Queensland University of Technology
CAB403: Systems Programming

Car Park Management System and Simulator Submission

Due: 11:59 pm Friday 28th October (Week 13)

Name: Jol Singh

ID Number: n10755756

Contents

1 Project Overview

The project specifications have tasked us to develop three individual pieces of software relating to a car park management system which consists of a 'manager', 'simulator', and 'firealarm'. Nonetheless, the car park management system also known as the 'manager' handles automated aspects of the car park including interactions with several hardware components stored within the shared memory segment such as the LPR (license plate reader), entrance, and exit boom gates along with electronic information displays to ensure efficient operation, and accurate reporting of the car park.

Moving on, as the name suggests, the 'simulator' simulates movements of cars throughout the system that utilises aforementioned hardware established within our shared memory. Lastly, the fire alarm system interacts with temperature sensors located on each level of the car park and follows the required safety procedures if a fire is detected, furthermore, this component is required as per the specifications to be a safety critical system, therefore, this report details the assessment, and reconstruction of the fire alarm system to be aligned with the safety-critical coding standards.

2 Group Contribution

ID	Member	%	Contribution
n10755756	Jol Singh	100	Manager, Simulator, Fire Alarm, Report, Video

3 Statement of Completeness

Simulator

- Functionality successfully implemented.

Manager

- Functionality successfully implemented.

Fire Alarm

- Safety Critical Analysis of Fire Alarm Code
- Fixed Fire Temperature Detection
- Rate of Rise Fire Detection
- Smoothing Data Correctly Calculated

4 Known Bugs

The functionality required by the specifications proves to work efficiently (see demonstration video) after thorough and rigorous testing.

5 Fire Alarm Analysis

We conducted thorough analysis of the provided fire alarm file by referencing the code against 'NASA's The Power of 10' in order to decipher its current MISRA C compliance, moreover, the analysis also helped in deciding the approach to take when it came to fixing the problem encapsulated within the file, which was to re-write the entire program from scratch. Nevertheless, after analysing the file, numerous breaches of the safety-critical standards were identified; the following table highlights some of the breaches discovered:

5.1 Safety-Critical Standards

Safety-Critical Rules	Fail	Explanation
Use of runtime assertions in each function (minimum 2).	No runtime assertions implemented.	Assertions should be implemented to pre and post-test conditions of functions, variables, and any other syntaxes.
Restrict function length to a single printed page.	Line 56 – 128: '**tempmonitor()**' exceeds 60-line limit. **(Appendix A)**	Each function should be a logical unit in the code that's understandable and verifiable as a unit. Excessively long functions are often a sign of poorly structured code. (Gerard J. Holzmann, 2006)
Give all loops fixed bounds.	Line 61: Infinite for loops. **(Appendix B)**	Loop bounds prevent runaway code. (Gerard J. Holzmann, 2006)
Do not use dynamic memory allocation after initialisation.	Line 82: Memory Allocation '**malloc()**' used several times. **(Appendix C)**	Memory allocators often have unpredictable behaviour which can impact performance. (Gerard J. Holzmann, 2006)
Restrict all code to very simple control flow constructs.	Line 159: Use of '**goto()**' not permitted. **(Appendix D)**	The '**goto()**' function should be avoided as they alter sequential flow. (Dinesh T, 2020)
The use of the pre-processor must be limited to the inclusion of header files and simple micro definitions.	Line 165: Use of '**fprint()**' from stdio.h library. **(Appendix E)**	'stdio.h' shouldn't be included in the pre-processor statements due to the lack of type-safety.

6 Fire Alarm Safety Code & Rules

As mentioned previously, from the analysis of the fire alarm code, we concluded that the best course of action was to re-write the entire code in order to comply with MISRA coding standards. Furthermore, during development of the new fire alarm system, a couple of procedures were established to ensure compliance was achieved, this included having a firm understanding of MISRA coding guidelines, continuously inspecting potential violations, setting baselines, as well as prioritising serve violations. Ultimately, these procedures allowed our code to be simplified, and act in accordance with the safety-critical conditions.

Moving on, we've included the following section below to outline the application of safety-critical standards within our revised fire alarm, which allows us to showcase how our rewritten code follows best practices based on 'The Power of 10: Rules for Developing Safety-Critical Code' by Gerard J. Holzmann.

6.1 Rule 1

Rule 1 requires code to be written only using simple control flow constructs, this translated into the fire alarm system code as it was rewritten to be simple having only the most basic flow controls such as 'for' and 'while' loops.

6.2 Rule 2

Rule 2 requires all loops to have an upper bound to prevent runaway code. We implemented this rule by setting a global loop limit to 1e9, which meant that every loop within the program had an upper bound limit, and if it were exceeded the program would terminate. Furthermore, the function for terminating the loop was written in a way where it takes an integer, loops the counter, and eventually terminates if it's greater or equal to the loop counter.

```
147  // Function checks if the upper bound of a loop exceeds or matches limit
148  // and terminates the program with an error message if true
149  void loopLim(int i)
150  {
151      if (i >= LOOPLIM)
152      {
153          printf("Error: Upper Bound Loop Limit Exceeded. Program Exiting...");
154          exit(1);
155      }
156  }
157
```

Figure 1 Loop Limit Function

6.3 Rule 3

Rule 3 states that the use of dynamic memory allocation after initialisation like ‘malloc’ should be avoided, but, unfortunately, we weren’t able to meet this requirement, and wasn’t implemented in our new fire alarm system.

6.4 Rule 4

Rule 4 states all functions should be less than 60 lines of code, which was implemented while we coded to ensure all functions were kept to less than 60 lines.

6.5 Rule 5

Rule 5 requires a minimum of two assertions per function. Surprisingly, prior to fixing the fire alarm, the code had no assert statements in any of the functions, nonetheless, we added assert statements when re-writing the code, and implemented them in a way that if removed the execution code isn’t altered.

6.6 Rule 6

Rule 6 requires data to be declared in the smallest scope possible, but due to the nature of the code, and specifications some of the data have a vast scope, but, nevertheless, we followed this requirement to the best of our ability when re-writing the code.

6.7 Rule 7

Rule 7 requires all non-void call functions must have return values verified, and all functions must verify their own provided parameters. Interestingly, from the unsafe code, the only function that returns a value is the ‘compare’ function which takes in two values and returns the difference of them. Nevertheless, we followed this requirement closely, and discovered that ‘tempmonitor’ and ‘deletenodes’ were also two functions that required an integer value, so while re-writing the code, we’ve included a piece of code that verifies weather or not an integer value has been inputted.

6.8 Rule 8

Rule 8 requires pre-processor's to be limited to the inclusion of header files, and simple macro definitions. We followed this requirement within our revised fire alarm code as the pre-processor is used for simple macros such as ‘defines’, as well as ‘shm’, and ‘queue’ header files.

6.9 Rule 9

Rule 9 requires all pointers to be restricted, allowing a simplified program that allows the use of tool-based analysers. Nevertheless, this rule was followed quiet closely as pointers were kept to minimum when re-writing the new fire alarm code.

6.10 Rule 10

Rule 10 requires code to be complied with all warnings enabled. Obviously, as per the specifications, strictly state that warnings flags need to be included in the MakeFile to ensure all errors are enabled and warnings are treated as errors, and do not permit compilation. In addition, we double-checked the program for any errors or warnings by utilising a third-party software called ‘VisualCodeGrepper’, and any errors/warnings were fixed when they appeared.

7 Safety-Critical Concerns

Our main safety-critical concern was when we discovered that the scope for some of the data was quite large, especially the fire alarm system, but, due to the nature of the specifications, we believe that the scope was scaled down to the smallest degree possible for as much of the code as possible.

8 References

[1] Web.eecs.umich.edu. 2006. *The Power of 10: Rules for Developing Safety-Critical Code*. [online] Available at: https://web.eecs.umich.edu/~imarkov/10rules.pdf [Accessed 22 October 2022].

[2] Thakur, D. (2020) *Why to avoid Goto in C, Computer Notes*. [online] Available at: https://ecomputernotes.com/what-is-c/control-structures/why-to-avoid-goto-in-c [Accessed 22 October 2022].

[3] Wikipedia. (2021). *The power of 10: Rules for developing safety-critical code*. [online] Available at: https://en.wikipedia.org/wiki/The_Power_of_10:_Rules_for_Developing_Safety-Critical_Code [Accessed October 22, 2022].

Appendix A – tempmonitor()

```c
void tempmonitor(int level)
{
    struct tempnode *templist = NULL, *newtemp, *medianlist = NULL, *oldesttemp;
    int count, addr, temp, mediantemp, hightemps;

    for (;;) {
        // Calculate address of temperature sensor
        addr = 0150 * level + 2496;
        temp = *((int16_t *)(shm + addr));

        // Add temperature to beginning of linked list
        newtemp = malloc(sizeof(struct tempnode));
        newtemp->temperature = temp;
        newtemp->next = templist;
        templist = newtemp;

        // Delete nodes after 5th
        deletenodes(templist, MEDIAN_WINDOW);

        // Count nodes
        count = 0;
        for (struct tempnode *t = templist; t != NULL; t = t->next) {
            count++;
        }

        if (count == MEDIAN_WINDOW) { // Temperatures are only counted once we have 5 samples
            int *sorttemp = malloc(sizeof(int) * MEDIAN_WINDOW);
            count = 0;
            for (struct tempnode *t = templist; t != NULL; t = t->next) {
                sorttemp[count++] = t->temperature;
            }
            qsort(sorttemp, MEDIAN_WINDOW, sizeof(int), compare);
            mediantemp = sorttemp[(MEDIAN_WINDOW - 1) / 2];

            // Add median temp to linked list
            newtemp = malloc(sizeof(struct tempnode));
            newtemp->temperature = mediantemp;
            newtemp->next = medianlist;
            medianlist = newtemp;

            // Delete nodes after 30th
            deletenodes(medianlist, TEMPCHANGE_WINDOW);

            // Count nodes
            count = 0;
            hightemps = 0;

            for (struct tempnode *t = medianlist; t != NULL; t = t->next) {
                // Temperatures of 58 degrees and higher are a concern
                if (t->temperature >= 58) hightemps++;
                // Store the oldest temperature for rate-of-rise detection
                oldesttemp = t;
                count++;
            }

            if (count == TEMPCHANGE_WINDOW) {
                // If 90% of the last 30 temperatures are >= 58 degrees,
                // this is considered a high temperature. Raise the alarm
                if (hightemps >= TEMPCHANGE_WINDOW * 0.9)
                    alarm_active = 1;

                // If the newest temp is >= 8 degrees higher than the oldest
                // temp (out of the last 30), this is a high rate-of-rise.
                // Raise the alarm
                if (templist->temperature - oldesttemp->temperature >= 8)
                    alarm_active = 1;
            }
        }

        usleep(2000);

    }
}
```

Appendix B – Infinite for loop

```
61      for (;;) {
```

Appendix C – Memory Allocation

```
82          int *sorttemp = malloc(sizeof(int) * MEDIAN_WINDOW);
```

Appendix D – goto()

```
159             goto emergency_mode;
```

Appendix E – fprint()

```
165     fprintf(stderr, "*** ALARM ACTIVE ***\n");
```