LEHRSTUHL FÜR RECHNERARCHITEKTUR
UND PARALLELE SYSTEME

TECHNISCHE UNIVERSITÄT MÜNCHEN

Lars Leimbach

Sensor report - Group 8

Contents

1 Software architecture **1**
1.1 Architecture 1
1.1.1 main.c/.h 1
1.1.2 mySetup.c 1
1.1.3 nvs.c 1
1.1.4 webFunctions.c 3
1.1.5 countingAlgo.c 3
1.2 Difficulties and Discarded Ideas 4

2 Comparison with Other Groups **6**

Bibliography **9**

1 Software architecture

1.1 Architecture

1.1.1 main.c/.h

The `main.h` file contains important, global definitions. In cases when we want to send data to elastic search without using `SEND_DATA`. It is possible with the `SEND_EVERY_EVENT` definition, to send every event to the database. However, this idea is discarded (for further details see section 1.2).
The `main.c` only calls `start_counting_algo()` after `my_setup()`, as we can see in figure 1.1.

1.1.2 mySetup.c

`my_setup()` function is defined in the `mySetup.h`. This header file contains all definitions relevant for the whole program. For example, `SIZE_BUFFER` which is the size of the buffer,buffers events registered by the barriers. In addition, it imports `main.h` such that every file has access to relevant information. We preferred a central file of definitions instead of distribution over all header files.
One interesting global example function is `error_message()`. This function adds a string to an error-array in the IoT-platform.
`mySetup.c` contains the implements this definition. Additionally, this implementation needed a function that replaces spaces with underscores for the HTTP GET message. Further, `mySetup.c` initializes following: the TaskHandles, that are needed in the test mode; the LCD display; NVS storage; SNTP; MQTT; and the GPIO pins. We placed the update function for the external display `displayCountPreTime()` in `mySetup.c` to make it possible to update the external display everywhere in the project.

1.1.3 nvs.c

The NVS is used to implement sending the count every 5 minutes to the elastic search database. Every key in the NVS can store 4000 Byte (= 4000 characters). However, for storing strings it is possible that ESP is unable to copy the string to the next pages leading to a git issue [1], which causes a `ESP_ERR_NVS_NOT_ENOUGH_SPACE`. That is why, we choose `sizeBuffer = 1500` Bytes for each of the `NUM_KEY_WORDS 7` many keys. Tests showed that no page errors occur with this size. Additional tests showed that about 25 count-events can be stored in the NVS. Before going into details, we noticed that the NVS might be a reason for the bad performance of the first counting algorithm (for further details see section 1.2).
Additionally, we used the cJSON library. This is used in `initNVS_json()` function, which

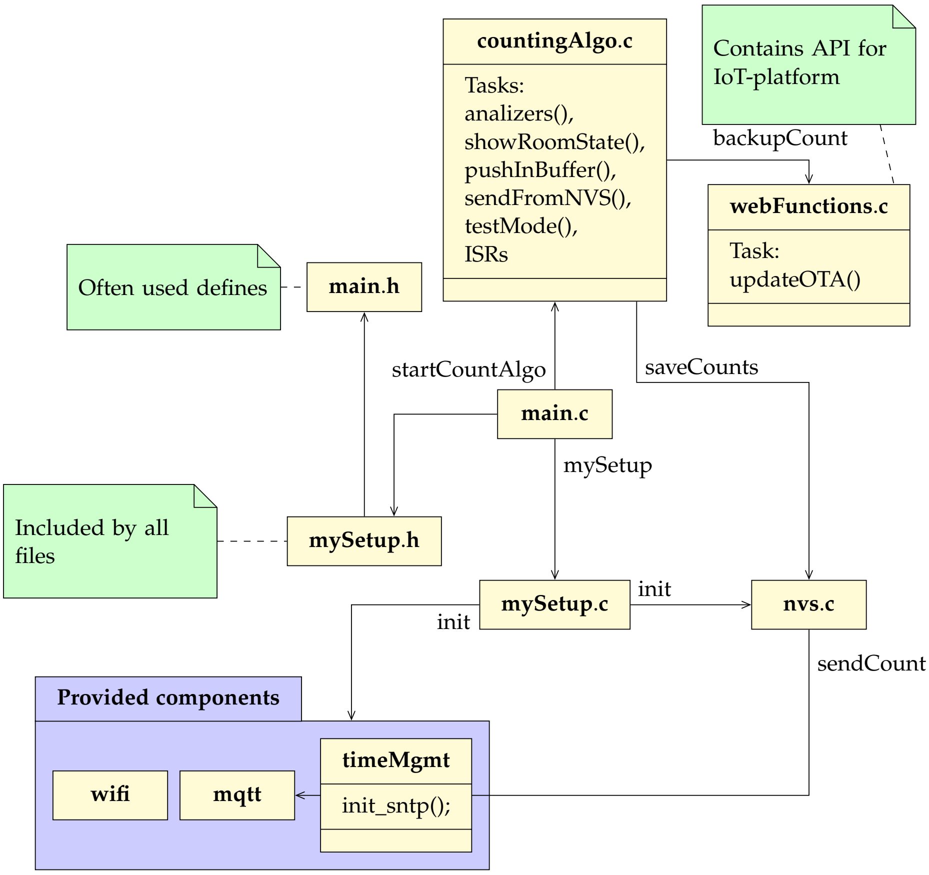


Figure 1.1: An overview of the software architecture for the counting algorithm on the ESP32.

creates a new json file with the required schema for the current key `keyWords[nvs_index]`. Here, `keyWords[]` contains all 7 keys and `nvs_index` is the current selected key with free space. It will be increased if the current key is full. Because this function is called with an open or closed NVS handle, there is the option to open the NVS handle.
This option occurs in `sendDataFromJSON_toDB()` too, which is periodically called every 5 minutes. This function deletes every key in the NVS until `nvs_index`, after sending its content with MQTT to the database. Finally, it creates a json file with `initNVS_json()` at `nvs_index = 0`.
The most important function is `writeToNVM(sensorName,peopleCount,state,time)` which appends to the array `sensorName`, a json dictionary with `peopleCount` at time `time`. `State` was is in a discarded idea. Before `addEventToStorage()` stores the event to the NVS, it checks whether we can store data. if we expand the limit of `NUM_KEY_WORDS 7`, the ESP sends the data to MQTT, otherwise, it creates a new index with `initNVS_json()`.
All these functions use heap allocation.

1.1.4 webFunctions.c

We can see in figure 1.1 that this files spawns one of 5 processes. `updateOTA()`'s function is self describing. In addition, it contains a memory leak detection which is necessary, but it never gets activated. And there is the option to restart the device over the IoT-platform. This feature was never used due to a discarded idea (sending the log via the network with [2]).
More over, this file contains the API for the IoT-platform using a function: to fetch number with a given key, to fetch the count's backup, and to send a system report. We save an array as a string (which contains the system reports) such that there are converting operation with the cJSON library while adding a new string to that array.

1.1.5 countingAlgo.c

The heart of the program is the counting algorithm which is started in the `main.c`. We start with the local variables, that are only accessible in `countingAlgo.c`. A `QueueHandle` is used to get sequentially the sensor's events in the right temporal order. `SemaphoreHandles` ensure read and write operations for multiple tasks for the count variable, the buffer, and whether the test mode button was pressed. The buffer has many flags and variables, we mention later. Next to these are the `Barrier_data` that is a struct containing the data for one sensor/barrier event. Affirmatively, `count`, `prediction`, and `state_counter` (state of the state machine) are local variables too. The ISRs have debouncing so that there are times-tamps for each ISR. The task `pushInBuffer()` waits for an event in the `QueueHandle`, and it pushes it to the buffer. For this action we need the Semaphore to block the buffer to avoid inconsistencies and to increase `fillsize`. This task and the task `anlyzer()` share the same high priority. Before executing the state machine, `checkBuffer()` tells the counting algorithm with `buffer_count_valid_elements` how many elements should be analyzed in a row. This variable is set to 4 if the buffer contains more than 4 events. However, `checkBuffer()` will empty the buffer and reset the state of the state machine, if there are no new events after

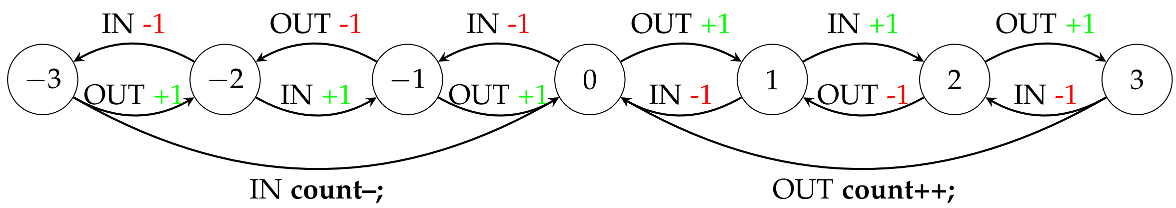


Figure 1.2: The state machine. The numbers in the circles represent the current state of the state machine. The colored number is the operation for the state, the bold text is the operation for the count of people. "IN" stands for an event for the indoor sensor and "OUT" for an event of the outdoor barrier.

`TIME_TO_EMPTY_BUFFER` seconds. This function is not needed with artificial events but might work well in a real life environment. Then, the `anlayzer()` uses a state machine (see figure 1.2). A previous counting approach can be found in section 1.2. We found out that if the state is even an OUT event increases and an IN event decreases the current state. Vice versa for an odd state. This reduces the code. After reaching state -4 or 4, there are sanity checks whether the count is negative or greater than 250. Then, the count is written with `writeToNVM()` to the NVS, and we make a backup to the IoT-platform. Every time, we delete the `head` of the buffer and there is the discarded option to send the analyzed event to the NVS.

The next task `showRoomState()` updates every `REFRESH_RATE_DISPLAY` seconds the external display.
Task `testMode()` activates a blinking text "TEST" on the display and blinking red LED. In addition, it pauses all tasks except `testMode` and removes the ISRs of the sensors. This action is done or redone only if the ISR of the test button gives a semaphore.
The last task `sendFromNVS()` calls every `SEND_COUNT_DELAY` seconds (which are normally 5 min) the function to back up the count to the NVS and to IoT-platform. Finally, it calls `sendDataFromJSON_toDB()`. Further, this task checks whether it is time to reset the count or fetch the prediction from the IoT-platform. It gets the current time with `localtime(time(NULL))`. The last three mentioned tasks do not have a high priority because the ESP should focus on the counting operations, which are tasks `analyzer()` and `pushInBuffer()`.

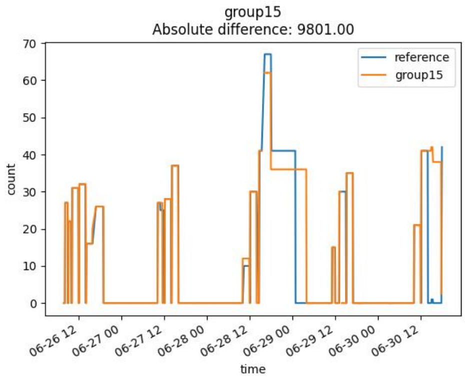
1.2 Difficulties and Discarded Ideas

In the beginning, we had problems defining global variable because of forgetting c specific syntax like `extern` and circular imports of header files. We used Timers. However, it was too complicated to expand the needed stack size, such that we used tasks to periodically do things.
One idea was to send every sensor event to the elastic search platform to train a model that might detect even incorrect sequence of events as an in-going event. The motivation for that was to avoid counts over 70 or higher. There were huge NVS bugs, such that the NVS was

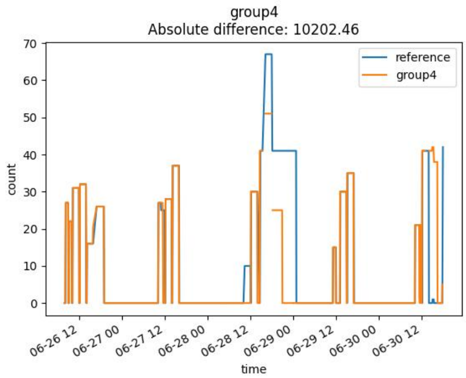
not able to process this high amount of data. In addition, the count and all events were only stored in the NVS such that after a crash everything was deleted. We tried to implement recovery routines, but they were too complex. So using the web client and fixing the NVS as described in section 1.1.3 was the better approach.
Another approach for the counting algorithm was to search in the buffer for the sequence of IN (state 1), OUT (state 1), IN (state 0), OUT (state 0) for an out-going event and visa vie for an in-going event. There was a sanity check whether the events are in the correct temporal order; however, this (and the state of the sensor) is not needed due to the queue. Unfortunely, this algorithm did not perform as good as a state machine with artificial data. That is why, we implemented a state machine, thus, there where situations were this search-algorithm performed better as one group's state machine.

2 Comparison with Other Groups

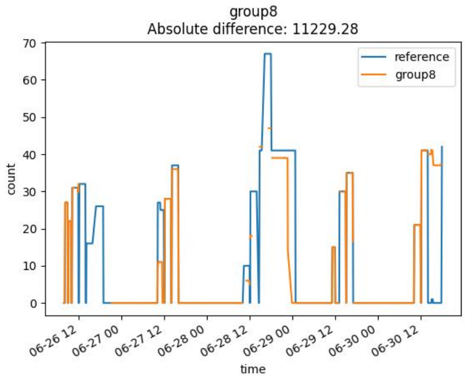
First, we used the reference count's timestamps as bins. One bin covers a timespan of 5 minutes. If there are multiple count of a group in one bin, we calculated the average. Because there are about 1900 bins for these 4 days, an offset of 3 or 4, to the reference, results in a high absolute difference of about 10000. We can see in figure 2.1 and 2.2 the group's count compared to the reference count. To compare the group's counting algorithm performance better, we calculated the absolute difference to the reference count. Null values were replaced by zero for this calculation.
Resulting, group 15 had the best performance and our counting algorithm (group 8) is on the third place for the metric absolute distance.



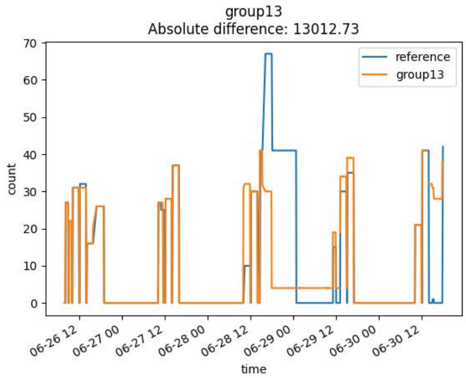
(a) Place 1 of 9



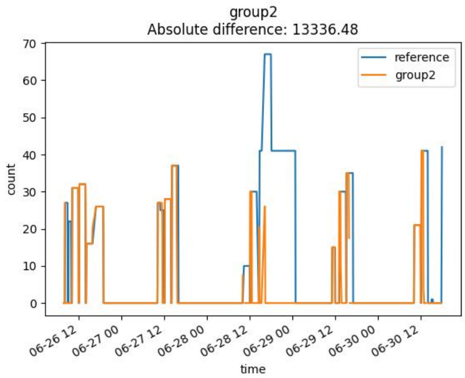
(b) Place 2 of 9



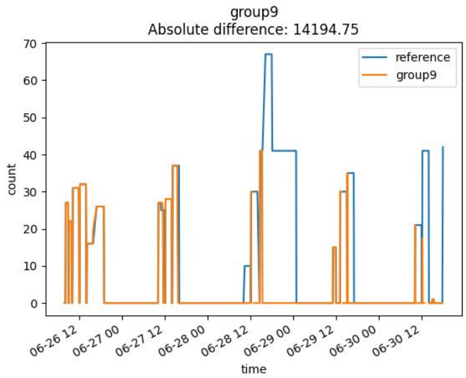
(c) Place 3 of 9



(d) Place 4 of 9

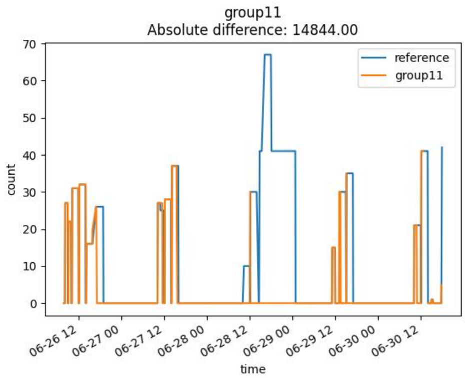


(e) Place 5 of 9

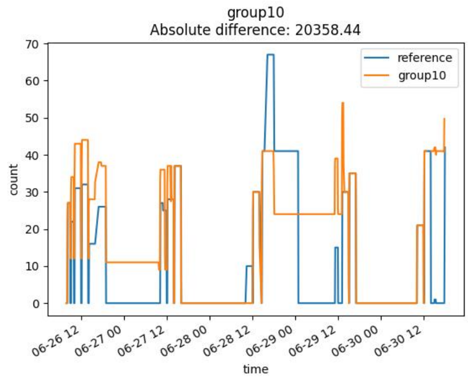


(f) Place 6 of 9

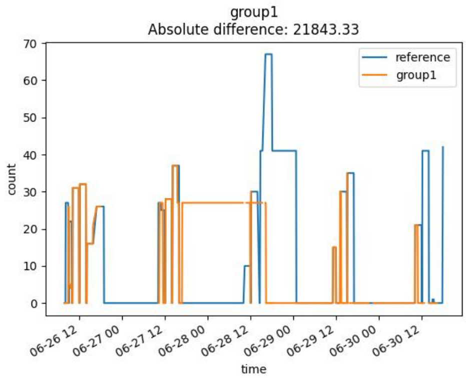
Figure 2.1: Group comparison. The absolute difference (in seconds) is the absolute difference to the reference count. The pictures are ordered descending by their counting algorithm performance.



(a) Place 7 of 9



(b) Place 8 of 9



(c) Place 9 of 9

Figure 2.2: Group comparison. The absolute difference (in seconds) is the absolute difference to the reference count. The pictures are ordered descending by their counting algorithm performance.

Bibliography

[1] *Git Hub Issue about NVS pages*. https://github.com/espressif/esp-idf/issues/5977. [Online; accessed 3-July-2023]. 2023.

[2] *Git Hub repo for sending the log over the network*. https://github.com/nopnop2002/esp-idf-net-logging. [Online; accessed 4-July-2023]. 2023.