Clémence Vanpouille, Tolga Erol

DT4012 – Computer System Engineering

01/01/2024

Smart Home Project

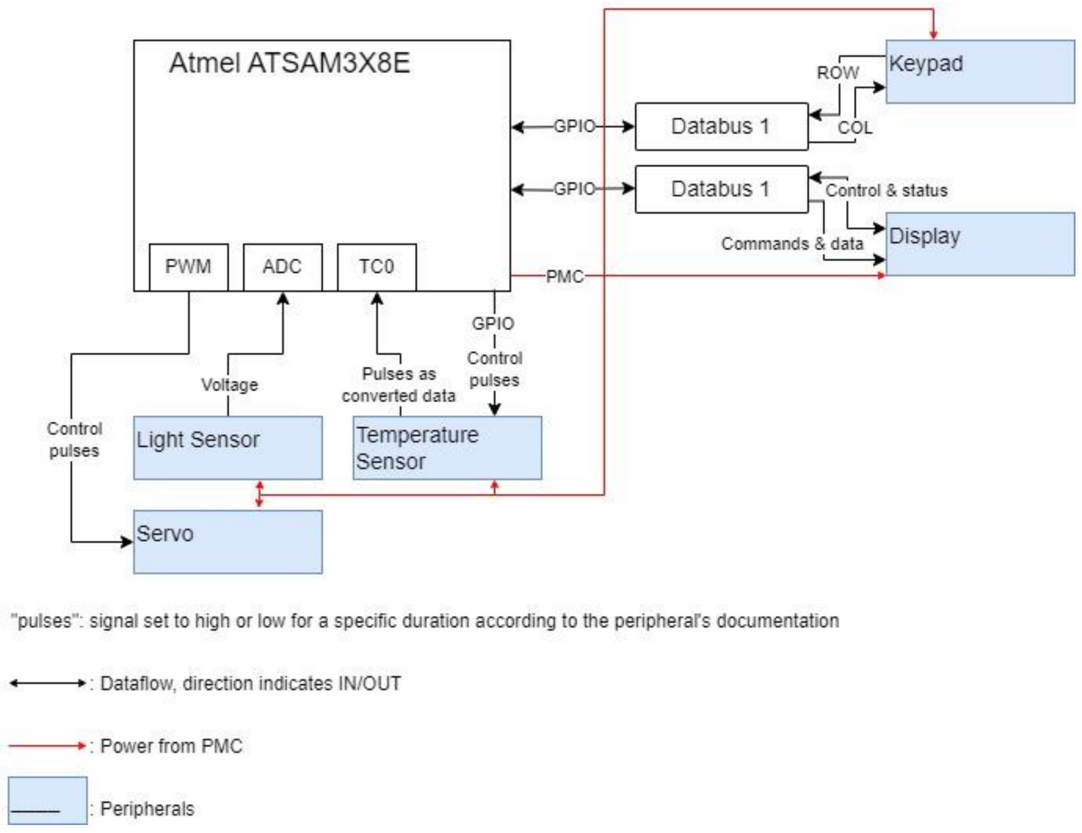
This project aims to develop a system to monitor and control home climate utilizing an Atmel SAM3X8E embedded computer platform (Arduino Due) integrated with various sensors and motors. The system monitors temperature and light intensity, ensuring a temperature range of 20 – 25 degrees Celsius is maintained.
The peripherals include a photosensor, temperature sensor, keypad, RC servo motor, display, LEDs, switch, and buttons, each responding to specific functionalities within the system. The system is designed to meet specific requirements which this report will devote itself to explain and discuss further in the following parts.

Starting with an **introduction** explaining and giving an overview of the system and the moving parts within. **Methods** going into further detail explaining drivers, libraries and data structures used to fulfill the requirements. **Results** presenting the fulfilment of said requirements and a **Discussion** section to discuss design choices and conclude the report.

Index

Introduction

As previously mentioned, the goal of this project is to create an interactive “Home system” that measures temperature and light intensity which can be viewed, controlled and expressed via the Display, Keypad, and Servo. Interactions within the system can be viewed in the functional block diagram below.



Peripheral Components

Temperature Sensor

The temperature sensor, like all other peripherals, is powered by the Pulse Width Modulation (PWM) and takes an external Control pulse through the time-select pins TS1 and TS0 as input which determines the delay between conversions. In the case of the project both TS1 and TS0 was set to low hence having a delay of 5μs/K.
The sensor then outputs the converted data through TC0 to the micro controller (MCU) in the unit of Kelvin (K).

Light Sensor

The light sensor outputs an analog voltage signal in response to light intensity, where increased intensities of light increase the internal resistance and lower the output voltage. To read the analog data from the sensor we use the Analog to Digital Converter (ADC) on the MCU.

Servo

The servo takes an analog input and rotates an angle specified in the waveform. The waveform is generated by the PWM Channel Duty Cycle Register (PWM_CDTYR) and can be configured with the formula $(X \times CPRD)/MCK$. The Master Clock (MCK) being the fundamental clock and X being some prescaler value multiplied by the period CPRD. The servo then rotates some angle determined by the pulse-width within the defined period, granted the angle is less or equal to 180 degrees.

Keypad

The MCU determines keypresses by comparing column values with row values. The rows are set as input to the keypad and columns are outputs from the keypad. By setting all columns high and rows set low by keypresses; the position of the pressed key can be determined by performing an AND operation between row and column. Communication between keypad and MCU is done through 7 GPIO lines connected to a databus. Having information sent in parallel decreasing amount of clock cycles needed to communicate in comparison to having serial. The use of the databuses comes when we use the same GPIO lines for both the keypad and the display, eliminating additional wiring for dataflow to the display aswell.

Display

The display takes data and commands as inputs from the MCU while outputting controls and status of the display. This data flows in conjunction with data between the keypad and MCU necessitating the use of these databuses to prevent interference. The quality of the databuses being able to switch the flow of data; reversing inputs to outputs and vice versa. While data or commands are being sent to the display, any interrupts from the keypad are ignored so as not disturb the workflow for the display. Furthermore, eliminating the need for interrupts for the display altogether.

Project Overview and Accomplishments

1. Requirements implemented

- Req. 1: Calendar System

The system incorporates a calendar based on SysTick. Users can configure the date and time, represented as DD/MM/YYYY and hh:mm:ss in the 24-hour clock system, respectively. The calendar is used to calculate timings within the system when its use is needed for specific peripherals.

- Req. 2: Periodic Temperature Recording

Temperature is recorded every minute for a duration of 7 days. The recorded data, timestamped at the moment of measurement, is stored in a linked list as part of the Calendar. If the memory buffer is full, the system will delete old data until it can store the new one.

- Req. 3: Recordings on the LCD

The system displays on the Display the minimum, average, and maximum values for every day for the past seven days.

- Req. 4: Sun Tracking System

Two light sensors oriented towards a window are used to calculate the position of the sun and the system will use this data to steer the servo to an angle that will face the sun.

- Req. 5: Temperature Alarms

User-configurable upper and lower temperature limits are checked periodically by the system. In case of a limit is met, an alarm is triggered indicated by blinking LED on the board and a blinking alarm message on the LCD. The alarm persists until the user acknowledges and manually resets it.

- Req. 6: Fast Mode

A fast mode is implemented to simulate each 30 minutes by a second. This feature is used for testing the system, especially the temperature recording and records display screen.

1. Technical approach

During the implementation of various project features, pivotal design choices were made to shape our system. This section elucidates and elaborates on these decisions, offering a comprehensive understanding that will be further detailed in subsequent sections.

The primary objective of our project is to interact with various peripherals and manage interrupts from these peripherals. To achieve this, we utilize handler functions provided by the library. These functions, however, do not allow for argument passing, necessitating the extensive use of global variables in the software component of our system.
To manage these global variables effectively, we have grouped them into distinct global structures, each corresponding to a specific peripheral. This organization not only brings order to the globals but also enhances the efficiency of our data structure.
To further optimize our code readability, we have included references to the associated functions of each device within these structures. This approach has resulted in a more organized and efficient codebase.
In practical terms, this translates to a code schema for each peripheral that includes a global structure, “device Device”, which consolidates access to functions (e.g., “Device.init()”)

and data (e.g., “Device.data”).

The structure of each device will be further explained in part 1.

One of the big challenges of C programming is memory management, even more so when working on embedded systems as our available memory is limited. To ensure the stability of the system, we have tried to limit memory usage by using appropriate typing and frequent checks to our recorded data to delete irrelevant older entries. When adding said recordings to the memory, we make sure to delete the oldest data in the case of memory saturation where we would not have the space to save more recent values. Testing has been done by letting the board run for a full night on Fast Mode (cf. Req. 6) and verifying the functionality of the different features afterward.

For embedded systems specifically, the other challenge is code efficiency. Those systems run on limited power, compared to our usual computers. Making our corde more efficient not only ensures prolonged component lifespan but also contributes to the overall efficiency and performance of the embedded system. Memory management, as specified previously, is already playing a significant role in overall efficiency. In addition to these measures, we have made sure to reduce the workload on the different tasks that have to be performed by the system. Some of these optimizations include the use of switch-cases instead of agglomerations of “if” statements and trying to implement algorithms with the least depth possible.

2. Peripheral devices software structure

The global structures for each peripheral are detailed as follows:

- Display
  - init function to configure used pins.
  - enable function for device specific initialization instructions.
  - printfAt function printing a string to the display at the specified position using variable arguments list and taking advantage of sprintf function.
  - clear function to clear the entire display.

- Keypad
  - init function to configure used pins.
  - poll function to fetch current key pressed.
  - key integer to store the last keypad’s key that was pressed.

- Light Sensor “Light”
  - init function to configure used pins.
  - enable function for device specific initialization instructions.
  - get function to get the last converted data.

- Servo
  - init function to configure used pins.
  - setPos function to set a desired angle to which the motor will rotate to.
  - steer function to add a specific angle to the current position and call the setPos function accordingly.

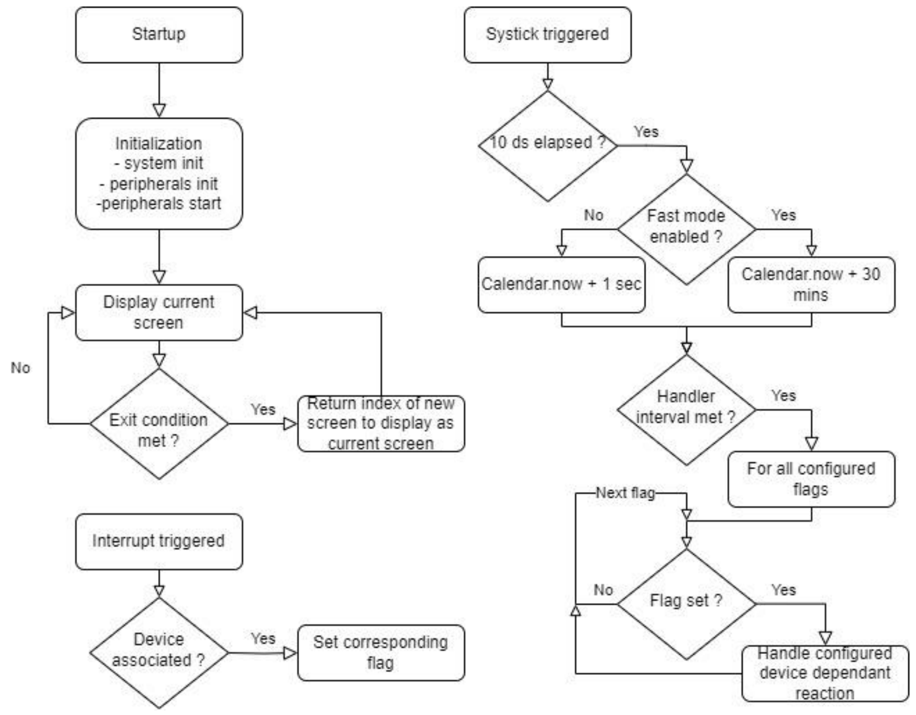
- position integer corresponding to the current position angle of the servo.

- Temperature Sensor “Temperature”
    - init function to configure used pins.
    - start function to reset the clock and send an impulse corresponding to starting a new reading as specified by the device’s documentation.
    - stop function to disable this device’s interrupts until the flag is acknowledged.
    - get function to get the last converted data and return it in degree Celsius.

- Timer
    - init function to configure system clock.
    - ds unsigned long long to keep track of the current decisecond. Using unsigned long long to ensure to always have enough space for counting deciseconds without having to reset it on runtime.

3. Main software structure

When it comes to the execution of the system, we have a main loop displaying the current screen interleaved with the handlers for the different peripherals interrupts setting corresponding flags, and a function triggered very 10 deciseconds to handle said flags.
Here is the corresponding flowchart:



4. Fields of improvements

Setting a critical eye on our work is crucial for advancing our competencies as programmers and engineers. Whether during the active development phase or through post-implementation reviews, identifying potential shortcomings allows us to address issues and discover opportunities for enhancements. This can involve refining the user experience or streamlining our workflows to make our work on the project more efficient and effective. Here are some of the areas that could have been enhanced if given more time:
More extensive memory management

The system already ensures that measures are taken when running out of memory to keep the program running. However, more testing and research would lead to a more memory optimized code using more appropriate types for variables and periodic memory cleanup when specific data will not be needed anymore.

Error handling and notification
During development, some errors were encountered at runtime, leading to undefined behavior. Catching those errors and finding the cause in bare metal programming can be challenging, so a more extensive error handling in different functions would make our work more efficient. The display could be used to indicate the type of error, or the controller's

amber LED could be used as well to give specific error codes corresponding to the reason for the error.

Deeper testing

All the features are tested to ensure the working conditions of the system, however some more extensive stress testing of the different components could reveal some specific cases when the system could fail. Finding those issues and fixing them would have led to a more stable system and overall better quality of the product.

Conclusion

The Smart Home Project achieved its goal of creating an interactive home climate monitoring system using the Atmel SAM3X8E embedded computer platform. The project used several peripherals including temperature and light sensors, a keypad, servo motor and display each contributing to specific functionalities.

The software focused on efficient programming practices and management of global variables, using structures for each peripheral. Challenges inherent to C programming in embedded systems, especially regarding memory management and code efficiency , were addressed through appropriate typing, frequent checks, and optimizations in algorithm depth.

We successfully implemented essential requirements to the project, such as a calendar system, recurrent temperature recording, display of recorded data, sun tracking system, temperature alarms, and a fast mode for testing. The software structure for each peripheral was detailed, emphasizing clear initialization and function procedures.

While the expected features of the project are present, possible improvements to the system were identified. These include enhancing memory management, a more extensive error handling and notification systems, and conducting deeper stress testing to ensure system stability.

In the end, this project not only marked a successful culmination of our efforts but also played a pivotal role in our learning journey. It laid the base for understanding embedded systems intricacies, instilled confidence in deciphering technical documentation, and provided firsthand experience in overcoming challenges inherent in such projects. This project, with its achievements and lessons, stands as a cornerstone in our exploration of the field of embedded systems.