600085 Embedded Systems Development

Green House Control System

Individual Report

Word count: 1954

Zak Catherall
December 2019

Contents

1 Introduction

This document details my personal contributions to the development of the Green House Control System, which was implemented on a PIC 16F877A microcontroller. The project focused on the creation of drivers to interface with various components found on a QL-200 development board, and then implementing them to create high-level features.

2 Artefacts Produced

This section outlines all contributions made regarding the module; this includes design, code and documentation. With respect to *code*, methods that are marked as 'private' are excluded from the header file for that driver, enabling some level of code encapsulation (which is not explicitly provided in C). If not specified, methods are 'public', which is achieved by including the definition in the header file.

2.1 Driver Code

This section details the interfaces with hardware components that I produced.

2.1.1 LCD (lcd_driver)

The LCD driver is responsible for managing system output, allowing the device to interface with the user via text printed to the screen.

The design of the driver aims to offer fundamental operations that can be used to create appropriately complex user-interfaces.

Specification

- Pins used:
  - A5 for command/data
  - A4 for read/write
  - A3 for enable/reset
  - A2 for parallel/serial
  - All PORTD for transmitting data.
- Allows output to 4 individual lines,16 characters long.
- Allows output to any cell (cursor position) on the screen.
- Operates in parallel mode to read data from PORTD and avoid LCD flicker.

API

- lcd_Init
  Initialises ports, pins and default values.
- lcd_WriteCmd *(private)*
  Writes a command byte to the LCD component to affect its internal configuration (hence it is privatised to limit control).
- lcd_PrintChar
  Outputs a single character to the LCD at the current cursor position.
- lcd_PrintString
  Outputs a string of characters to the LCD at the defined cursor position.
- lcd_SetCursorPos
  Uses lcd_WriteCmd() to set the on-screen cursor position, where any outputs will begin from.
- lcd_Clear
  Clears the LCD of all outputs.

Testing

The following tests were conducted manually to verify the driver. Some methods do not produce an observable output by themselves and cannot be tested without integration (e.g. Init).

Test	Steps	Expected	Result (PASS/FAIL)
lcd_Init	-	*lcd_PrintChar() passes.*	PASS
lcd_WriteCmd	-	*lcd_PrintChar() passes.*	PASS
lcd_PrintChar	1. Call lcd_Init() 2. Call lcd_PrintChar('a')	'a' appeared in the top left corner of the LCD.	PASS
lcd_PrintString	1. Call lcd_Init() 2. Call lcd_PrintString('test', 0, 0)	'test' appeared in the top left corner of the LCD.	PASS
lcd_SetCursorPos	1. Call lcd_Init() 2. Call lcd_SetCursorPos (3, 3) 3. Call lcd_PrintChar('1')	'1' appeared on the 4th line of the LCD, in the 4th character cell (cells are 2 characters wide).	PASS
lcd_Clear	1. Call lcd_Init() 2. Call lcd_PrintString('test', 0, 0) 3. Call lcd_Clear()	There is no output on the LCD.	PASS

Table 1: LCD test report.

2.1.2 Thermometer (themometer_driver)

The thermometer driver is responsible for allowing the developer to interface with the thermistor component, to read the temperature of the environment in degrees Celsius.

The driver is designed for visual systems, so a single public method handles the internal process of reading and converting the temperature to a string.

Specification

- Pins used:
  - E0 for read/write
- Operates via 1-pin mode, where the component receives power during the read/write cycles instead of continually drawing power ("parasite power").
- Allows the temperature to be obtained to 1 decimal place, with units, as a string.

API

- therm_WriteByte *(private)*
  Writes a command byte to the component.
- therm_ReadByte *(private)*
  Reads a part of the temperature from the component, dependent on the timing (e.g. decimal portion).
- therm_ReadTemp *(private)*
  Performs the sequence required to read the temperature in 1-pin mode and returns it as an int.
- therm_Reset *(private)*
  Performs the reset part of the read sequence required in 1-pin mode to determine if data has been sent.

- therm_GetTemp
  Utilises the private methods to read the temperature and format it as a string (to one decimal place and with units) and returns it.

Testing

The following tests were conducted manually to verify the driver. Some methods do not produce an observable output by themselves and cannot be tested without integration (e.g. WriteByte).

Test	Steps	Expected	Result (PASS/FAIL)
therm_WriteByte	-	*therm_GetTemp() passes.*	PASS
therm_ReadByte	-	*therm_GetTemp() passes.*	PASS
therm_ReadTemp	-	*therm_GetTemp() passes.*	PASS
therm_Reset	-	*therm_GetTemp() passes.*	PASS
therm_GetTemp	1. Call lcd_Init() 2. Call lcd_PrintString(therm_GetTemp(), 0, 0)	The correct temperature is printed in the top left corner of the LCD.	PASS

Table 2: Thermometer test report.

2.1.3 RTC (rtc_driver)

The RTC (real-time clock) driver interfaces with the clock component to allow the system datetime to be written and read.

The driver is designed for visual systems, so it contains a public method handles the internal process of reading and converting the temperature to a string. The datetime (string) format is based on ISO 8601, where:

- *Date* = YY-MM-DD (year, month, date)
- *Time* = hh:mm:ss (hour, minutes, seconds in 24hr format)

Contrary to this format, the weekday is appended to the end of the date as a 3-letter abbreviation (e.g. 'Fri').

This driver depends on the utilities library, which provides conversions from binary coded decimal (BCD) - the default format for these clock values.

Specification

- Pins used:
  - B0 for read/write
  - B4 for clock sync signal
  - B5 for enable/reset
- Operates in 24hr mode.
- Uses, "burst mode", to read all time registers sequentially (into a table).
- Allows the time or date to be retrieved as a formatted string.
- Calculates the day of the week automatically, based on the system time.

API

- rtc_Init
  Initialises ports, pins and default values.
- rtc_WriteByte *(private)*
  Writes a command byte to the component.

- rtc_ReadByte *(private)*
  Reads a component of time from the clock, depending on the given address (or iteration, in burst mode).
- rtc_SetDay *(private)*
  Calculates the day of the week as a number from 0-6 and sets the weekday register in the clock.
- rtc_Update
  Uses, "burst mode", to read the clock registers and update the values stored in a table/array.
- rtc_SetTimeComponent
  Sets a specific register/component of time (e.g. day).
- rtc_GetString
  Reads the date or time indexes in the values table and returns a string in the format described.

Testing

The following tests were conducted manually to verify the driver. Some methods do not produce an observable output by themselves and cannot be tested without integration (e.g. Init).

Test	**Steps**	**Expected**	**Result (PASS/FAIL)**
rtc_Init	-	*rtc_SetDay passes.*	PASS
rtc_WriteByte	-	*rtc_SetDay passes.*	PASS
rtc_ReadByte	-	*rtc_SetDay passes.*	PASS
rtc_SetDay	1. Call rtc_Init() 2. Call lcd_Init() 3. Call lcd_PrintString(therm_GetTemp(), 0, 0)	The weekday displayed is correct for the system date.	PASS
rtc_Update	1. Call rtc_Init() 2. Call lcd_Init() In a loop: 3. Call rtc_Update() 4. Call lcd_PrintString(rtc_GetString(0)), 0, 0)	The time is displayed in the top left corner of the LCD and the seconds update.	PASS
rtc_SetTimeComponent	1. Call rtc_Init() 1. Call lcd_Init() 2. Call rtc_Update() 3. Call rtc_SetTimeComponent(0x8C, 0x06) 4. Call lcd_PrintString(rtc_GetString(1), 0, 0)	The date is printed to the top left of the LCD and the year is 06.	PASS

rtc_GetString	1. Call rtc_Init() 2. Call lcd_Init() 3. Call rtc_Update() 4. Call lcd_PrintString(rtc_GetString(1)), 0, 0) 5. Call lcd_PrintString(rtc_GetString(0)), 1, 0)	The date is printed in the format "YY-MM-DD Day" on the top line of the LCD, and the time is printed on the second line in the format, "hh:mm:ss"	PASS

Table 3: RTC test report.

2.1.4 Buzzer (buzzer_driver)

The buzzer driver interfaces with the buzzer component to allow the developer to produce a variety of patterns and tones.

Specification

- Pins used:
  - E0 for on/off

API

- buzzer_Init
  Initialises ports, pins and default values.
- buzzer_Sound
  Produces a pattern of buzzes and pauses, *n* number of times. The durations and *n* are defined by the developer as method arguments.

Testing

The following tests were conducted manually to verify the driver. Some methods do not produce an observable output by themselves and cannot be tested without integration (i.e. Init).

Test	**Steps**	**Expected**	**Result (PASS/FAIL)**
buzzer_Init	-	*All other methods work.*	PASS
buzzer_Sound	1. Call buzzer_Init() 2. Call buzzer_Sound(15000, 15000, 3)	The buzzer produces 3 steady beeps with pauses between them.	PASS

Table 4: Buzzer test report.

2.3 Application Code

This section describes the parts of the system that I was responsible for that were not drivers.

2.3.1 Utilities (utils)

The concept of a utilities library was not my own, but I did implement much of its final functionality, which mostly handles BCD conversions for the *rtc_driver* and some quality of life variables (e.g. true and false) – as seen below.

```
// DEFINES.
#define uch unsigned char // Shorthand for unsigned char.
#define true 0x01 // Makeshift boolean.
#define false 0x00 // Makeshift boolean.
#define toInt -0x30 // Add (+) to chars to convert to ints.
#define eol '\0' // String terminate char.

// PUBLIC VARS.
//- Precise timings for DelayT.
uch t503us[0x02] = {0x02, 0x46};
uch t430us[0x02] = {0x02, 0x3C};
uch t70us[0x02] = {0x02, 0x08};
uch t63us[0x02] = {0x02, 0x07};

// PUBLIC METHODS.
void Delay(int n); // Delay n cycles.
void DelayT(uch t[]); // Delay a specific time given by a pre-calcuated, char array.
uch StrLen(char*); // Get length of a string.
char* BcdToStr(char); // Convert BCD (binary coded decimal) to string.
char StrToBcd(char str[]); // Convert string to BCD (binary coded decimal).
uch IsLeapYear(int yr); // Return whether the year is a leap year or not.
float StrToFloat(char str[]); // Convert a string with format ##.# to float representation.
uch BcdToDec(char bcd); // Convert a BCD (binary coded decimal) to decimal.
```

Figure 1: Screen capture of utils.h.

Testing

Testing of utilities was performed slightly differently than for the drivers. Since they didn't interface with hardware, an online C compiler was used to debug the code via console prints. The tests performed are included in the table below.

Test	Steps	Expected	Result (PASS/FAIL)
StrLen	StrLen("123")	Returns 3.	PASS
BcdToStr	BcdToStr(0x16)	Returns "16"	PASS
StrToBcd	StrToBcd("10")	Returns 0x10	PASS
IsLeapYear	1. IsLeapYear(19) 2. IsLeapYear(20) 3. IsLeapYear(21) 4. IsLeapYear(22) 5. IsLeapYear(23) 6. IsLeapYear(24)	1. Returns false 2. Returns true 3. Returns false 4. Returns false 5. Returns false 6. Returns true	PASS
StrToFloat	StrToFloat("12.3")	Returns 12.3	PASS

BcdToDec	BcdToDec(0x10)	Returns 10	PASS

Table 5: Utilities test report.

2.3.2 User Interface (ui)

I designed and implemented the user interface including the various screens, button scheme, navigation and input validation. These are discussed in greater depth in the group report, especially in the user manual, which is included in the appendix of this document.

2.3.3 Main Program (main)

I *designed* the primary system loop and we implemented it collaboratively as our individual features were completed.

2.4 Other Contributions

In addition to the code already described, I helped to refactor the implementations of the *matrix_driver* and the *io_driver*, which helped me to develop my understanding of these components. For the *matrix_driver*, I completed the button mappings. For the *io_driver*, I generalised it to manage a collection of pins, rather than hard-coded "heating" and "cooling" pins.

Additionally, I trawled the codebase to amend inconsistencies in naming conventions and to change *int* datatypes to *char* where possible (as a space saving exercise). Whilst doing this I added additional comments to unexplained code.

Finally, I contributed a significant amount to the group report, producing the:

- Introduction
- Requirements
- Logic-flow diagram
- Driver/API listings
- Integration testing (and test report)
- User manual
- Evaluation.

4 Evaluation

By completing the Green House Control System, I have gained some understanding of simple electronics by reading oscillation diagrams, block diagrams, and datasheets in general. With a pure software background, this was a daunting but necessary challenge, which I am proud to have overcome.

Implementing the system under the constraints imposed by embedded systems has made me much more conscious and appreciative of datatypes and program structure, with some of the key techniques learnt (to reduce program size) being:

- Convert integer literals under 255 to unsigned chars.
- Limit function returns in favour of using value arrays (‘tables’) .
- Use bitwise operations such as bit-shifting, bitwise-and, and bitwise-or – to simplify expressions.
- Avoid using floats and float math as much as possible.

A significant handicap in developing the system was the absence of debugging tools. After some time, I realised logic could often be abstracted and implemented as a function in an ordinary C program. This was leveraged by essentially unit testing methods in online C compilers, where the values of variables and return values could be printed.

Two of the biggest take-aways from the project are the importance of referencing the datasheet and proper Git etiquette. Both habits took some getting used to, and some of the biggest problems in the group were caused by our differences in Git fluency.

Overall, the I deem the project a success both as a deliverable and as a learning experience. If I were to produce the same work again, I am confident I could do so to a higher quality and in less time.

5 Appendices

5.1 Appendix 1: User Manual

Contents

1 System Overview

The greenhouse control system monitors time and temperature to manage temperature deviations, by providing power to external cooling and heating systems and alerting the user of failure of these systems.

2 First time use

When starting the device for the first-time, the user is presented with the settings screen where they must input values for daytime and thresholds before normal operation resumes. This ensures that the system is configured before use.

3 User Interface

3.1 Inputs (Button Mappings)

The following is a diagram of the button mappings in the 4x4 matrix.

7 *Input the number 7*	**8** *Input the number 8*	**9** *Input the number 9*	**x** *Cancel/back*
4 *Input the number 4*	**5** *Input the number 5*	**6** *Input the number 6*	**<** *Up/previous*
1 *Input the number 1 /select menu item 1*	**2** *Input the number 2 /select menu item 2*	**3** *Input the number 3 /select menu item 3*	**>** *Down/next*
0 *Input the number 0*	**.** *Input a decimal point*	**b** *Backspace*	**s** *Enter/select*

Figure 2: Device inputs and button map.

3.2 Screens

The following are representations of the different screens displayed by the device, with explanations for screen items and diagrams of available inputs (highlighted in green).

Name	Screen	Description	Inputs
Standby	Da: YY-MM-DD Day Ti: HH:MM:SS Te: ##.#C St: <status text>	Date (year: month: day) Time (hours: minutes: seconds) Temperature (degrees Celsius) Status (OK/HEATING/COOLING)	7 8 9 x 4 5 6 < 1 2 3 **>** 0 . b s

Settings (page 1)	Settings 1. Date 2. Time …	Screen Title Access the *Set Date* screen. Access the *Set Time* screen. Next-page indicator	7 8 9 x 4 5 6 < 1 2 3 > 0 . b s
Settings (page 2)	… 1. Daytime 2. Threshold (d) 3. Threshold (n)	Previous-page indicator Access the *Set Daytime* screen. Access the *Set Thresholds (daytime)* screen. Access the *Set Thresholds (night)* screen.	7 8 9 x 4 5 6 < 1 2 3 > 0 . b s
Set Date	Date Cur: YY-MM-DD New: …	Screen Title Current value Input area Confirm-input indicator	**7 8 9 x** **4 5 6** < **1 2 3** > **0** . **b s**
Set Time	Time Cur: HH:MM:SS New: …	Screen Title Current value Input area Confirm-input indicator	**7 8 9 x** **4 5 6** < **1 2 3** > **0** . **b s**
Set Daytime	Daytime Cur: HH:MM HH:MM New: …	Screen Title Current values (start & end) Input are Confirm-input indicator	**7 8 9 x** **4 5 6** < **1 2 3** > **0** . **b s**
Set Thresholds (daytime)	Threshold (d) Cur: ##.#C ##.#C New: …	Screen Title Current values (lower & upper) Input area Confirm-input indicator	**7 8 9 x** **4 5 6** < **1 2 3** > **0** . **b s**
Set Thresholds (night time)	Threshold (n) Cur: ##.#C ##.#C New: …	Screen Title Current values (lower & upper) Input area Confirm-input indicator	**7 8 9 x** **4 5 6** < **1 2 3** > **0** . **b s**
Alarm	Hold to disarm!	Message	**7 8 9 x** **4 5 6 <** **1 2 3 >** **0 . b s**
Valid Input	Success!	Message	None

Invalid Input	Invalid!	Message	None

Figure 2: The different views of the system (UI).

4 Settings

For all settings there is an individual input screen which accepts a limited number of numeric inputs. All input screens exhibit the following behaviours:

- The expected number of inputs must be provided i.e. a single digit value of '1' must be provided as '01', if the expected number inputs is 2.
- Previous inputs can be deleted by pressing the backspace button.
- The user can press the cancel button to return to the settings screen without saving changes.
- When the expected number of inputs have been provided, "…", appears at the bottom of the screen to indicate that the user can press the select button to submit.
- If the inputs are **invalid**, the 'Invalid Input' screen will display for a short time and the user will be returned to the input screen.
- If the inputs are **valid**, the 'Valid Input' screen will display for a short time, the time will be set, and the user will be returned to the settings screen.

4.1 Setting the Date

NB: *the weekday is automatically calculated.*

Navigation:

Standby → Settings (page 1) → Set Date

Purpose:

The system date is used for displaying the current date on the standby screen.

Input:

- The screen accepts 6 numeric input characters in the format: YYMMDD (year 00-99, month 01-12, date 01-31).
- A separation character ('-') is inserted automatically after the YY and MM inputs for readability.
- Dates must be valid.

4.2 Setting the Time

NB: *the system clock operates in 24hr mode.*

Navigation:

Standby → Settings (page 1) → Set Time

Purpose:

The system time is used for displaying the current time on the standby screen and deciding which operation mode (daytime or night time) is active.

Input:

- The screen accepts 6 numeric input characters in the format: HHMMSS (hours 00-23, minutes 00-59, seconds 00-59).
- Separation characters (':') are inserted automatically after the HH and MM inputs for readability.

4.3 Setting the Daytime Operating Period

Navigation:

Standby → Settings (page 1) → Settings (page 2) → Set Daytime

Purpose:

Within the bounds of the configured time, the system operates in daytime mode. Outside of those bounds, the system operates in night time mode; each mode has individually configurable temperature thresholds.

Input:

- The screen accepts 6 numeric input characters in the format: HHMHHM (hours 00-23, minutes 0-5), where the first HHM represents the daytime start and the second represents the end (when night time begins).
- Start and end times can only be configured to the nearest 10 minutes and a zero is automatically inserted after a 'M' input.
- Separation characters (':') are inserted automatically after the HH inputs for readability.
- A space is automatically inserted between the start and end times.

4.4 Setting Temperature Thresholds

Navigation:

For daytime operation:

Standby → Settings (page 1) → Settings (page 2) → Set Thresholds (daytime)

For night time operation:

Standby → Settings (page 1) → Settings (page 2) → Set Thresholds (night time)

Purpose:

The temperature thresholds are used by the system to determine if the heating or cooling systems should be active.

Input:

- The screen accepts 6 numeric input characters 0-9, where the first 3 represent the lower threshold (when heating should be activated) and the second 3 represent the upper threshold (when cooling should be activated).
- Values are to 1 decimal place i.e. the thresholds are in the format '00.0'.
- Decimal points are inserted automatically.
- A 'C' is inserted after the every 3rd input to indicate the units of measurement (Celsius).
- A space is automatically inserted between the lower and upper thresholds.
- A lower threshold must be less than its corresponding upper threshold by at least 0.2C.

5 Passive Operations

The following describe the fundamental functions of the Green House Control System.

5.1 Day and Night Modes

The system automatically switches between day and night operation based on the current system time and configured "Daytime" setting.

5.2 Heating and Cooling

Whenever the system temperature moves outside of the user-specified range for the current operating period, the heating or cooling system will be activated accordingly.

When either system is activated, the system emits an indicative beep, and the LCD status on the standby screen changes to, “COOLING”, or, “HEATING”. At normal temperatures this status will display as, “OK”, plus a reference to the current mode (day or night).

5.3 Alarm

NB: The alarm will not sound whilst the user is not on the standby screen, as avoid disrupting user input whilst configuring the system.

The alarm sounding indicates that the heating or cooling systems are not operating sufficiently to correct the system temperature.

The alarm will sound if the temperature falls below the configured lower temperature threshold of the current operating mode (day or night) and continues to fall after the heating system has been activated. Likewise, the alarm will also sound the temperature continues to rise after the cooling system is enabled.

Whilst the alarm is sounding, the screen will display, “Hold to disarm!”, indicating that the user can hold any button to deactivate the alarm.

The alarm will remain disarmed for approximately 5 seconds before sounding again (if the temperature is not improving).