ESS: Exercise Set 3

Software Design

Question 1:

Design an embedded system to control a hot-water tank/boiler. Start from a set of requirements and follow through the Vee diagram, looking at architecture, block-diagram, modules, modelling, V&V and deployment. How would your system be different for a domestic boiler, as compared with an industrial plant?

Solution:

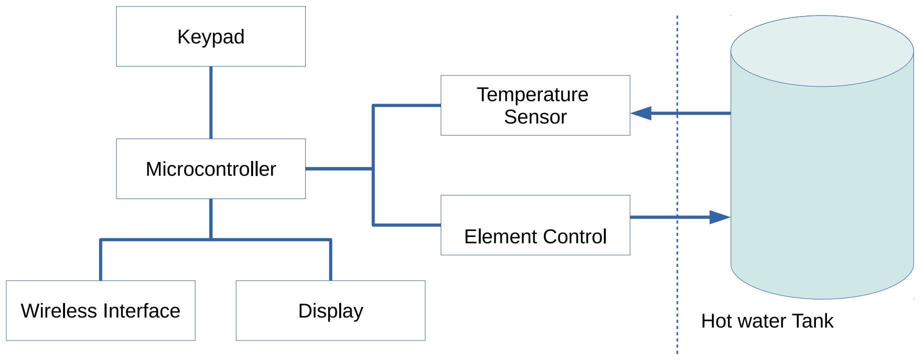
This is a high-level overview of some dimensions that you may consider - there are many different requirements.

Requirements

- R1: Water should be heated to within 3°C of a set-point.
- R2: The system should indicate when the water is at desired temperature.
- R3: If there is no water in the tank, the element should be protected from burnout.
- R4: If the tank is over-pressure, an alarm should sound and heating cut-out [maybe only for the industrial case?].
- R5: The control system should be linked to the internet [maybe only for the domestic case?]
- R6: The setpoint should be adjustable [remotely?]
- R7: The control system should exploit periods of cheaper energy [maybe only for the domestic case?]
- R8: The controller should cost under £100.

Architecture

Block diagram:



Other factors to consider:

- What about a power supply?
- What about an audible alarm?
- What about a water level sensor?
- What about a real-time-clock?

At the next level of the design, consider how the individual software components and modules could interact with each other. Also consider more precisely which family of microntroller is needed, and what kind of specifications it needs (e.g. maybe a 32 bit microntroller, but it would not need a large RAM, perhaps 64 kByte would be sufficient).

For the wireless interface, should it be Bluetooth, WiFi, or perhaps long range narrow band e.g. LORA or SigFox. What are the costs (power, complexity, dollars) of each of these modules?

For the temperature sensor, what to use? Maybe a digital temperature sensor. Or perhaps this is not physically robust enough, and you need to use a thermocouple. But if you use a thermocouple, then you need a microcontroller with an on-board ADC to convert the readings. What resolution of ADC would you need? 8 bit/10 bit/16 bit?

What about the display? Should it be a simple LCD, or do you want to display colours/graphics (unlikely for a unit locked in a cupboard). What is the relative cost/complexity of each variant? An LCD is simple and inexpensive, but then the microntroller needs to support LCD driving (alternating voltage). If you use a TFT display, then the microntroller most likely needs more FLASH space (and RAM) to store the graphics.

Do you need any logging capability? Where/how are you going to store logging data? What temporal resolution do you need? What history do you need (hours/months/years?).

Question 2:

Show how to wrap the led driver from Lab 1 in a harness to do unit testing. The full API is as follows:

```c
// initialize led
void led_init(LED_t * led, volatile uint32_t * port, uint32_t pin);
// turn led on
void led_on(LED_t * led);
// turn led off
void led_off(LED_t * led);
// toggle led from off to on and vice-versa
void led_toggle(LED_t * led);
// returns whether led is set (1) or clear (0)
uint8_t led_read(LED_t * led);
```

The harness should not depend on the hardware itself - how could we decouple the logic from the actual hardware (e.g. PORTD?).

Solution:

The solution here is to use a technique called dependency injection. The LED driver is dependent on the port register, which is very hardware/platform specific. However, a port is just a register (memory location). We could use any memory location that we can pass in as a pointer, and then we can spy on the register to check that it is being set/cleared properly.

```c
#include <assert.h>
// Create a register that we will spy on
uint32_t fake_led_port;
// Turn it into a pointer
volatile uint32_t * port_ptr;
// Set the pointer
port_ptr = &fake_led_port;
// Create an led type
LED_t my_led;
// Initialize the port
led_init(&my_led,port_ptr,0);
// do a test
assert(fake_led_port == 0x00);
// Set the pin
led_set(&my_led);
// do a test
assert(fake_led_port == 0x01);
```

(a) Write some tests[1] e.g. `led_init()` should set the bit for the LED to off, but leave the rest of the port register unaltered. Write some sample assertions that you could use to check these.

[1]See `http://throwtheswitch.org/` for a site dedicated to TDD on embedded platforms

Solution:

A non-exhaustive list of tests are:

Test_init_clear: After initialization, the bit for the LED should be clear. The `fake_led_port` register will be seeded with an arbitrary bit pattern e.g. `0x31F1`.

Test_led_set: If `led_set()` is called, the bit for the LED should be set. The `fake_led_port` register will be seeded with an arbitrary bit pattern e.g. `0x31F0` and the bit in question will be set.

Test_led_toggle: If `led_toggle()` is called, the bit for the LED should be set. The `fake_led_port` register will be seeded with an arbitrary bit pattern e.g. `0x31F0` and the bit in question will be toggled. After each instruction, the register shall be checked to see if it is correct.

Test_multi_led: To check that multiple leds do not affect each other, two led ADTs will be created on different bit positions. A number of set/ clear/ toggle operations will be run. After each operation, the register will be checked.

Test_bounds: If an led pin greater than 15 is given, the routine should handle this safely[2].

Test_has_been_initialized: This is a challenging one - if `led_set()` or similar is called before `led_init()`, the port that the LED type points to could be anything (but most likely to be a null (`0x0000`) pointer). We could end up writing to a completely arbitrary location in memory with potentially catastrophic consequences. Can you think of some solutions to this problem - perhaps adding another variable to the LED struct that is set to a magic sequence once initialized? Or a checksum based on the pin and port? Or, with knowledge of the hardware, a sensible range of pointer values?

(b) To do the testing, we can use `assert()` statements, or if we have support for `printf`)), a very minimal test suite (minunit[3])is defined by the following macro:

```
/* file: minunit.h */
#define mu_assert(message, test) do { if (!(test)) return message; } while (0)
#define mu_run_test(test) do { char *message = test(); tests_run++; \
                               if (message) return message; } while (0)
extern int tests_run;
```

This shows how to run some tests:

```
/* file test_suites.c */

 #include <stdio.h>
 #include "minunit.h"

 int tests_run = 0;

 int foo = 7;
 int bar = 4;
```

[3]http://www.jera.com/techinfo/jtns/jtn002.html

```c
static char * test_foo() {
    mu_assert("error, foo != 7", foo == 7);
    return 0;
}

static char * test_bar() {
    mu_assert("error, bar != 5", bar == 5);
    return 0;
}

static char * all_tests() {
    mu_run_test(test_foo);
    mu_run_test(test_bar);
    return 0;
}

int main(int argc, char **argv) {
    char *result = all_tests();
    if (result != 0) {
        printf("%s\n", result);
    }
    else {
        printf("ALL TESTS PASSED\n");
    }
    printf("Tests run: %d\n", tests_run);

    return result != 0;
}
```

Using the minunit framework, show how to write a few of the tests you specified above.

Question 3:

Write a test strategy for an alarm clock module which triggers an alarm when the time exceeds the preset alarm time. The API for the alarm clock is as follows:

```c
#include "real_time.h"
// initialize alarm module - initially no alarm is set
void alarm_init(void);
// set the alarm
void set_alarm(clock_time_t alarm_time);
// clear the alarm.
void clear_alarm(void);
```

The relevant API for the dependency file real_time.h is as follows:

```c
struct clock_time
{
  uint8_t year;
  uint8_t month;
  uint8_t day;
  uint8_t hour;
  uint8_t minute;
  uint8_t second;
};
```

```c
typedef struct clock_time clock_time_t;

// provide a pointer to a clock_time_t struct, this function will update it with the current tim
void get_current_time(clock_time_t * my_time);
```

Solution:

We have a dependency chain problem, in that the alarm clock module depends on real_time.h. This could be very platform dependent. In addition, we need to set the time to various boundary points such as 23:59:59 to check whether the alarm will trigger properly. The solution is to mock out real_time.h with our own implementation e.g. our own real_time.c file. This can be done in the project or by using makefiles. By writing our own implementation, we can inject arbitrary times into the alarm clock implementation.