Conductivity Experiment Control Program

Adriano J. Holanda, Mariana Merino e Marcelo Mulato

November 16, 2010

Abstract

This manuscript describes the specification and implementation of a program to control a conductivity experiment using the equipments Agilent HP 4140B, Agilent HP 34970A and Tektronix PS2511G. The manuscript is divided into introduction, specification and implementation. The introduction gives a general overview of the experiment, the specification formalize the program construction guidelines, and the implementation shows all aspects of the instructions used to realize the specification.

Contents

1 Introduction

1.1 Experiment description

The aim of the experiment is to characterize a sample by variating the temperature in a resistance attached to it and to take the measure of temperature and current in the sample.

The equipment setup can be roughly described as:

Environment heater A *Agilent HP4140B* (Figure 1(a)) device supplies the resistance attached to sample with voltage, heating the environment. The sample and resistance are placed in an closed cylinder and a vacuum is produced by a pump before the experiment is started;

Sample voltage supply A *Tektronix PS2511G* (Figure 1(b)) is connected to the sample to apply a constant voltage and read the resulted current at each time interval;

Temperature reader A *Agilent HP34970A* (Figure 1(c)) data acquisition/switch unit is connected to the sample to read the temperature at each time interval.

(a) Agilent HP 4140B. (b) Tektronix PS2511G. (c) Agilent HP34970A.

Figure 1: Devices

A proportional-integral-derivative control [1] is added to the computer system to limit the error due delays of the rate temperature adjust. This adjust is made in the system when the rate measured during the experiment is different from the desired rate entered by the user.

The experiment is divided in four phases (Figure 2):

1. In the phase 1, all devices are configured according to the experiment requirements entered by the user. At the beginning, the sample is about

[1] This control must be tuned yet to find the best parameters to the experiment, those that maintains the measured temperature rate near from the desired.

the environment temperature. The resistance starts to heat the cylinder environment, and the sample is heated up to maximum temperature specified by the user. After it's been reached the maximum, the system pass to another phase. The purpose of this first heating is to eliminate the impurities in the sample;

2. In the phase 2, liquid nitrogen is added to the cylinder to freeze the sample to a minimum temperature entered by the user. The freezing phase must be controlled by PID to avoid high rates of temperature that can damage the sample;

3. After the minimum temperature is reached, another shift is made and phase 3 has its beginning. The sample is heated up to the maximum temperature again and this phase is the one to be analyzed to characterize the sample. After the maximum is reached again, the freezing comes to the scene again, starting phase 4;

4. In the phase 4, the temperature of the physical system is decreased with liquid nitrogen. The experiment is finished when the environment temperature is reached.

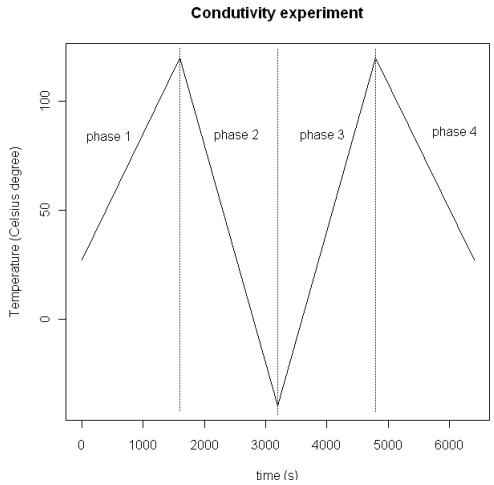

Figure 2: Approximated desired behavior of the conductivity experiment.

1.2 Program description

The Figure 3 presents a screen shot of the graphical interface developed to facilitate and control the data entry. Some default values are presented to the user to help the newbie to know the order of magnitude of each measure.

The experiment parameters to be entered are:

- Maximum temperature;
- Minimum temperature;
- Environment temperature;
- Delta time;
- Set point (desired temperature rate);
- Initial voltage applied in the resistance;
- Constant current applied in the resistance;
- Constant voltage applied in the sample.

  The PID parameter can also be specified by the user because the default ones do not cause any perturbation in the physical system, the terms are:

  - Proportional;
  - Integral;
  - Derivative.

The data retrieved in the experiment are stored in a file to be create the sample characterization graph. The measures of *resistance x time* are placed in a two column table.

The sample resistance is calculated by

$$R = \frac{V}{i} \tag{1}$$

where
V is the voltage applied in the sample;
i is the measured current in the sample.

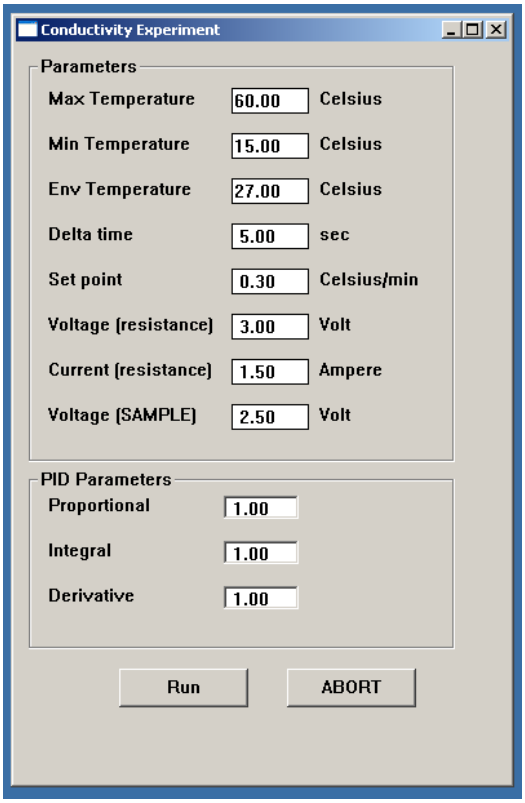

Figure 3: Graphical user interface for the conductivity experiment.

2 Specification

The specification was done using the specification language *Temporal Logic of Actions 2*, or $TLA+^2$ for short, using the TLA toolbox. The $TLA+^2$ language helps in the specification of the behavioral properties of a system using first order logic, temporal logic, sets and functions integrated into the language [1].

The specified modules are:

1. Conductivity experiment aka `exp`;
2. PID Controller aka `pid`;
3. Device interface aka `dev`;
4. User interface aka `ui`;
5. Real Time.

The code are modularized according to the functions needed by the system. `exp` module controls the experiment, `pid` offers PID functions to adjust the desired experiment behavior, `dev` handles communication with the equiments and `ui` is the interface to the user. `Real Time` module contains the specification of `delay` function, used to make the system wait a specified time until the next operation to be executed.

The module specifications are listed in the next pages, with the name of each module in the header.

MODULE *ConductivityExperiment*

EXTENDS *Reals*, *Sequences*, *DeviceInterface*, *RealTime*

VARIABLES

parameter, Parameters for the experiment
pidParam,
pidState,
measure,
phase

The experiment is divided in 4 phase:
0. Heat the sample to eliminate impurities;
1. Freeze the sample to prepare to another heating;
2. Anothe heating, but now it is the phase that is the main part of the experiment because the sample characterization will be made based on the data measured in this phase;
3. Freeze the sample, stopping the experiment when the environment temperature is reached.

Possible values of device status

$OFF \triangleq \{0\}$
$ON \triangleq \{1\}$
$Status \triangleq \{OFF, ON\}$

A sample measure consists of a temperature and current tuple. Even though temperature is used only to control the limits of the experiment, it is important to store it because the *PID* control is done based on temperature data history. But the main measure of the experiment is the current that is used to know the sample resistence using Ohm's law, $U = Ri \Rightarrow R = U/i$, or its condutivity that is the inverse of the resistence.

$SampleMeasure \triangleq [Temp : Real, cur : Real]$

Time is associated with the measures.

$SerieMeasure \triangleq [Time \to SampleMeasure]$

setpoint − Rate at which temperature must varies. The controller tries to maintain the measure Minimum temperature to reach in the experiment Minimum temperature to reach in the experiment

$PIDCon \triangleq$ INSTANCE *PIDController*
WITH $state \leftarrow pidState, parameter \leftarrow pidParam$

ParamLabel are the set of labels used for parameters that will be entered by the user.
TEMP_MAX − Temperature maximum to be reached
TEMP_MIN − Temperature minimum to be reached
TEMP_ENV − Temperature of the environment *SET_POINT* − Rate at which the temperature varies per time *DELTA_TIME* − *Time* interval at which the measures are read
VOLT_RES − Voltage to be applied in the resistence
CURR_RES − Current to be applied to resistence *VOLT_SAMPLE* − Voltage to be applied to sample

$ParamLabel \triangleq \{$"TEMP_MAX", "TEMP_MIN", "TEMP_ENV", "SET_POINT", "DELTA_TIME", "VOLT_RES", "CURR_RES", "VOLT_SAMPLE"$\}$

$ExpDev \triangleq [label \mapsto$ "HP34970A",

$busLabel \mapsto$ "gpib0,9",
$id \mapsto iOpen($"gpib0,9", $devInt,\ devInt')]$
$\cup$
$[label \mapsto$ "PS2511G",
$busLabel \mapsto$ "gpib0,8",
$id \mapsto iOpen($"gpib0,8", $devInt,\ devInt')]$
$\cup$
$[label \mapsto$ "HP4140B",
$busLabel \mapsto$ "gpib0,17",
$id \mapsto iOpen($"gpib0,17", $devInt,\ devInt')]$

$\land\ parameter.Tmax = 100 \land parameter.Tmin = 15 \setminus *$ default arbitrary
$\land\ devParam.Vres = 3 \land devParam.ires = 1.5 \land devParam.Vsample = 2.5$

ASSUME $(Time \in Real) \land (Time > 0)$

$CEInvariant \triangleq$
$\land\ PIDCon!PIDInvariant$
$\land\ parameter \in [ParamLabel \to Real \cup pidParam]$
$\land\ devInt \in [Dev \to InitDevInt]$
$\land\ measure \in Measure$
$\land\ phase \in \{0, 1, 2, 3\}$ the experiment has 4 phases

---

$CEInit \triangleq$
$\land\ PIDCon!PIDInit$
$\land\ devInt \in InitDevInt$
$\land\ measure = [Time \mapsto \langle 0, 0, 0\rangle]$ initialize sample measures to zero
$\land\ phase = 0$

$\land\ userInt \in InitUserInt$

Voltage
$ReadVoltage(d) \triangleq$ Voltage
$\exists\, v \in Val :$
$\land\ d \in \{ExpDev[$"PS2511G"$], ExpDev[$"HP4140B"$]\}$
$\land\ Send(d.id,$ "SOURce:VOLTage?", $NoVal,\ devInt,\ devInt')$
$\land\ Read(d.id,\ v,\ devInt,\ devInt')$

Current
$ReadCurrent(d) \triangleq$ Current
$\{i \in Val :$
$\land\ d \in \{ExpDev[$"PS2511G"$], ExpDev[$"HP4140B"$]\}$
$\land\ Send(d.id,$ "SOURce:CURRent?", $NoVal,\ devInt,\ devInt')$
$\land\ Read(d.id,\ i,\ devInt,\ devInt')\}$

$ReadTemperature \triangleq$
$\{T \in Val :$
$\land Send(ExpDev[\text{“HP34970A”}].id,\ \text{“SOURce:TEMPerature?”},\ NoVal,\ devInt,\ devInt')$
$\land Read(ExpDev[\text{“HP34970A”}].id,\ T,\ devInt,\ devInt')\}$

$TemperatureConstraint \triangleq$
$\land parameter.Tmin < parameter.Tmax$

Perform sample measurements
$PerformSampleMeasurement \triangleq$
$\land \{sm \in SampleMeasure :\ \land sm.T = ReadTemperature$
$\land sm.i\ = ReadCurrent(ExpDev[\text{“HP4140B”}])\}$

Initialization
$InitializeSystems(Vres,\ i,\ Vs) \triangleq$
Set the unit temperature
$\land Send(ExpDev[\text{“HP34970A”}].id,\ \text{“UNIT:TEMPerature”},\ \text{“C”},\ devInt,\ devInt')$

Set the voltage in the resistence
$\land Send(ExpDev[\text{“HP4140B”}],\ \text{“SOURce:VOLTage”},$
$parameter[\text{“VOLT\_RES”}],\ devInt,\ devInt')$

Set the current in the resistence
$\land Send(ExpDev[\text{“HP4140B”}],\ \text{“SOURce:VOLTage”},$
$parameter[\text{“CURR\_RES”}],\ devInt,\ devInt')$

Set voltage in the sample
$\land Send(ExpDev[\text{“PS2511GB”}],\ \text{“SOURce:VOLTage”},$
$parameter[\text{“VOLT\_SAMPLE”}],\ devInt,\ devInt')$

Turn on the devices
$\land \forall\, d \in ExpDev :$
$\land Send(d.id,\ \text{“OUTPut:STATe”},\ ON,\ devInt,\ devInt')$ Turn on the device
$OUTP$:$STAT$ enables = 1 or disable = 0 the output of power supply.

$\land$ UNCHANGED $\langle measure,\ phase,\ parameter\rangle$

$Halt \triangleq$
$\land \forall\, d \in ExpDev :$
Turn off devices
$\land Send(d.id,\ \text{“OUTPut:STATe”},\ OFF,\ devInt,\ devInt')$
Close sessions
$\land d.id' = iClose(ExpDev.busLabel,\ devInt,\ devInt')$
$\land$ UNCHANGED $\langle measure,\ phase\rangle$

$DoPerodicMeasures \triangleq$
$\land \exists\, m \in SerieMeasure,\ rate,\ err,\ Tenv \in Real :$

$\land\ m.t = 0$ Initialize t
$\land\ m.t = PerformSampleMeasurement$ Get the first measure in $t0$
$\land\ Tenv = m.t.T$
Environment temperature: could be the first sample measure or set. Environment temperature is set the first measure of sample temperature
$\land$ LET $DPM[t \in Time] \triangleq$
$\land\ Wait(parameter[\text{“DELTA\_TIME”}])$
$\land\ t' = t + parameter[\text{“DELTA\_TIME”}]$
$\land\ m.t = PerformSampleMeasurement$
$\land\ measure \cup m$
Current measure were calculated and insert into the set. Becomes persistent in the implementation
$\land$ IF $(m.t.T > parameter[\text{“TEMP\_MAX”}] \land (phase = 0 \lor phase = 2)) \lor$
$(m.t.T < parameter[\text{“TEMP\_MIN”}] \land phase = 1)$ THEN
$\land\ phase' = phase + 1 \land DPM[t]$
Converse the effect of rate
$\land\ parameter[\text{“SET\_POINT”}]' = -\ parameter[\text{“SET\_POINT”}]$
ELSE IF $m.t.T < parameter[\text{“TEMP\_ENV”}] \land phase = 3$ THEN
$Halt$
ELSE $\land\ rate = m.t.T / parameter[\text{“DELTA\_TIME”}]$
$\land\ err = parameter[\text{“SET\_POINT”}] - rate$
$Vnew = Vold * (setpt/PIDadjust)$
$\land\ parameter[\text{“VOLT\_RES”}]' =$
$parameter[\text{“VOLT\_RES”}] *$
$(parameter[\text{“SET\_POINT”}]/$
$PIDCon!PIDOutput(err,\ parameter[\text{“DELTA\_TIME”}]))$
$\land\ Send(ExpDev[\text{“PS2511G”}],\ \text{“SOURce:VOLTage”},$
$parameter.Vres,\ devInt,\ devInt')$
$\land\ DPM[t]$
IN $DPM[0]$

$RunExperiment \triangleq$
$\land \forall\, l \in \{\text{“VOLT\_RES”},\ \text{“CURR\_RES”},\ \text{“VOLT\_SAMPLE”}\} :$
$\land\ parameter.l \in Real \land parameter.l > 0$
$\land\ InitializeSystems(parameter[\text{“VOLT\_RES”}],$
$parameter[\text{“CURR\_RES”}],$
$parameter[\text{“VOLT\_SAMPLE”}])$
Pseudo arbitrary initial numbers, must be tested
$\land\ parameter[\text{“DELTA\_TIME”}] \in Time \land DoPerodicMeasures$

$Next \triangleq \lor \exists\, d \in ExpDev :\ \lor ReadVoltage(d) \lor ReadCurrent(d)$
$\lor\ ReadTemperature \lor PerformSampleMeasurement$
$\lor\ DoPerodicMeasures \lor RunExperiment$
$\lor \exists\, V,\ i \in Measure : InitializeSystems(V,\ i,\ V)$

$$Spec \triangleq CEInit \wedge \Box[Next]_{\langle parameter,\, measure,\, phase\rangle}$$

$$\text{THEOREM } Spec \Rightarrow \Box(CEInvariant \wedge TemperatureConstraint)$$

MODULE *PIDController*

EXTENDS *Reals*, *RealTime*

VARIABLES

parameter, Parameters are settings that persist over time

state State variables change sample changes sample to sample, depending on what happens in the feedback

$ParamLabel \triangleq \{\text{“Proportional”}, \text{“Integral”}, \text{“Derivative”}\}$

The idea to pack the variables in the *pidParams* and *pidState* records was “adapted” from http://*www.mstarlabs.com/apeng/*techniques/*pidsoftw.html*

$PIDInvariant \triangleq$

$\land\ parameter \in [ParamLabel \to Real]$

$\land\ state \in [$
$\quad integral : Real,$ The summation of the error
$\quad prevErr : Real$ The previous error
$\quad]$

$PIDInit \triangleq$

$\land\ parameter.Kp = 1 \land parameter.Ki = 1 \land parameter.Kd = 1$
$\land\ state.integral = 0 \land state.prevErr = 0$

Functions to calculate the *PID* terms

$calcP(e) \triangleq$ Proportional term
$\land\ \{p \in Real : \land\ p = parameter[\text{“Proportional”}] * e\}$
$\land$ UNCHANGED $\langle parameter, state\rangle$

$calcI(e, dt) \triangleq$ Integral term
$\land\ state.integral' = state.integral + e$ update the integral
$\land\ \{p \in Real : \land\ p = parameter[\text{“Integral”}] * state.integral\}$
$\land$ UNCHANGED $parameter$

$calcD(e, dt) \triangleq$ Derivative term
$\land\ \{p \in Real : \land\ p = parameter[\text{“Derivative”}] * ((e - state.prevError)/dt)$
$\qquad\qquad\ \land\ state.prevError' = e\}$
$\land$ UNCHANGED $parameter$

$PIDOutput(e, dt) \triangleq$ Calculate the pid output in one pass
$\land \{\exists\, u \in Real : u = calcP(e) + calcI(e, dt) + calcD(e, dt)\}$

$PIDNext \triangleq$

$\lor (\exists\, e \in Real, dt \in Time : \lor\ calcP(e) \lor calcI(e, dt)$
$\qquad\qquad\qquad\qquad\qquad\qquad \lor\ calcD(e, dt) \lor PIDOutput(e, dt))$

$PIDSpec \triangleq$

$\land\ PIDInit \land \Box[PIDNext]_{\langle parameter, state\rangle}$

THEOREM $PIDSpec \Rightarrow \Box PIDInvariant$

MODULE *DeviceInterface*

VARIABLE *devInt*

CONSTANTS *Send*(_, _, _, _, _), A *Send*(*d.id*, *c*, *v*, *devInt*, *devInt′*) step represents controller p sending to device "*d*" with session "*id*" a command "*c*" with value "*v*"

Read(_, _, _, _), A *Read*(*d.id*, *c*, *devInt*, *devInt′*) step represents controller p requesting a response from device "*d*" with session "*id*" storing into buffer "b"

iOpen(_, _, _), *iOpen*(*d.busLabel*, *devInt*, *devInt*) Open a session to device *d.busLabel*

iClose(_, _, _), *iClose*(*d.busLabel*, *devInt*, *devInt*) Close a session to device *d.busLabel*

Com, The set of commands

Val, The set of possible values

Session, Session to be opened to device

InitDevInt The set of possible values of *devInt*

ASSUME $\forall\ d, c, v, diOld, diNew :$

$\wedge\ Send(d, c, v, diOld, diNew) \in$ BOOLEAN

$\wedge\ Read(d, v, diOld, diNew) \in$ BOOLEAN

$Dev \triangleq [devLabel : \text{STRING}, busLabel : \text{STRING}, id : Session]$ The set of device identifiers

$Unit \triangleq \{\text{"Ampere"}, \text{"Celsius degree"}, \text{"Volts"}\}$

$Measure \triangleq [val : Val, un : Unit]$

$MReq \triangleq [op : \{\text{"Snd"}\}, d : Dev, com : Com, val : Val] \cup [op : \{\text{"Rd"}\}, dev : Dev, com : Com]$

$NoCom \triangleq$ CHOOSE $c : c \notin Com$ An arbitrary command not in *Com*

$NoVal \triangleq$ CHOOSE $v : v \notin Val$ An arbitrary value not in *Val*

$NoDev \triangleq$ CHOOSE $d : d \notin Dev$

$NoSes \triangleq$ CHOOSE $s : s \notin Session$

MODULE *GraphicalUserInterface*

EXTENDS *ConductivityExperiment*

VARIABLES *runButton*,
abortButton,
param, *expParams* ∪ *pidParams*
entry

CONSTANTS *Widget*, Generic *UI* component
InitUserInt The set of possible values of *userInt*

$OneOf(s) \triangleq \{\langle s[i]\rangle : i \in \text{DOMAIN } s\}$
$ValidString \triangleq OneOf(\text{"0123456789"}) \cup \{\text{"."}\}$
$EmptyString \triangleq \{\text{""}\}$
$EntryText \triangleq [class : Widget,\ val : ValidString \cup EmptyString]$
$Button \triangleq [class : Widget,$
$action : \{\text{"released"},\ \text{"pressed"}\},$
$state : \{\text{"enabled"},\ \text{"disabled"}\}]$

$GUIInvariant \triangleq$
$\land\ runButton \in Button$
$\land\ abortButton \in Button$
$\land\ entry \in [ParamLabel \to EntryText]$

$GUIInit \triangleq$
$\land\ CEInit$
$\land\ param = parameter \cup pidParam$ *expParam + pidParam*
$\land\ \forall\, b \in \{runButton,\ abortButton\} : b.action = \text{"released"}$
All buttons are released
$\land\ \forall\, e \in entry : e.val = EmptyString$
All entry texts are empty

$Run \triangleq$ I don't know how to treat events in the spec.
$\land\ runButton.action = \text{"pressed"} \land abortButton.action = \text{"released"}$
$\land\ runButton.state = \text{"disabled"}$ Disable to avoid multiple or successive clicks
$\land\ \forall\, e \in entry : e.val = ValidString$
$\land\ \forall\, l \in ParamLabel : param.l = entry.l.val$ No types at all, ok! Test?
$\land\ RunExperiment$ Run experiment has no arguments. Has it become obscure?

$Abort \triangleq$ events again
$\land\ abortButton.action = \text{"pressed"} \land runButton.action = \text{"released"}$
$\land\ runButton.state = \text{"enabled"}$ Enable run button
$\land\ Halt$

$GUINext \triangleq Run \lor Abort$

$GUISpec \triangleq GUIInit \land \Box[GUINext]_{\langle runButton,\ abortButton,\ entry\rangle}$

THEOREM $GUISpec \Rightarrow GUIInvariant$

MODULE *RealTime*

EXTENDS *Reals*
CONSTANT $Time$,
$Wait(\_)$ $Wait(t)$ elapse $t \in Time$
VARIABLE now

ASSUME $(Time \geq 0) \land (Time \in Real)$

$RTBound(A, v, D, E) \triangleq$
LET $TNext(t) \triangleq t' = \text{IF } \langle A \rangle_v \lor \neg(\text{ENABLED } \langle A \rangle_v)' \text{ THEN } 0 \text{ ELSE } t + (now' - now)$

$Timer(t) \triangleq (t = 0) \land \Box[TNext(t)]_{\langle t, v, now \rangle}$

$MaxTime(t) \triangleq \Box(t \leq E)$

$MinTime(t) \triangleq \Box[A \Rightarrow t \geq D]_v$
IN $\exists\, t : Timer(t) \land MaxTime(t) \land MinTime(t)$

$RTnow(v) \triangleq$ LET $NowNext \triangleq \land now' \in \{r \in Real : r > now\} \land \text{UNCHANGED } v$
IN $\land now \in Real$
$\land \Box[NowNext]_{now}$
$\land \forall\, r \in Real : \mathrm{WF}_{now}(NowNext \land (now' > r))$

3 Implementation

The implementation was made in an interleaving way with the specification, cause some understanding was becoming strong when the implementation aspects were faced.

The C language was the choice to implement the computer system and the documentation of each module was weaved with the code and extracted with the help of CWEB program. This programming methodology is known as literate programming and was invented by Donald Knuth to help him document TEX. The details can be found in [2].

The module implementations in `C` language are listed in the next pages, with the name of each module in the header.

1. Header.

```
⟨ exp.h 1 ⟩ ≡
#ifndef EXP_INCLUDED
#define EXP_INCLUDED
#define EXP_MODULE_NAME "Conductivity␣Experiment"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include "dev.h"
#include "pid.h"
#define LOG(m) ( ( fprintf (stderr, "%s:%d:␣LOG:␣%s\n", 39
  __FILE__, (int) __LINE__, #m ) ) )
#define PARAMETER_CLASSES 2
  ⟨ Assert declaration for experiment module 2 ⟩
  ⟨ Definitions or also called macros 9 ⟩
  ⟨ Type definitions 4 ⟩
  extern void EXP_run_experiment( );
  extern bool EXP_is_limit_constraint_satisfied( );
  extern void EXP_set_parameter_value( );
#endif /* EXP_INCLUDED */
```

See also section 6.

2. The defined assertion allows that all procedures needed to halt the systems used in the experiment may be turn off correctly before the system exits. The *assert*(*e*) uses a left to right evaluation trick to compact the statement without the need to use conditionals. C evaluates the second part of a *OR* ("||") statement only if the first part is `false`, and `HALT` will be used only if the assertion fails.

```
⟨ Assert declaration for experiment module 2 ⟩ ≡
#undef assert
#ifdef NDEBUG
#define assert(e) ((void) 0)
#else
  void assert(int e);
  extern int EXP_halt(void);
#define HALT (EXP_halt( ))
#define assert(e) ((void)((e) ∨ (HALT)))
#endif
```

This code is used in section 1.

3. Experiment.

⟨ Preprocessor definitions ⟩

```
#include "exp.h"
```

⟨ Global variables 7 ⟩

⟨ Internal functions 12 ⟩

⟨ Functions 11 ⟩

4. Measure types.

```
#define NDEVICES 3
```

⟨ Type definitions 4 ⟩ ≡

```
enum measure_type_enum {
  VOLT, TEMP, CURR
};
```

See also sections 5 and 8.

This code is used in section 1.

5. Phases.

⟨ Type definitions 4 ⟩ +≡

```
typedef enum phases_enum {
  PHASE1, PHASE2, PHASE3, PHASE4
} PHASES;
```

6. Macros

⟨ exp.h 1 ⟩ +≡

```
#define EXP_PARAMETER_LABEL "Parameters"
#define EXP_PARAMETERS 4 /* Set the initial default values of the devices. */
  /* RESISTANCE */
#define EXP_DEFAULT_INITIAL_RES_VOLTAGE 3 /* Volts */
#define EXP_DEFAULT_RES_CURRENT 2 /* Ampere */ /* SAMPLE */
#define EXP_DEFAULT_SAMPLE_VOLTAGE 2.5 /* Volts */
#ifdef EXP_NDEBUG
#define EXP_DEBUG(m) ((void) 0)
#else
#define EXP_DEBUG(m) (fprintf(stderr, m))
#endif
#define EXP_MAX_VOLTAGE 100.0_F
#define EXP_MIN_VOLTAGE DEV_MAX_VOLTAGE
#define EXP_VOLTAGE_ACCURACY 0.025_F
```

7. ⟨ Global variables 7 ⟩ ≡

```
static Dev_Info devices[3] = {[VOLT] = {"PS2511G", "gpib0,8",
    "Tektronix␣PS2511G␣Programmable␣Power␣Supply", 0}, [TEMP] = {"HP34970A", "gpib0,9",
    "HP␣34970A␣Data␣Acquisition/Switch␣Unit", 0}, [CURR] = {"HP4140B", "gpib0,17",
    "HP␣4140B␣pA␣Meter/DC␣Voltage␣Source", 0}, };
```

See also section 10.

This code is used in section 3.

8. Experiment parameters.

```
⟨ Type definitions 4 ⟩ +≡
  typedef enum {
    MAX_TEMPERATURE,      /* maximum temperature */
    MIN_TEMPERATURE,      /* minimum temperature */
    ENV_TEMPERATURE,      /* environment temperature */
    DELTA_TIME,      /* time to wait until next read */
    SET_POINT,      /* Rate */
    VOLT_RES,      /* voltage to be applied in the resistance */
    CURR_RES,      /* current to be applied in the sample */
    VOLT_SAMPLE      /* voltage applied in the sample */
  } EXP_PARAMS_ENUM;
```

9. Default values are set if the user does not furnish values to main parameters.

```
⟨ Definitions or also called macros 9 ⟩ ≡
#define EXP_PARAMETERS_N  8
```

This code is used in section 1.

10. Limits. Enumeration of the limits specified in the header are declared to facilitate the elements transversal.

```
⟨ Global variables 7 ⟩ +≡
  float exp_values[EXP_PARAMETERS_N] = {[MAX_TEMPERATURE] = 60.0_F,
  [MIN_TEMPERATURE] = 15.0_F,
  [ENV_TEMPERATURE] = 27.0_F,      /* Celsius degree */
  [DELTA_TIME] = 5.0_F,      /* seconds */
  [SET_POINT] = 0.3_F,      /* Celsius per minute */
  [VOLT_RES] = 3.0_F,      /* Volt */
  [CURR_RES] = 1.5_F,      /* Ampere */
  [VOLT_SAMPLE] = 2.5_F      /* Volt */
  }; Labeled_Value exp_labeled_values[EXP_PARAMETERS_N] = { [MAX_TEMPERATURE] =
  { . label = "Max␣Temperature" , . value = &exp_values[MAX_TEMPERATURE] } ,
  [MIN_TEMPERATURE] =
  { . label = "Min␣Temperature" , . value = &exp_values[MIN_TEMPERATURE] } ,
  [ENV_TEMPERATURE] =
  { . label = "Env␣Temperature" , . value = &exp_values[ENV_TEMPERATURE] } ,
  [DELTA_TIME] =
  { . label = "Delta␣time" , . value = &exp_values[DELTA_TIME] } ,
  [SET_POINT] =
  { . label = "Set␣point" , . value = &exp_values[SET_POINT] } ,
  [VOLT_RES] =
  { . label = "Voltage␣(resistance)" , . value = &exp_values[VOLT_RES] } ,
  [CURR_RES] =
  { . label = "Current␣(resistance)" , . value = &exp_values[CURR_RES] } ,
  [VOLT_SAMPLE] =
  { . label = "Voltage␣(SAMPLE)" , . value = &exp_values[VOLT_SAMPLE] }
  } ;
  float exp_limit_values[EXP_PARAMETERS_N] = {[MAX_TEMPERATURE] = 120.0_F,
  [MIN_TEMPERATURE] = −50.0_F,
  [ENV_TEMPERATURE] = 50.0_F, [DELTA_TIME] = 10.0_F,
      [SET_POINT] = 10.0_F, [VOLT_RES] = 50.0_F, [CURR_RES] = 25.0_F, [VOLT_SAMPLE] = 50.0_F, };
      Parameter EXP_params[EXP_PARAMETERS_N] = { { &exp_labeled_values[MAX_TEMPERATURE]
      , . unit = "Celsius" , . limit = &exp_limit_values[MAX_TEMPERATURE]
      } , { &exp_labeled_values[MIN_TEMPERATURE] , . unit = "Celsius" , .
      limit = &exp_limit_values[MIN_TEMPERATURE] } , { &exp_labeled_values[ENV_TEMPERATURE]
      , . unit = "Celsius" , . limit = &exp_limit_values[ENV_TEMPERATURE] } , {
      &exp_labeled_values[DELTA_TIME] , . unit = "sec" , . limit = &exp_limit_values[DELTA_TIME] } , {
      &exp_labeled_values[SET_POINT] , . unit = "Celsius/min" , . limit = &exp_limit_values[SET_POINT]
      } , { &exp_labeled_values[VOLT_RES] , . unit = "Volt" , . limit = &exp_limit_values[VOLT_RES] } , {
      &exp_labeled_values[CURR_RES] , . unit = "Ampere" , . limit = &exp_limit_values[CURR_RES] } , {
      &exp_labeled_values[VOLT_SAMPLE] , . unit = "Volt" , . limit = &exp_limit_values[VOLT_SAMPLE] } } ;
```

11. The functions are used to handle global variables.

⟨ Functions 11 ⟩ ≡

```
void EXP_set_parameter_value(exp_params_enum, value)
    EXP_PARAMS_ENUM exp_params_enum;
    float value;
{
  exp_values[exp_params_enum] = value;
}
```

See also sections 13, 22, and 23.

This code is used in section 3.

12. Instrumentation. The function *initialize_devices* is responsible to turn on the devices and send initial values before starting running the experiment. The sequence are:
O pen the session to the device furnishing its bus address and getting the id to send commands to;

```
⟨ Internal functions 12 ⟩ ≡
  void open_device_sessions( )
  {
    int i = 0;
    for (i = 0; i ≤ NDEVICES; i++) devices[i].id = DEV_open_session(devices[i].gpib_addr);
  }
```

This code is used in section 3.

13. Run.

```
⟨ Functions 11 ⟩ +≡
  void EXP_run_experiment( )
  {
    long time_sum = 0;      /* time sumation */
    int phase = PHASE1;      /* Initialize phase */
    int reads = 0;      /* number of reads */      /* Voltage applied to resistance */
    float Vres = EXP_DEFAULT_INITIAL_RES_VOLTAGE;
    float Told = 0.0F, Tnow = 0.0F;      /* temperature past and "present" */
    float current = 0.0F;      /* Ampere */
    float rate = 0;      /* Variation at a time interval */
    float err = 0.0F;      /* Error */      /* sample resistance to be calculated */
    float sample_res = 0.0F;      /* PARAMETERS SET */
    float setpt = exp_values[SET_POINT];
    float delta_time = exp_values[DELTA_TIME];
    ⟨ Local run declarations 19 ⟩
    ⟨ Initialize devices 14 ⟩
    ⟨ Initialize IO system 20 ⟩
    Told = DEV_read_temperature(devices[TEMP].id);
    do {
      delay(delta_time);
      time_sum += delta_time;
      Tnow = DEV_read_temperature(devices[TEMP].id);
      current = DEV_read_current(devices[CURR].id);
      reads++;      /* Calculate the sample resistance */
      assert(current > 0);
      sample_res = EXP_DEFAULT_SAMPLE_VOLTAGE/current;
      fprintf(data_out, "%ld\t%6.3f\n", time_sum, sample_res);
      ⟨ Make PID adjust 16 ⟩
      ⟨ Evaluate the need to change the phase 17 ⟩
      ⟨ Finish the experiment? 18 ⟩
    } while (1);      /*HALTING */
    ⟨ Close IO resources 21 ⟩
    EXP_halt( );
  }
```

14. ⟨ Initialize devices 14 ⟩ ≡

```
open_device_sessions( );
⟨ Turn on the resistance power supply device 15 ⟩
DEV_set_voltage(devices[CURR].id, EXP_DEFAULT_SAMPLE_VOLTAGE);
  /* Setup initial commands to the device responsible to acquire data of temperature measures */
DEV_setup_temperature_device(devices[TEMP].id);
```

This code is used in section 13.

15. Turn on the power supply attached to resistance. Notice that only current is set to default value. Voltage is the parameter that is going to change during the experiment.

⟨ Turn on the resistance power supply device 15 ⟩ ≡

```
assert(devices[VOLT].id);
DEV_set_voltage(devices[VOLT].id, Vres);
DEV_set_current(devices[VOLT].id, EXP_DEFAULT_RES_CURRENT);
DEV_set_state(devices[VOLT].id, ON);
```

This code is used in section 14.

16. The *rate* at which temperature varies in a specified *delta_time* is calculated subtracting from the temperature *Told* measured before current measure *Tnow*, and dividing by the elapsed time *delta_time*. The error *err* is the difference between the desired rate and the rate obtained. The correction based on error *err* and *delta_time* using *PID_adjust* is done, and the new voltage *Vnew* to be applied at the resistance is set in the device. The new voltage is calculated as $V_{res} = V_{res} \frac{(setpt - PID_{out})}{setpt}$.

⟨ Make PID adjust 16 ⟩ ≡

```
rate = (Tnow − Told)/(delta_time * 60);      /* Convert time to minute: oC/min */
err = rate − setpt;
Vres = Vres * ((setpt − PID_output(err, delta_time * 60)/setpt));
if (Vres < 0) {
  Vres = 0;
}
else if (Vres > EXP_MAX_VOLTAGE) {
  Vres = EXP_MAX_VOLTAGE;
}
DEV_set_voltage(devices[VOLT].id, Vres);      /* Set the new voltage */
```

This code is used in section 13.

17. The experiment has four phases, the changing in phase is triggered four times according the temperature measured: two for maximum temperature is reached, one for minimum temperature, and one environment temperature. The main point to be stressed is when the experiment is in the decreasing phase, the set point or rate temperature must have a negative value because the value of temperature in time decreases.

The control at decreasing phase is somewhat difficult because the freezing rate is hard to manipulate. The nitrogen put in the recipient containing the sample freezes it in a way difficult to get control. But the PID control is maintained in this phase to avoid high rates of freezing that can cause damage in the sample.

⟨ Evaluate the need to change the phase 17 ⟩ ≡

```
if ((Tnow ≥ exp_values[MAX_TEMPERATURE] ∧ (phase ≡ PHASE1 ∨ phase ≡ PHASE3)) ∨ (Tnow ≤
      exp_values[MIN_TEMPERATURE] ∧ phase ≡ PHASE2)) {
  phase++;
  setpt = −setpt;
}
```

This code is used in section 13.

18. When the experiment is in the fourth phase, in the freezing phase again, but only to allow the retrieval of the material tested, the freezing stops after the first value measured is reached again, and this value is about the environment variable.

```
⟨ Finish the experiment? 18 ⟩ ≡
  if (Tnow ≤ *(float *) EXP_params[ENV_TEMPERATURE].lbvalue→value ∧ phase ≡ PHASE4) {
    break;
  }
```

This code is used in section 13.

19. The data is written in a file during the experiment. The name of the file has the format `data-year_month_day-hour_minute.dat` to help the user to know the file to be worked according to the time when the experiment were performed.

```
⟨ Local run declarations 19 ⟩ ≡
  FILE *data_out;
  char data_fn[32];      /* data filename */
  struct tm *timeinfo;
  time_t cur_time;
```

This code is used in section 13.

20.

```
⟨ Initialize IO system 20 ⟩ ≡
  time(&cur_time);      /* get the current time */
  timeinfo = localtime(&cur_time);
  sprintf(data_fn, "data-%d_%d_%d-%d-%d.dat", timeinfo→tm_year + 1900,      /* year */
  timeinfo→tm_mon + 1,      /* month */
  timeinfo→tm_mday,      /* day */
  timeinfo→tm_hour,      /* hour */
  timeinfo→tm_min      /* minutes */
  );
  data_out = fopen(data_fn, "w");
  if (data_out ≡ Λ) {
    fprintf(stderr, "Could␣not␣open␣%s␣to␣write␣data.\n", data_fn);
    exit(EXIT_FAILURE);
  }
```

This code is used in section 13.

21.

```
⟨ Close IO resources 21 ⟩ ≡
  if (data_out ≠ Λ) fclose(data_out);
```

This code is used in section 13.

22. When the system must be halted, all the device handles and files are closed.

```
⟨ Functions 11 ⟩ +≡
  int EXP_halt(void)
  {
    int i;
    for (i = 0; i ≤ CURR; i++) {
      DEV_close_session(devices[i].id);
    }
    return EXIT_SUCCESS;
  }
```

23. Constraints. Some preconditions are needed to test to improve the consistency of the experiment and to avoid device damage caused by a value set above the device can afford.

```
⟨ Functions 11 ⟩ +≡
  bool EXP_is_limit_constraint_satisfied(exp_params_enum, value)
      EXP_PARAMS_ENUM exp_params_enum;
      float value;
  {
    if (exp_params_enum ≡ MIN_TEMPERATURE) {
      if (value < exp_limit_values[exp_params_enum]) return false;
      ⟨ Verify if minimum temperature is greater the maximum 24 ⟩
    }
    else {
      if (value > exp_limit_values[exp_params_enum]) return false;
    }
    return true;
  }
```

24. In this fragment I must confess that it is a dangerous place to put this constraint verification. The constraint is that the minimum temperature must have a value that is less than the value of the maximum temperature. The place the constraint was inserted fit with the sequence of constraint checks and parameter value sets in the `win32ui.w`.

```
⟨ Verify if minimum temperature is greater the maximum 24 ⟩ ≡
  if (value > exp_values[MAX_TEMPERATURE]) return false;
```

This code is used in section 23.

25. Index.

Conductivity Experiment Module

November 16, 2010 at 10:59

1. Header.

⟨ dev.h 1 ⟩ ≡
```
#ifndef DEV_INCLUDED
#define DEV_INCLUDED
#include <stdlib.h>
#include <assert.h>
#include <sicl.h>
#include <float.h>
  ⟨ Type definitions 4 ⟩
  inline float read_dev( );
  inline void send_cmd( );
  extern inline void DEV_reset( );
  extern inline INST DEV_open_session( );
  extern inline void DEV_close_session( );
  extern inline float DEV_read_state( );
  extern inline void DEV_set_state( );
  extern inline float DEV_read_current( );
  extern inline void DEV_set_current( );
  extern inline float DEV_read_voltage( );
  extern inline void DEV_set_voltage( );
  extern inline float DEV_read_temperature( );
  extern inline void DEV_reset_resistance_voltage( );
  extern inline void DEV_setup_temperature_device( );
#endif    /* DEV_INCLUDED */
```

2. The C `dev.c` has the following structure and content:

⟨ Preprocessor definitions ⟩
```
#include "dev.h"
```
⟨ Generic communication subroutines 6 ⟩

⟨ Specialized device communication subroutines 11 ⟩

3. If not reassigned, default session timeout value.

```
#define DEV_SESSION_TIMEOUT 1000
```

4. Device status is an important mechanism to make the stops the device feed. *Dev_Status* conform to standard setting 0 to `OFF` and 1 to `ON` like SCPI standard recommends.

⟨ Type definitions 4 ⟩ ≡
```
  typedef enum dev_status {
    OFF, ON
  } Dev_Status;
```
See also section 5.

This code is used in section 1.

5. *Dev_Info* groups information about device in a structure and reduce the chance to commit an error due wrong variable access.

```
⟨ Type definitions 4 ⟩ +≡
  typedef struct dev_info {
    char *label;
    char *gpib_addr;
    char *desc;
    INSTid;
  } Dev_Info;
```

6. Read device according to the SCPI command

```
⟨ Generic communication subroutines 6 ⟩ ≡
  inline float read_dev(INSTid, char *scpi)
  {
    char buf[255];
    iprintf(id, scpi);
    return atof(buf);
  }
```

See also sections 7, 8, 9, 10, 13, and 14.

This code is used in section 2.

7. Set the specified device *id* according to the *scpi* command to value *val*.

```
⟨ Generic communication subroutines 6 ⟩ +≡
  inline void send_cmd(id, scpi, val)INSTid;
  char *scpi;
  float val;
  {
    assert(id);
    if (val ≡ FLT_MIN) iprintf(id, scpi);
    else iprintf(id, scpi, val);
  }
```

8. ⟨ Generic communication subroutines 6 ⟩ +≡

```
  inline void DEV_reset(id)INSTid;
  {
    assert(id);
    iprintf(id, "RST\n");
  }
```

9. Open session.

Send a request to open a session communication to a bus address. *addr* is the bus address to send the request. Return The device (instrument) identifier.

Open the session with the specified device id. param id is the device identifier returned by *DEV_open_session*.

```
⟨ Generic communication subroutines 6 ⟩ +≡
  inline INST DEV_open_session(addr)
      char *addr;      /* address */
  {
    INST id;
    ionerror(I_ERROR_EXIT);
    id = iopen(addr);
    itimeout(id, DEV_SESSION_TIMEOUT);
    return id;
  }
```

10. Close session.

```
⟨ Generic communication subroutines 6 ⟩ +≡
  inline void DEV_close_session(id)INST id;
  {
    assert(id);
    iclose(id);
  }
```

11. Read current. Read the current from the specified device identifier. *id* Instrument identifier. Return Value read.

```
⟨ Specialized device communication subroutines 11 ⟩ ≡
  inline float DEV_read_current(id)INST id;
  {
    assert(id);
    return read_dev(id, "MEASure:CURRent?␣(@103)\n");
  }
```

See also sections 12, 15, 16, 17, 18, and 19.

This code is used in section 2.

12. Set current. Set the current of the specified device id to a parameter value val. *id* Instrument identifier. *val* Value to be set.

```
⟨ Specialized device communication subroutines 11 ⟩ +≡
  inline void DEV_set_current(id, val)INST id;
  float val;
  {
    send_cmd(id, "SOURCE:CURRent\n", val);
  }
```

13. Read state.

```
⟨ Generic communication subroutines 6 ⟩ +≡
  inline float DEV_read_state(id)INST id;
  {
    assert(id);
    return read_dev(id, "MEASure:STATe?␣(@103)\n");
  }
```

14. Set state.

```
⟨ Generic communication subroutines 6 ⟩ +≡
  inline void DEV_set_state(id, val)INSTid;
  float val;
  {
    assert(val ≡ ON ∨ val ≡ OFF);
    send_cmd(id, "SOURCE:STATe\n", val);
  }
```

15. Read voltage.

```
⟨ Specialized device communication subroutines 11 ⟩ +≡
  inline float DEV_read_voltage(id)INSTid;
  {
    assert(id);
    return read_dev(id, "MEASure:VOLTage?␣(@103)\n");
  }
```

16. Set voltage.

```
⟨ Specialized device communication subroutines 11 ⟩ +≡
  inline void DEV_set_voltage(id, val)INSTid;
  float val;
  {
    send_cmd(id, "SOURCE:VOLTage\n", val);
  }
```

17. Read temperature. A request of temperature measure is sent to device id and a string with the value is returned. *id* of the device to read the temperature.

```
⟨ Specialized device communication subroutines 11 ⟩ +≡
  inline float DEV_read_temperature(id)INSTid;
  {
    return read_dev(id, "MEASure:TEMPerature?␣(@103)\n");
  }
```

18. Reset resistance voltage. Reset the voltage supplied to the resistance. *id* of the device to reset the voltage.

```
⟨ Specialized device communication subroutines 11 ⟩ +≡
  inline void DEV_reset_resistance_voltage(id)INSTid;
  {
    assert(id > 0);
    DEV_reset(id);
  }
```

19. Data Acquisition. Setup initial commands to device that is responsible to read temperature measures. *id* Device or instrument identifier. FLT_MIN is the minimum float number is used only as a flag to NoValue.

```
⟨ Specialized device communication subroutines 11 ⟩ +≡
  inline void DEV_setup_temperature_device(id)INSTid;
  {
    send_cmd(id, "Uni:Temp␣C,(@103)\n", FLT_MIN);      /* Se t the thermocouple type */
    send_cmd(id, "CONF:TEMP␣TC,J,(@103)\n", FLT_MIN);      /*Reset t he thermocouple type */
    send_cmd(id, "S␣␣␣ENS:TEMP:TRAN:TC:TYPE␣J,(@103)\n", FLT_MIN);
  }
```

addr: 9. *assert*: 7, 8, 10, 11, 13, 14, 15, 18.

⟨ Generic communication subroutines 6, 7, 8, 9, 10, 13, 14 ⟩ Used in section 2.
⟨ Specialized device communication subroutines 11, 12, 15, 16, 17, 18, 19 ⟩ Used in section 2.
⟨ Type definitions 4, 5 ⟩ Used in section 1.
⟨ `dev.h` 1 ⟩

1. Introduction. This is the PID Controller Module, responsible to calculate the control signal needed to act on a plant to reduce the error during the time of system working.

The transfer function of a PID controller is often expressed in the ideal form

$$G_{PID}(s) = \frac{U(s)}{E(s)} = k_P \left(1 + \frac{1}{sT_I} + sT_D\right),$$

where

$U(s)$ – control signal applied on the error signal;
$E(s)$ – error signal;
K_P – proportional gain;
T_I – integral time constant;
T_D – derivative time constant;
s – argument of Laplace transform.

The control signal can also be expressed in three terms as

$$U(s) = k_P E(s) + \frac{k_P}{sT_I} E(s) + sk_P T_D E(s),$$

and time constant can be expressed in terms of gain where $K_I = K_P/T_I$ is the integral gain, and $K_D = K_P T_D$ is the derivative gain.

The feedback represented in the Figure 2; where r is the command signal, y is the output of the plant, $G(s)$ given by Formula 1 is the plant to be controlled, and $C(s)$ is the controller; can be described by

$$U(s) = K_P E(s) + \frac{K_I}{s} E(s) + K_D sE(s)$$

where $\langle K_P, K_I, K_D \rangle$ are the controller parameters to be determined such that the closed-loop system becomes stable.

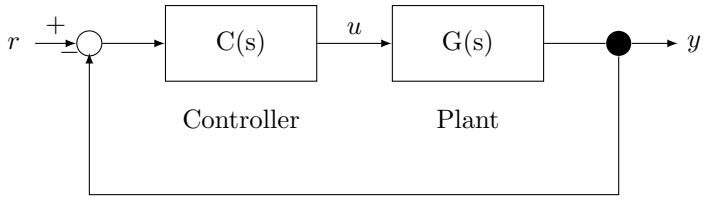

Figure 1. Feedback control system.

The Formula 1 in the time domain is described as

$$U(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{d}{dt} e(t).$$

The algorithm that implements the PID controller is formulated as:

```
previous_error = 0
integral = 0
start:
     error = setpoint - current_value
     integral = integral + error*dt
     derivative = (error - previous_error) / dt
     output = (kp*error) + (ki*integral) + (kd*derivative)
     previous_error = error
     wait(dt)
goto start
```

2. Interface. The PID module has 3 functions to calculate the terms of the PID feedback system, one for each gain, and one function to calculate the PID response in one pass, to avoid the decoupling of terms. The *delay* function is a facility that implements an elapsed time at which the plant is submitted.

```
⟨ pid.h 2 ⟩ ≡
#ifndef PID_INCLUDED
#define PID_INCLUDED
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include "types.h"
  ⟨ Definitions or also called macros 4 ⟩
  ⟨ Type definitions 5 ⟩
  extern inline float calc_P_term( );
  extern inline float calc_I_term( );
  extern inline float calc_D_term( );
  extern inline float PID_output( );      /* one pass response */
  extern int delay( );
#endif      /* PID_INCLUDED */
```

3. Program.

```
  ⟨ Preprocessor definitions ⟩
#include "pid.h"
#include "exp.h"
  ⟨ Global variables 8 ⟩
  ⟨ Functions 12 ⟩
```

4. PID Parameters. The parameters to be set in the experiment are grouped into the **PID_Parameter** structure, that is a type of *Absolute_Parameter* type.

⟨ Definitions or also called macros 4 ⟩ ≡
```
#define PID_PARAMETER_LABEL  "PID␣Parameters"
```
See also section 7.

This code is used in section 2.

5. ⟨ Type definitions 5 ⟩ ≡
```
typedef Absolute_Parameter PID_Parameter;
```
See also sections 6 and 9.

This code is used in section 2.

6. As the sequence of operation in the PID realm is almost always proportional, integral, derivative, an enumeration was declared to index this sequence.

⟨ Type definitions 5 ⟩ +≡
```
enum PID_Params_Enum {
  PROPORTIONAL_IDX,
  INTEGRAL_IDX,
  DERIVATIVE_IDX
};
```

7. PID controller can be initialized using the default parameters, if no initial value is set. To not impact in the experiment values, the initial values are neutral. The values set is a way to eliminate PID controller influence in the experiment.

⟨ Definitions or also called macros 4 ⟩ +≡
```
#define PID_DEFAULT_KP  1.0F
#define PID_DEFAULT_KI  1.0F
#define PID_DEFAULT_KD  1.0F
#define PID_PARAMETERS_N  3
```

8. ⟨ Global variables 8 ⟩ ≡
```
float kp_default = PID_DEFAULT_KP;
float ki_default = PID_DEFAULT_KI;
float kd_default = PID_DEFAULT_KD;
PID_Parameter PID_params[PID_PARAMETERS_N] = {
  [PROPORTIONAL_IDX] = { . label = "Proportional" , . value = &kp_default } ,
  [INTEGRAL_IDX] = { . label = "Integral" , . value = &ki_default } ,
  [DERIVATIVE_IDX] = { . label = "Derivative" , . value = &kd_default }
} ;
```
See also sections 10 and 11.

This code is used in section 3.

9. PID State. The PID integral part is the cumulative error during the experiment, $\int edt$, and the error indicates how far from the setpoint is the feedback of the system. The *integral* variable in the *PID_State* is the error accumulator.

The derivative part is the difference between actual and old value of error, de/dt. The *prev_err* store the value of old value of error.

```
⟨ Type definitions 5 ⟩ +≡
  typedef struct PID_State {
    float integral;
    float prev_err;
  } PID_State;
```

10. The PID state may change at s specified delta time to accommodate the changes in the plant. The macro `PID_DEFAULT_DELTA_TIME` has the value used when no change in delta time is made. There is only one option to change *dt* value:

W hen the the function *delay* is used, inside it *dt* is set to the value to elapse in time. Before any delay is applied the value of *dt* is set to `PID_DEFAULT_DELTA_TIME` with a value that does not take into account the time influence.

```
#define PID_DEFAULT_DELTA_TIME 1
⟨ Global variables 8 ⟩ +≡
  static float dt = PID_DEFAULT_DELTA_TIME;
```

11. Declaration and initialization of the **PID_State** instance.

```
⟨ Global variables 8 ⟩ +≡
  PID_State PID_state = { . integral = 0.0F , . prev_err = 0.0F , } ;
```

12. Proportional. Calculate the proportional term, *err* is the difference of measure read from set point. Proportional term is returned.

```
⟨ Functions 12 ⟩ ≡
  inline float calc_P_term(err)
       float err;
  {
    return err * (*((float *) PID_params[PROPORTIONAL_IDX].value));
  }
```

See also sections 13, 14, 15, and 16.

This code is used in section 3.

13. Integral. Calculate the integral term, where *err* Difference from setpoint. Integral term is returned.

```
⟨ Functions 12 ⟩ +≡
  inline float calc_I_term(err)
       float err;
  {     /* integral of e(t) */
    PID_state.integral += err * dt;
    return PID_state.integral * (*(float *) PID_params[INTEGRAL_IDX].value);
  }
```

14. Derivative. Calculate the derivative term, where *err* Difference from set point. Derivative term is returned.

```
⟨ Functions 12 ⟩ +≡
  inline float calc_D_term(err)
       float err;
  {
    float derr = (err − PID_state.prev_err)/dt;
    PID_state.prev_err = err;       /* update the past */
    return derr;
  }
```

15. PID output. The PID adjust can be calculated at one pass if the current value of the plant were passed as argument.

```
⟨ Functions 12 ⟩ +≡
  float PID_output(err)
        float err;
  {
    return
    calc_P_term(err) + calc_I_term(err) + calc_D_term(err);
  }
```

16. Facilities. Some set of commands are enclosed in the facilities module to not change the context at which we are thinking, mainly to solve PID control problems. *delay* receives time to wait before continue the execution of the resting commands.

```
⟨ Functions 12 ⟩ +≡
  int delay(time_to_wait)
      float time_to_wait;
  {
    time_t start;
    time_t current;
    if (time_to_wait ≠ PID_DEFAULT_DELTA_TIME) dt = time_to_wait;
    time(&start);
    fprintf(stdin, "delay␣for␣%2.2f␣seconds.\n", dt);
    do {
      time(&current);
    } while (difftime(current, start) < dt);
    return EXIT_SUCCESS;
  }
```

17. Index.

⟨ Definitions or also called macros 4, 7 ⟩ Used in section 2.
⟨ Functions 12, 13, 14, 15, 16 ⟩ Used in section 3.
⟨ Global variables 8, 10, 11 ⟩ Used in section 3.
⟨ Type definitions 5, 6, 9 ⟩ Used in section 2.
⟨ `pid.h` 2 ⟩

PID Controller Module

1. Program.

```
#include <windows.h>
#include <string.h>
#include <math.h>
#include <stdbool.h>
#include "types.h"
#include "exp.h"
#define BUFFER_LEN 256
```

⟨ Global declarations 2 ⟩

⟨ Internal functions 30 ⟩

⟨ Callback 4 ⟩

⟨ Main window 3 ⟩

2. Main window. The widgets position are tracked in the variables *x_pos* and *y_pos*. Default values are adopted to window width and height based on the number of components to be showed.

```
#define WIN32UI_WIDGET_HEIGHT 20
#define WIN32UI_PADDING 15
#define WIN32UI_LABEL_WIDTH 135
#define WIN32UI_EDIT_WIDTH 60 /* widget to entry text */
#define WIN32UI_TOTAL_WIDTH (2 * (WIN32UI_LABEL_WIDTH + WIN32UI_EDIT_WIDTH))
#define WIN32UI_TOTAL_HEIGHT
    ((EXP_PARAMETERS_N+PID_PARAMETERS_N+1)*(WIN32UI_WIDGET_HEIGHT+2*WIN32UI_PADDING))
```

⟨ Global declarations 2 ⟩ ≡

```
const char g_szClassName[] = "Conductivity␣Experiment␣GUI";
HINSTANCE g_hinst;
int x_pos, y_pos, y_origin;
char str_buf[BUFFER_LEN];
float cur_val; /* global buffer for float values */
```

See also sections 7 and 13.

This code is used in section 1.

3. ⟨ Main window 3 ⟩ ≡

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
  HWND hwnd;
  MSG msg;
  WNDCLASS wc = {0};
  wc.lpszClassName = TEXT("Application");
  wc.hInstance = hInstance;
  wc.hbrBackground = GetSysColorBrush(COLOR_3DFACE);
  wc.lpfnWndProc = WndProc;
  wc.hCursor = LoadCursor(0, IDC_ARROW);
  g_hinst = hInstance;
  ⟨ Initialization of global variables 8 ⟩
  RegisterClass(&wc);
  hwnd = CreateWindow(wc.lpszClassName, TEXT(EXP_MODULE_NAME),
      WS_OVERLAPPEDWINDOW | WS_VISIBLE, 100, 100,      /* width */
  WIN32UI_TOTAL_WIDTH,      /* height */
  WIN32UI_TOTAL_HEIGHT,
  0, 0, hInstance, 0);
  while (GetMessage(&msg, Λ, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return (int) msg.wParam;
}
```

This code is used in section 1.

4. Callback.

```
⟨ Callback 4 ⟩ ≡
  LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
  WPARAM wParam, LPARAM lParam)
  {
    ⟨ Local callback variables 9 ⟩
    switch (msg) {
    case WM_CREATE: ⟨ Experiment parameters group 12 ⟩
      ⟨ PID parameters group 16 ⟩
      ⟨ Buttons 18 ⟩
      ⟨ Menu creation 27 ⟩
      break;
    case WM_COMMAND:
      switch (LOWORD(wParam)) {
      case ID_RUN_BUTTON: ⟨ Run the experiment button pressed 20 ⟩
        break;
      case ID_ABORT_BUTTON: ⟨ Abort the experiment button pressed 25 ⟩
        break;
      case ID_LIMITS: printf("limits");
        ⟨ Show limits in a message box 29 ⟩
        break;
      }
      break;
    case WM_CLOSE: DestroyWindow(hwnd);
      break;
    case WM_DESTROY: PostQuitMessage(0);
      break;
    }
    return DefWindowProc(hwnd, msg, wParam, lParam);
  }
```

This code is used in section 1.

5. Edit controls. Widgets are composed by the needs of the experiment and PID parameters elements. A common widget height `WIN32UI_WIDGET_HEIGHT` is declared to give an uniform appearance. The total height is accumulated in the `WIN32UI_TOTAL_HEIGHT` macro as widgets that increase the interface height are added. A common padding `WIN32UI_PADDING` is also set. Also, in the benefit of common view the label width is suggested by `WIN32UI_LABEL_HEIGHT`.

⟨ Edit control styles 5 ⟩ ≡

```
WS_CHILD | WS_VISIBLE | WS_BORDER | ES_AUTOHSCROLL | ES_NUMBER ,
```

This code is used in sections 12 and 16.

6. The length text to be entered is set by `EDIT_LIMIT_TEXT_LENGTH`. All buffers that are used to store text from the form use this macro to control the number of characters allowed.

```
#define EDIT_LIMIT_TEXT_LENGTH 8
```

7. Experiment parameters.

```
#define ID_EXP ((pow('E',0)+pow('X',1)+pow('P',2)))
```

⟨ Global declarations 2 ⟩ +≡

```
extern Parameter EXP_params[];
unsigned long ID_EXP_EDIT[EXP_PARAMETERS_N];
```

8. IDs are used to select the edit controls appropriated for the context. Here a room to improve the index mechanism.

⟨ Initialization of global variables 8 ⟩ ≡

```
ID_EXP_EDIT[MAX_TEMPERATURE] = ID_EXP + pow('M',3) + pow('A',3) + pow('X',3);
ID_EXP_EDIT[MIN_TEMPERATURE] = ID_EXP + pow('M',3) + pow('I',3) + pow('N',3);
ID_EXP_EDIT[ENV_TEMPERATURE] = ID_EXP + pow('E',3) + pow('N',3) + pow('V',3);
ID_EXP_EDIT[DELTA_TIME] = ID_EXP + pow('D',3) + pow('T',3);
ID_EXP_EDIT[SET_POINT] = ID_EXP + pow('S',3) + pow('T',3);
```

See also section 14.

This code is used in section 3.

9. ⟨ Local callback variables 9 ⟩ ≡

```
int i;
char val_str[EXP_PARAMETERS_N][BUFFER_SIZE];
static HWND exp_params_hwnd_edit[EXP_PARAMETERS_N];
```

See also sections 15, 17, 19, 23, 26, and 28.

This code is used in section 4.

10. `X_ORIGIN` and `Y_ORIGIN` indicate the initial position in the frame where the widgets are started to group. *y_pos* are initialized and updated to the value of widget height plus a padding set in `WIN32UI_PADDING`. *x_pos* is initialized summing this padding and the origin in x axis set in `X_ORIGIN`.

```
#define X_ORIGIN 10
#define Y_ORIGIN 10
```

⟨ Update y position 10 ⟩ ≡

```
y_pos += WIN32UI_WIDGET_HEIGHT + WIN32UI_PADDING;
```

This code is used in sections 12 and 16.

11. ⟨ Update y origin 11 ⟩ ≡

```
y_origin = y_pos + WIN32UI_PADDING;
```

This code is used in sections 12 and 16.

12. ⟨ Experiment parameters group 12 ⟩ ≡

```
y_origin = Y_ORIGIN;
x_pos = WIN32UI_PADDING + X_ORIGIN;
⟨ Update y position 10 ⟩
for (i = 0; i < EXP_PARAMETERS_N; i++) {
  CreateWindow(TEXT("STATIC"), TEXT(EXP_params[i].lbvalue→label),
      WS_CHILD | WS_VISIBLE | SS_LEFT, x_pos,      /* x */      /* y */
  y_pos,      /* width, height */
  WIN32UI_LABEL_WIDTH, WIN32UI_WIDGET_HEIGHT,
  hwnd, (HMENU)1, Λ, Λ);      /* Parameters edit */
  float2str((*(float *) EXP_params[i].lbvalue→value), &val_str[i]);
  exp_params_hwnd_edit[i] = CreateWindow(TEXT("Edit"), TEXT(val_str[i]), ⟨ Edit control styles 5 ⟩
    /* x */
  x_pos + WIN32UI_LABEL_WIDTH +
  WIN32UI_PADDING/2,      /* y */
  y_pos,      /* width */
  WIN32UI_EDIT_WIDTH,      /* width */
  20, hwnd,
  (HMENU)ID_EXP_EDIT[i], Λ, Λ);      /* Unit labels */
  CreateWindow(TEXT("STATIC"), TEXT(EXP_params[i].unit), WS_CHILD | WS_VISIBLE | SS_LEFT,
    /* x */
  x_pos + WIN32UI_LABEL_WIDTH +
  WIN32UI_PADDING +
  WIN32UI_EDIT_WIDTH,      /* y */
  y_pos,      /* width */
  WIN32UI_LABEL_WIDTH,      /* height */
  WIN32UI_WIDGET_HEIGHT,
  hwnd, (HMENU)1, Λ, Λ);
  ⟨ Update y position 10 ⟩
}      /* end for int i */      /* Group box for Experiment Parameters */
CreateWindow(TEXT("button"), TEXT(EXP_PARAMETER_LABEL), WS_CHILD | WS_VISIBLE | BS_GROUPBOX,
  /* x*/
X_ORIGIN,      /* y*/
Y_ORIGIN,      /* width */
2 * WIN32UI_LABEL_WIDTH +
WIN32UI_EDIT_WIDTH,      /* height */
y_pos,      /* space in the origin and end*/
hwnd, (HMENU)0, g_hinst, Λ);
⟨ Update y origin 11 ⟩
```

This code is used in section 4.

13. PID Parameters.

```
#define ID_PID ((pow('P',0)+pow('I',1)+pow('D',2)))
⟨ Global declarations 2 ⟩ +≡
  extern PID_Parameter PID_params[PID_PARAMETERS_N];
  unsigned long ID_PID_EDIT[PID_PARAMETERS_N];
```

14. ⟨ Initialization of global variables 8 ⟩ +≡

```
ID_PID_EDIT[PROPORTIONAL_IDX] = ID_PID + pow('P',3);
ID_PID_EDIT[INTEGRAL_IDX] = ID_PID + pow('I',3);
ID_PID_EDIT[DERIVATIVE_IDX] = ID_PID + pow('D',3);
```

15. ⟨ Local callback variables 9 ⟩ +≡

```
char pid_val_str[PID_PARAMETERS_N][BUFFER_SIZE];
static HWND pid_params_hwnd_edit[PID_PARAMETERS_N];
```

16. ⟨ PID parameters group 16 ⟩ ≡

```
⟨ Update y position 10 ⟩ /* advance y position */
for (i = 0; i < PID_PARAMETERS_N; i++) { /* PID parameter labels */
  CreateWindow(TEXT("STATIC"), TEXT(PID_params[i].label), WS_CHILD | WS_VISIBLE | SS_LEFT,
    /* x */
  x_pos, /* y */
  y_pos, /* width */
  WIN32UI_LABEL_WIDTH, /* height */
  WIN32UI_WIDGET_HEIGHT,
  hwnd, (HMENU)1, Λ, Λ); /* PID parameter edit controls */
  float2str((*(float *) PID_params[i].value), &pid_val_str[i]);
  pid_params_hwnd_edit[i] = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", TEXT(pid_val_str[i]), ⟨ Edit
      control styles 5 ⟩ /* x */
  x_pos +
  WIN32UI_LABEL_WIDTH, /* y */
  y_pos, /* width */
  WIN32UI_EDIT_WIDTH, /* width */
  20, hwnd,
  (HMENU)ID_PID_EDIT[i], GetModuleHandle(Λ), Λ);
  ⟨ Update y position 10 ⟩
} /* end of for */ /* Group box for PID Parameters */
CreateWindow(TEXT("button"), TEXT(PID_PARAMETER_LABEL), WS_CHILD | WS_VISIBLE | BS_GROUPBOX,
  /* x*/
X_ORIGIN, /* y*/
y_origin, /* width */
2 * WIN32UI_LABEL_WIDTH +
WIN32UI_EDIT_WIDTH, /* height */
(WIN32UI_PADDING + WIN32UI_WIDGET_HEIGHT) * (PID_PARAMETERS_N + 1), hwnd, (HMENU)0, g_hinst, Λ);
⟨ Update y origin 11 ⟩
```

This code is used in section 4.

17. Running.

```
#define ID_RUN ((unsigned int)(pow('R',0)+pow('U',1)))
#define WIN32UI_BUTTON_HEIGHT 1.5*WIN32UI_WIDGET_HEIGHT
#define WIN32UI_BUTTON_WIDTH 100
#define ID_RUN_BUTTON ID_RUN
```

⟨ Local callback variables 9 ⟩ +≡

```
static HWND run_hwnd_button;
```

18. ⟨ Buttons 18 ⟩ ≡

```
run_hwnd_button = CreateWindow(TEXT("button"),
TEXT("Run"),
WS_CHILD | WS_VISIBLE,      /* x */
WIN32UI_TOTAL_WIDTH/2 -
WIN32UI_BUTTON_WIDTH -
WIN32UI_PADDING,      /* y */
y_pos + 2*WIN32UI_PADDING,      /* width */
WIN32UI_BUTTON_WIDTH,      /* height */
WIN32UI_BUTTON_HEIGHT,
hwnd,(HMENU)ID_RUN_BUTTON,g_hinst,Λ);
```

See also section 24.

This code is used in section 4.

19. ⟨ Local callback variables 9 ⟩ +≡

```
static int len;
static TCHAR text[EDIT_LIMIT_TEXT_LENGTH + 1];
static HWND hwnd_edit;      /* local handle used only for edit */
```

20. The running button is disabled untill some tests can be done. The restrictions are verified in the following order:
V erify the emptiness of edit controllers;
V erify if the value of each parameter is positive;
V erify if the parameter values is less than the limit, or greater than in the specific case of `MIN_TEMPERATURE`.

```
#define DISABLE_RUN
```

```
⟨ Run the experiment button pressed 20 ⟩ ≡
  for (i = 0; i < EXP_PARAMETERS_N; i++) {
    hwnd_edit = exp_params_hwnd_edit[i];
    len = GetWindowTextLength(hwnd_edit) + 1;
    GetWindowText(hwnd_edit, text, len);
    ⟨ Is the edit control empty 21 ⟩
    assert(atof(text) > 0.0F);
    if (¬EXP_is_limit_constraint_satisfied(i, atof(text))) ⟨ Limit constraint was not satisfied 22 ⟩
    EXP_set_parameter_value(i, atof(text));
  }
  for (i = 0; i < PID_PARAMETERS_N; i++) {
    hwnd_edit = pid_params_hwnd_edit[i];
    len = GetWindowTextLength(hwnd_edit) + 1;
    GetWindowText(hwnd_edit, text, len);
    ⟨ Is the edit control empty 21 ⟩
  }
#ifdef DISABLE_RUN
  MessageBox(hwnd_edit, "Run␣button␣is␣disabled␣by␣default!",
  "Run␣button␣is␣disbled!",
  MB_OK | MB_ICONEXCLAMATION);
#else
  EXP_run_experiment( );
#endif
```

This code is used in section 4.

21. Is the edit control empty?

```
⟨ Is the edit control empty 21 ⟩ ≡
  if (is_edit_empty(hwnd_edit)) {
    SetFocus(hwnd_edit);
    return false;
  }
```

This code is used in section 20.

22. Limit constraint was not satisfied, warn the user about the problem and the limit to be respected.

```
⟨ Limit constraint was not satisfied 22 ⟩ ≡
  {
    sprintf(str_buf, "%s%s%s", "Limit␣value␣was␣not␣satisfied!\n",
    "Please␣adjust␣the␣value.\n",
    "See␣Help--Limits␣in␣the␣Menu␣for␣aditional␣information.");
    show_message_box(hwnd_edit, str_buf);
    SetFocus(hwnd_edit);
    return false;
  }
```

This code is used in section 20.

23. Aborting.

```
#define ID_ABORT ((unsigned int)(pow('A',0) + pow('B',1)))
#define ID_ABORT_BUTTON ID_ABORT
```

⟨ Local callback variables 9 ⟩ +≡

```
static HWND abort_hwnd_button;
```

24. ⟨ Buttons 18 ⟩ +≡

```
abort_hwnd_button = CreateWindow(TEXT("button"), TEXT("ABORT"),
WS_CHILD | WS_VISIBLE,      /* x */
WIN32UI_TOTAL_WIDTH/2 +
WIN32UI_PADDING,      /* y */
y_pos + 2 * WIN32UI_PADDING,      /* width */
WIN32UI_BUTTON_WIDTH,      /* height */
WIN32UI_BUTTON_HEIGHT,
hwnd, (HMENU)ID_ABORT_BUTTON, g_hinst, Λ);
```

25. ⟨ Abort the experiment button pressed 25 ⟩ ≡

```
#ifdef DISABLE_RUN
  MessageBox(hwnd_edit, "Abort␣button␣is␣disabled␣by␣default!",
  "Abort␣button␣is␣disbled!",
  MB_OK | MB_ICONEXCLAMATION);
#else
  EXP_halt( );
#endif
```

This code is used in section 4.

26. Menu.

```
#define ID_LIMITS #1000 /* IMPROVE THE ID INDEXATION */
⟨ Local callback variables 9 ⟩ +≡
  HMENU hMenu, hSubMenu;
```

27. ⟨ Menu creation 27 ⟩ ≡

```
hMenu = CreateMenu( );
SetMenu(hwnd, hMenu);
hSubMenu = CreatePopupMenu( );
AppendMenu(hSubMenu, MF_STRING, ID_LIMITS, "&Limits...");
InsertMenu(hMenu, 0, MF_POPUP | MF_BYPOSITION, (UINT_PTR)hSubMenu, "Help");
DestroyMenu(hSubMenu);
DestroyMenu(hMenu);
DrawMenuBar(hwnd);
```

This code is used in section 4.

28. Limits.

```
⟨ Local callback variables 9 ⟩ +≡
  char *limits_str;
  limits_str = (char *) malloc(128);
```

29. ⟨ Show limits in a message box 29 ⟩ ≡

```
sprintf(str_buf, "%s", "Parameter␣limits:\n");
strcpy(limits_str, str_buf);
for (i = 0; i < EXP_PARAMETERS_N; i++) {
  cur_val = *(float *)(EXP_params[i].limit);
  sprintf(str_buf, "%d.␣%s␣=>␣value=%6.2f,␣unit=%s\n", i + 1, EXP_params[i].lbvalue→label, cur_val,
      EXP_params[i].unit);
  strcat(limits_str, str_buf);
}
MessageBox(hwnd, limits_str,
"Limits␣information.",
MB_OK | MB_ICONEXCLAMATION);
free(limits_str);
```

This code is used in section 4.

30. Facilities.

```
#define BUFFER_SIZE #100
```

⟨ Internal functions 30 ⟩ ≡

```
void float2str(val, str)
    float val;
    char str[BUFFER_SIZE];
{
  int result = 0;
  result = snprintf(str, BUFFER_SIZE, "%5.2f", val);
  if (result > BUFFER_SIZE)
    fprintf(stderr, "[%s:%d] The string has been truncated!", __FILE__, __LINE__);
}
```

See also sections 31 and 32.

This code is used in section 1.

31. Message box.

⟨ Internal functions 30 ⟩ +≡

```
int show_message_box(hwnd, msg)HWND hwnd;
char *msg;
{
  MessageBox(hwnd, msg,
  "Warning!",
  MB_OK | MB_ICONEXCLAMATION);
  SetFocus(hwnd);
  return 0;
}
```

32. Constraints. edit not empty constraint.

```
#define NOT_EMPTY_CONSTRAINT_LABEL "Not␣empty␣edit␣constraint."
#define NOT_EMPTY_WARNING "The␣entry␣could␣be␣empty.␣Please,␣fill␣it!"
```

⟨ Internal functions 30 ⟩ +≡

```
static bool is_edit_empty(hwnd_edit)HWND hwnd_edit;
{
  int len;
  TCHAR text[EDIT_LIMIT_TEXT_LENGTH + 1];
  len = GetWindowTextLength(hwnd_edit) + 1;
  GetWindowText(hwnd_edit, text, len);
  if (len ≤ 1) {
    MessageBox(hwnd_edit, NOT_EMPTY_WARNING,
    NOT_EMPTY_CONSTRAINT_LABEL,
    MB_OK | MB_ICONEXCLAMATION);
    return true;
  }
  return false;
}
```

Graphical User Interface Module

4 Todo

Improve IO system. The IO output was implemented but it was not taken into account the user directory path that can be extracted from system profile. This was made to make the system the most simple as possible when debugging. Another improvement in the IO system could be store the value of temperature and time to allow PID debug.

Improve index of UI components. The better index values to components seems to be the hexadecimal, so a change of the user interface components **ID** is reasonable.

Graph system. The suggested way to know the trend of data behavior is to use LiveGraph software, but this program was not tested yet, open a room for a home-made real-time graph system.

References

[1] Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Professional, Massachusetts, 1st Edition, 2002.

[2] Donald E. Knuth, *Literate Programming*, Center for the Study of Language and Inf., 1st Edition, 1992.