GR18 Project Report

Elias Vahlberg and Isak Ringdahl

January 2021

Abstract

This report details all parts of the project, each section describes **our approach** to tackle a certain requirement. **The methodology** we used and the **connection to the task** requirement.

Requirement 1

The system should have a calendar that allows showing date and time. The user should be able to configure it whenever it is needed. The date should be represented as DD/MM/YYYY, while the clock as hh:mm:ss. (Use the 24-hour clock system). The system can use SysTick to measure time.

Purpose: For knowing date and time and timestamp the recorded data.

Located in:

DateTime.c

Functions:

```
init_date_time();
display_date_time();
set_timedate();
config_time();
config_date();
toggle_fastmode();
```

Our approach, use microseconds incremented in the systick interrupt handler to calculate the hours, minutes and seconds.

These values(h,min,sec) are saved in the global variables current_time and current_time_hm. They are updated when the *set_timedate()* is called.

The date is in contrast to the time incremented rather than set every time, this is also done in set_timedate().

The time/date are configured via config_time() and config_date()

First take in the time, each digit at a time (and update display).

Then take in the date each digit at a time (and update display).

Then:

Time saved in current_time (char[3]).

Date saved in current_date (char[3]).

Methodology we use one central global variable for the time (microseconds) to reduce the operations in the systick interrupt handler (less computation). The data type used for all date and time values is unsigned char. This aims to use the least amount of memory (mostly to save space of timestamps) and fits all the values except the year. The solution to this was to add an offset that is applied when displayed but understood under normal operations (offset = 2000, maximum year value = 2255).

Requirement 2

SUPERSEDED, by REQ 9

Requirement 3

Presentation of recorded data on the LCD by text. Each day is presented by minimum, average, maximum and variance values for temperature. Maximum and minimum values should be presented along with their timestamp.

Purpose: Presentation of logged data.

Located in:

TempStatistics.c

Functions:

```
update_statistics()
calc_statistics()
print_day_data()
```

Our approach,

Firstly we had to find an efficient way to calculate the statistics that should be presented to the user.

Secondly, how should this data be stored.

Lastly how should the data be converted and displayed.

Methodology

The method used to calculate these statistics is to initialize the values and update these values when the list containing the temperature values is full or if it has overwritten all previously calculated values(the data list is circularly converted). Since these updates happen multiple times for each day (due to the small heap memory) we simply sum the temp values and the square of the temp values to be used at the end of the day to calculate the average and variance.The function that does this is *update_statistics.* When the clock records a new day the flag *new_day_flag* is set and the calc_statistics is called and finalizes the values by first calling *update_statistics()* for the last remaining values and after that the average and variance from the two sums: var_sum1=SUM(x), var_sum2=SUM(xˆ2).

The statistics data is initialized at startup and is gradually filled up day by day (this not to cause memory issues that occur after multiple days). Each day is stored in a struct named *day_temp_data*

Containing the following values:

date day - The date of the data represented

double min - Minimum temperature during the day

time_hm tmin - Timestamp where the minimum value was recorded

double avg - Average temperature during the day

double max - Maximum temperature during the day

time_hm tmax - Timestamp where the maximum value was recorded

double vari - Temperature variance during the day

Since we do not store more than seven of these structs we could afford to save the temp values as doubles, mainly benefits average and variance.

This achieves the requirement "Each day is presented by minimum, average, maximum and variance values for temperature."

To display these values the function *print_day_data()* is used this function then calls the *day_temp_data_to_string()* given a specific day data and a destination char array. Using some other string conversion functions located in CommonFunctions.c the day data is converted to a string that is returned in the char array provided. After this the string is printed and separated to multiple lines on the screen.

The function *print_day_data()* is called from the main loop with a day corresponding to a keypress (1-7).

This achieves the requirement "Presentation of recorded data on the LCD by text".

Requirement 4

SUPERSEDED, by REQ 10

Requirement 5

The home temperature should be maintained between a defined upper and lower limits. An alarm should be raised in case these limits were crossed.

Purpose: Alarm at under or over temperature.

Located in:

TemperatureSensor.c

Functions:

```
temperature_alarm();
temperature_display_alarm();
```

Our approach,

For this requirement only two pieces had to work, detect an abnormal temperature and prompt the user.

Methodology

The function *temperature_alarm* is called from the main loop when it is displayed in the sidebar, in this function it checks if there already is an alarm and if this is no longer the case the temp_alarm_flag is cleared. If there is no current alarm the function checks if the temperature is outside the upper and lower bounds (defined in *TEMP_ALARM_UPPER_VALUE* , *TEMP_ALARM_LOWER_VALUE*).

If the temperature is outside these bounds a temporary flag is set to *1*: "temp too low" or *2*: "temp too high", to make sure this value is accurate the temperature is checked several more times (number of times saved in *TEMP_ALARM_RECHECK*). If the temporary gives the same result for all the measurements the function *temperature_display_alarm()* is called.

In the function *temperature_display_alarm()* the display is cleared and an appropriate message is displayed to the user(saved as : *TEMP_ALARM_DISPLAY_MES_LOW or TEMP_ALARM_DISPLAY_MES_HIGH*). After this the write access to the screen is revoked from all functions until the alarm has been cleared.

This achieves the requirement "An alarm should be raised in case these limits were crossed".

Requirement 6

Create a fast mode that does exactly the same as Req. 2 where each minute is simulated by a second.

Purpose: To simplify testing.

Located in: DateTime.c

Functions:

```
toggle_fastmode()
```

Our approach,

The smallest part of this requirement was to implement a toggle that the user could press and "make time flow faster".

The majority of the work surrounding this task was optimizing other parts of the system that ran too slow to be able to handle temperature recordings 10 times every seconds (maximum number of measurements per min in Req 9 was 10).

Methodology

In order to implement the fast mode we simply changed the conversion from microseconds to seconds depending on *fast_mode_flag* (in *set_timedate()*). To toggle the fast mode we added a case to the main menu, pressing *"*"* toggles the fast mode . When this key is pressed from the main menu the function *toggle_fastmode* is called. In this function it simply inverts the flag and divides/multiples the *microseconds* variable. When this flag is active some parts of the system are no longer accessible by the user since fast mode requires the system to operate much faster so that the main loop takes no longer than 70 milliseconds (70 since it records every 100 milliseconds and the recording can sometimes take a bit longer due to faulty values). The functions no longer accessible during fast mode are : Graph mode, Test mode, logout and the servo. But since the fast mode is mainly used to test the temp recordings these tradeoffs are in our opinion acceptable.

This achieves the requirement "Create a fast mode that does exactly the same as Req. 2 where each minute is simulated by a second. ".

Requirement 9

Periodic temperature recording every minute. Each minute, temperature is recorded N times and averaged to give one number; data is stored for as long as there is memory available. When memory is full an indication is made in the user interface and the oldest values are overwritten, in a circular fashion. N is selectable, range 1-10, in a settings screen by the keypad. The N temperature values, as well as the average value of them, should be stored with their timestamp. (This requirement supersedes Req. 2)

Purpose: Recording of temperature.

Located in:

SingleLinkedList.c, TemperatureSensor.c

Functions:

```
append_to_list();
create_node_with_data();
convert_to_circular_list();
circular_overwrite();
delete_first_node();
get_num_mes_per_min();
```

Our approach,

Datatypes and conversions.

Our first objective was to determine how this data should be stored (what kind of data type is used).

The second objective was to determine what kind of datastructure should be used to house this data and how we could easily interact with it.

The last objective was to configure this datastructure to work efficiently on the memory and cause the least amount of errors (hardfault:IMPRECISERR: memory/ buss faults).

Methodology

The datatype we used to store the measurements, timestamp and the average was a custom struct called linked_node which was a pointer to a container struct storing:

Next node - pointer

Hour - unsigned char

Minute - unsigned char

Temp - unsigned char (the first temperature recording, but used as a starting point for the rest of the N selectable temperature values along with the average at the end).

Since the number of measurements per min is not fixed it would not be feasible to store them in an array within the same allocated memory space as container struct(*Linked_Element*). Furthermore the allocation of this data separately would require more memory since now both the container struct and this array would be allocated, and by virtue of how the memory allocation functions this would both fragment the memory and take more space (observed during tests).

The solution we used to solve this was to allocate enough memory so that the struct along with the extra N values (N-1 but + 1 for average) would fit. This means when a node is allocated the allocated memory size is: sizeof(struct Linked_element) + N, (N since each temp recording takes 1 byte).

When accessing this memory the address of the variable *temp* in the container acts as the starting point of an "dynamic array" within the struct.

The data structure type we used was a single linked list so that memory could easily be freed if it was necessary. This also allowed us to configure so that it worked circularly after enough data was stored in it.

When we need to store a new node, the function append_to_list() is called provided the head, tail and data. For the first two nodes the head and tail is filled with data. After this the data is appended to the tail and then the tail is moved. To create these new nodes the function *create_node_with_data()* is called which allocates an appropriate amount of memory. If this allocation is not successful it is concluded that the memory is full.

When this happens the list is reduced in size by 10% (so that other parts of the system can use some heap as well). When this is done it is converted to a circular list, meaning that the next value of the current tail is connected to the head and this is prompted on the display. To make sure that the system does not overwrite data that has not yet been gathered for statistics a pointer named circular_start is saved at the current tail.

When adding new data after the list has been converted, the function *circular_overwrite()* is called instead of *create_node_with_data().* As the name suggests it overwrites the saved data in a circular fashion.

This achieves the requirement "When memory is full an indication is made in the user interface and the oldest values are overwritten, in a circular fashion.".

To make sure that the temperature is recorded periodically a flag named *measure_temp_flag* is updated in DateTime.c :set_timedate() to the current second of the current minute. This flag is compared in the main loop to the flag *last_temp_measure.* If these two are not the same it means that it should perform a temperature recording. After the temperature recording is done the *last_temp_measure* flag is set equal to *measure_temp_flag* . This is simply

to prevent the temp sensor from recording multiple times for the same second. When the flags are not the same the function *add_temp_recording()* is called and the current temperature reading is added to a temporary array until the *new_minute_flag* is set then this array is provided to the *append_to_list()* function.

This achieves the requirement "N is selectable, range 1-10, in a settings screen by the keypad. The N temperature values, as well as the average value of them, should be stored with their timestamp. " .

Requirement 10

Use two photosensors to achieve the function of Req. 4. (This requirement supersedes Req. 4)

Purpose: Tracking of sun position and controlling window shades.

Located in:

Servo.c

Functions:

```
set_to_max_light();
get_light_rotation_data();
servo_set_position();
```

Our approach To track the light, we use our two photosensors to help us detect the light. We use adc converters and the resistive properties of the photosensors to produce easier discrete numbers.

Photosensor: We use terms together with interrupt handler to measure relevant voltage. We mask bits 0 to 11 in from address *AT91C_ADCC_CDR1* and *AT91C_ADCC_CDR2* to get the correct data for the two light sensors.

Then we divide by our "*light_to_volt_coefficient*" which describes our "analog to digital conversion" calculation.

Servo: Pulse width modulation is used to control the servo and we set both the time period (channel period) and the time for the duty cycle through *AT91C_PWMC_CH1_CPRDR* & *AT91C_PWMC_CH1_CDTYR*.

We set our servo to a rotation of 10 degrees in our function *servo_set_position (int x)* with the condition that the previous set parameter is not the same as the current one.

We used soldering tools to assemble the photosensors with the servo and from there measure the light. An array with the size 36 is created where we loop and save each rotation and light measurement for each index (i) with the servo and

photosensors. The first half of the array saves data from CH1 and the second half CH2.

Finally we use our function *set_to_max_light()* to compare the array-values and rotate the servo to where the max light was measured.

This achieves the requirement "Use two photosensors to achieve the function of Req. 4 " .

Requirement 11

Presentation of recorded data on the LCD display using graphs. Each day (specify the date) is presented by minimum, average, maximum and variance values for temperature.

Purpose: Presentation of logged data

Located in: *DisplayGraphics.c*

Functions:

```
All contained in the file DisplayGraphics.c
```

Our approach

To help us achieve this task we implemented a simple library with the following functionalities :

Toggle the graphics mode on and off

Set the cursor to a give x,y pixel block

Clear the whole graphics area

Clear parts of the graphics area

Draw horizontal/vertical lines with a given start position, length and width

Convert a temperature value in to a vertical line with the height determined by the value of the double

These functionalities combined allows the system to take a set of double values (within set limits and provided in a *day_temp_data* struct) and draw a histogram with a given value type (i.e. average, min..).

Methodology

The process of writing in graphics mode was somewhat different than text mode not only because of the cursor offset that has to be added. But also since the display addresses are mapped to blocks of 1x8 pixels rather than 8x8 blocks. So the sheer amount of writing that is done in graphics mode quickly becomes a source of complications. The approach we used for the text mode also became inefficient.

To remedy this the operations done in graphics mode are more manual and with less fine grain control than the text mode. But since the requirement is possible to complete using only horizontal and vertical lines it is manageable.

From the main menu if the key "0" is pressed the *toggle_graphics_mode()* function is called.

The variable *menu_type* is toggled between -1:graphics mode, 0: main menu and the displaymode is set to either "graphics and text mode" or back to "text mode". After this the pointer to the array containing all the temperature statistics saved for the last 7 days are provided to the function *display_staple_plot* along with the value type that should be displayed (1:Average, 2: Minimum, 3: Maximum, 4: Variance). This function combines all the rest of the functions in the *DisplayGraphics.c* file to display a simple histogram with this value type.

This achieves the requirement "Presentation of recorded data on the LCD display using graphs.Each day (specify the date) is presented by minimum, average, maximum and variance values for temperature. " .

Requirement 12

Create a test mode for Req. 11 using pre-recorded data to test the graph mode

Located in: TestMode.c

Functions:

```
graph_data_test()
```

Methodology

The function graph_data_test() is called from the test menu accessed by pressing "9" and then "1".

This function creates seven unique *day_temp_data* nodes temporarily that are then provided to the *display_staple_plot* and utilizes the same functionality as in the main menu (pressing keys 1-4 changes the value type). If the key "#" is pressed the function returns to the main menu.

This achieves the requirement "Create a test mode for Req. 11 using pre-recorded data to test the graph mode

" .

Connection to the task

Requirement 13

The smart home system should include a Username/Password feature to allow users to access its features. It is not secure to store the password as plaintext. A general approach is to store a hash of the password which is the output of a secure one way function. When the user inputs the password, the hash of the password is computed and compared to the stored hash.Implement MD5 hashing in order to hash the password.

Located in: *Loginsystem.c, MD5.c*

Functions:

```
All functions in Loginsystem.c, MD5.c
```

Our approach

To create a functioning login system we required the following functionalities:

Halt normal operations before login.

Take in a username and password from the user via the keypad.

Hash a password.

Validate that a username and password correspond to a saved username.

Allow edits to the user that is currently logged in.

Methodology

Since this requirement regards data security and the handling of sensitive information (user passwords) some of the functions appear to execute redundant code such as excessive deleting of local data and convoluted nested if statements. In the file MD5.c there is also heavy usage of macros containing values that could be computed. The reason for the redundant/convoluted is simply to minimize the risk of leaving information that could be used to compromise the system. As for the heavy usage of macros is simply to save on computation time and the risk that hashing system could access data that it is not meant to (which could be one point of attack by simply injecting data to the memory via some bug, not likely but still).

The login system starts by putting the Program Counter in a "sandbox" i.e. the *login()* function which keeps it there until a valid login has been entered. This is practically locking the user out of the system unless they have a valid login.

The login screen is displayed to the user and allows input (0-9, which is displayed as it is written) followed by a "#" to indicate that the input is done if there was a typo in the input the user can press "*" to go back to the previous character. The maximum length of the username and password is 56 (due to constraints in the hashing set to reduce heavy memory usage when hashing). This is all

done from the *login_get_user_input()* function that is called two times for the username then the password .

After this the password is hashed with the username as the salt and done for 1000 iterations(from the function *hashed_password() and md5_definitive_hash(),* this solves the problems mentioned in the requirement description). Then the plain text password is deleted along with its original length and both the username and password is compared with all the saved users and if there is a perfect match the *current_user* variable is set to the user which matched the login information.

Since the MD5 hashing implementation would take more than this entire report to detail we chose to not detail it here but will do so during the presentation and if requested we can send an appendix detailing this part of the project.

This achieves all of the parts of requirement 13 .

Requirement 14

The system should have one administrator with username admin. You should initially have a default password for which you store the hash. However, the user should be able to change this password. To change the password, the user should first login, then select a new password. Only the hash of the new password should be stored.

Located in: LoginSystem.c

Functions:

```
admin_edit_user()
edit_user()
```

Methodology

To indicate that a user is an admin there is one value in each stored user named *acess_level* that is either 0:normal or 1:admin, if this tag is set to 1 then the user is allowed to change the username and password of other users by calling *admin_edit_user()* . When first starting the system the user which is stored as default has this tag set to 1 (and the username: “1234” , password: “1234”). After this a new username and password is set as the admin removing the default username and password.

This achieves all of the parts of requirement 14.

Requirement 15

Extended test mode. Create a component testing mode so that the operation of each component can be validated (unit test).

Purpose: To simplify testing.

Located in: TestMode.c

Functions:

```
All functions in TestMode.c
```

Methodology

The tests that we settled on creating where for the components that caused the most issues while creating this project and where feasible to test in this manner:

Display - *graph_data_test()*

Detailed in **req 12**

Temperature sensor - *temperature_sensor_test()*

Since the temperature sensor has to operate quickly and accurately but the system has no way of knowing if the recordings are accurate. Therefore the test measures the temperature, displays it along with the time it took to measure this temperature (important for fast mode).

Hashing system - *hash_test()*

This test uses a more ordinary type of unit test where the user is prompted to first input a username and password that is set as the reference. After this the user is again prompted to enter a username and password but this time a different one. Lastly the user should again enter the reference username and password. From this we can check if the hash is unique and if it generates the same hash given the same username and password, the method used for hashing is the same as in the login system.

Keypad - *keypad_test()*

Every key is test in order , when a successful keypress is entered

It displays OK and moves on.

LightSensor - *light_sensor_test()*

The same problem as the temperature sensor arises here where the system itself cannot diagnose since it has no auxiliary way of measuring light. So the light values are displayed on the screen so that the user can easily check.

This achieves the requirement "Extended test mode. Create a component testing mode so that the operation of each component can be validated".