef RCC_ClkInitStruct;

    /**Configure the main internal regulator output voltage
    */
  __HAL_RCC_PWR_CLK_ENABLE();

  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /**Initializes the CPU, AHB and APB busses clocks
    */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 4;
  RCC_OscInitStruct.PLL.PLLN = 168;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 7;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Initializes the CPU, AHB and APB busses clocks
```

```
    */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

#if 0
  /**Configure the Systick interrupt time
    */
  HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

    /**Configure the Systick
    */
  HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

  /* SysTick_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
#endif
}

/**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM1 interrupt took place, inside
  * HAL_TIM_IRQHandler(). It makes a direct call to HAL_IncTick() to increment
  * a global variable "uwTick" used as application time base.
  * @param  htim : TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM1) {
```

```
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  /* USER CODE END Callback 1 */
}

/**
  * @brief  This function is executed in case of error occurrence.
  * @param  file: The file name as string.
  * @param  line: The line in file as a number.
  * @retval None
  */
void _Error_Handler(char *file, int line)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  while(1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t* file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     tex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/**
```

```
  * @}
  */
#define USE_VCP 1
#define USE_MYMUTEX 0
#if USE_VCP
#ifdef __GNUC__
/* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
set to 'Yes') calls __io_putchar() */
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
int __io_putchar(int ch);
int _write(int file,char *ptr, int len)
{
 int DataIdx;
#if USE_MYMUTEX
 static        OS_MUTEX           MyMutex;
 OS_ERR  os_err;
 CPU_TS  ts;
                // use mutex to protect VCP access
                OSMutexPend((OS_MUTEX *)&MyMutex,
                                (OS_TICK   )0,
                                (OS_OPT    )OS_OPT_PEND_BLOCKING,
                                (CPU_TS   *)&ts,
                                (OS_ERR   *)&os_err);
#endif
 for(DataIdx= 0; DataIdx< len; DataIdx++)
 {
 __io_putchar(*ptr++);
 }
#if USE_MYMUTEX
                OSMutexPost((OS_MUTEX *)&MyMutex,
                                (OS_OPT    )OS_OPT_POST_NONE,
                                (OS_ERR   *)&os_err);
#endif
 return len;
}
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

PUTCHAR_PROTOTYPE
{
```

```
        while(CDC_Transmit_HS((uint8_t *)&ch, 1) != USBD_OK);
        return ch;
}

#else
#ifdef __GNUC__
int _write(int32_t file, uint8_t *ptr, int32_t len)
{
        /* Implement your write code here, this is used by puts and printf for example */
        /* return len; */
        int i;
        for(i=0; i<len; i++)
                ITM_SendChar(*ptr++);
        return len;
}
#endif
volatile int32_t ITM_RxBuffer = ITM_RXBUFFER_EMPTY;

int fputc(int ch, FILE *f) {
  return (ITM_SendChar(ch));
}

int fgetc(FILE *f) {                /* blocking */
  while (ITM_CheckChar() != 1);
  return (ITM_ReceiveChar());
}

#endif


/******END OF FILE****/
```