CAB403 Carpark Management System - Comprehensive Analysis -

Darcy Truman `n10517651`

Don Kaluarachchi `n10496262`

Jack Dittman `n10485911`

Cody Van Beek `n10491996`

November 2021

Contents

1 Group Contribution

Member	**%**	**Contribution**
Jack Dittman	25	Manager, Simulator, Editor
Don Misura Kaluarachchi	25	Firealarm, Report
Darcy Truman	25	Firealarm, Report
Cody Van Beek	25	Manager, Simulator, Video

2 Statement of Completion

Simulator

- All functionality has been successfully implemented.

Manager

- All functionality has been successfully implemented.

Fire Alarm

- Safety critical analysis of fire alarm code
- Fixed temperature fire detection
- Rate of rise fire detection
- Correctly calculated smoothing data

3 Known Bugs

- After rigorous testing, all functionality required by the specification has proves to work efficiently (see demonstration video attached at the end of the document).

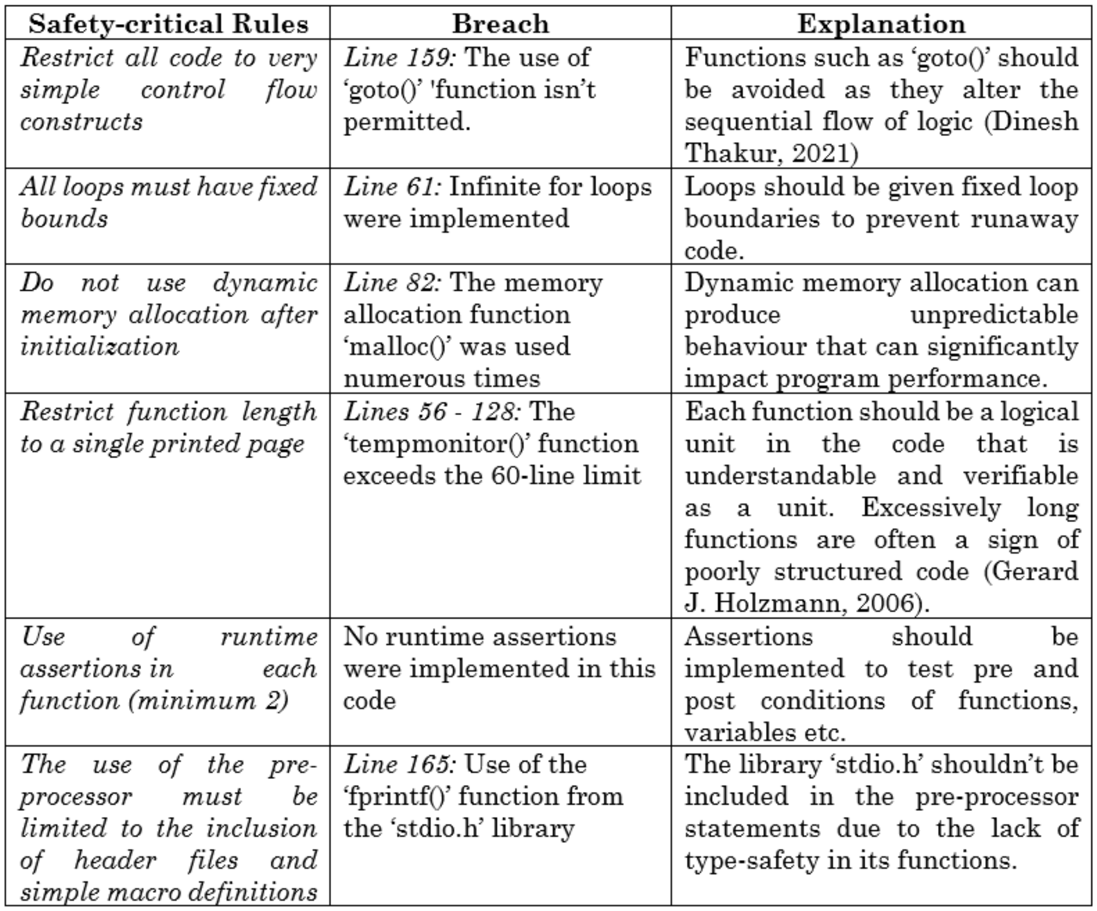
4 Project Overview

The specification of this project was to develop numerous pieces of software, establishing a configurable multi-level car park management system. The required files consisted of a 'manager', 'simulator' and 'firealarm' that were all communicating and hence operating upon a shared memory segment. As the name suggests, the responsibility of the 'manager' file was handling the automated aspects of the car park. This includes the interaction with a number of hardware components stored within the shared memory segment, such as LPR (license plate readers) sensors, entrance and exit boom gates and electronic information displays, hence ensuring efficient operation and detailing of the car park. Another significant segment of this software system was the car park 'simulator' that simulates the movement of cars throughout the system, utilising the aforementioned hardware established in the shared memory. The final software component required by this project was the fire alarm system which interacts with temperature sensors located on each level of the car park, following required safety procedures upon the detection of a fire. This component was required by the specification to be a safety critical system, and hence this report details the assessment and reconstruction of the fire alarm to align with the safety-critical coding standards.

5 Original Fire Alarm Analysis

Prior to deciding on the approach to take for fixing the problems encapsulated within the original fire alarm file, an analysis was conducted by referencing the code against 'NASA's power of 10' to decipher its current MISRA C compliance. Through this analysis numerous breaches to the safety critical standards were located, some of which can be seen in the table below.

5.1 Failed Safety-critical Standards



Safety-critical Rules	Breach	Explanation
Restrict all code to very simple control flow constructs	*Line 159:* The use of 'goto()' 'function isn't permitted.	Functions such as 'goto()' should be avoided as they alter the sequential flow of logic (Dinesh Thakur, 2021)
All loops must have fixed bounds	*Line 61:* Infinite for loops were implemented	Loops should be given fixed loop boundaries to prevent runaway code.
Do not use dynamic memory allocation after initialization	*Line 82:* The memory allocation function 'malloc()' was used numerous times	Dynamic memory allocation can produce unpredictable behaviour that can significantly impact program performance.
Restrict function length to a single printed page	*Lines 56 - 128:* The 'tempmonitor()' function exceeds the 60-line limit	Each function should be a logical unit in the code that is understandable and verifiable as a unit. Excessively long functions are often a sign of poorly structured code (Gerard J. Holzmann, 2006).
Use of runtime assertions in each function (minimum 2)	No runtime assertions were implemented in this code	Assertions should be implemented to test pre and post conditions of functions, variables etc.
The use of the pre-processor must be limited to the inclusion of header files and simple macro definitions	*Line 165:* Use of the 'fprintf()' function from the 'stdio.h' library	The library 'stdio.h' shouldn't be included in the pre-processor statements due to the lack of type-safety in its functions.

6 Fire Safety Code

After thorough examination of the unsafe fire alarm system it was determined that the best course of action would be to rewrite the entirety of the fire alarm code from scratch. In order to achieve compliance with the MISRA coding standard, a number of steps were enforced whilst developing the new fire alarm. These include, ensuring we all had extensive knowledge on the MISRA coding guidelines, continuous inspection for potential violations, setting baselines and prioritising unpreventable violations based on their severity. This allowed for the code to be simplified and hence in accordance with all of the safety-critical conditions.

The following section outlines the application of the safety-critical standards within our new fire alarm file. These rules will be based on "The Power of 10: Rules for Developing Safety-Critical Code" report by Gerard J. Holzmann, NASA/JPL Laboratory for Reliable Software and will hence explore how the rewritten code follows best practices for the development of safety-critical software systems.

6.1 Rule 1

The first rules requires that the code is written only using simple control flow construct. As the fire alarm code was rewritten, it was as simple as restricting the code to only the most basic flow controls such as 'for loops' and 'while loops'.

6.2 Rule 2

This rule requires that all loops have an upper bound to prevent runaway code. A loop limit was set to 1e9, all loops in the program were given this loop limit as an upper bound where they upper bound for the loop. Additionally, a loop termination function was written that takes an integer, the loop counter, and will terminated the program after writing an error message if it is greater than or equal to the loop counter. The loop termination function is shown below.

```
void loopLim(int i)
{
  if (i >= LOOPLIM)
  {
    printf("ERROR: Loop Limit Upper Bound Exceeded, Program Will Close");
    exit(1);
  }
}
```

6.3 Rule 3

Rule 3 simply states to not use dynamic memory allocation after initialisation, for example 'malloc'. This requirement was kept in mind and left out of the new version of the fire alarm system.

6.4 Rule 4

This rules mandates that all functions must be less than 60 lines of code. This rule was kept in mind while writing the code and all functions we kept to less than 60 lines, as to be able to be printed on a standard piece of paper.

6.5 Rule 5

Statistics show that for every 10 to 100 lines of code there is at least one error. Rule 5 therefore states that every function should have a minimal of two assertions per function.
In the unsafe code there are no assert statements in any of the functions. This was kept in mind and assert statements were added when re-writing the new fire alarm. These assert statements were added in a way that it performs but when removed does not cause the execution of the code to change.

6.6 Rule 6

Rule 6 mandates that data is declared in the smallest scope possible. This rule was followed to the best of our abilities but due to the nature of the code and the assignment, some pieces of data have a very large scope. This is certainly one part of the code that can be a potential safety-critical concern.

6.7 Rule 7

This rule states that all non-void call functions must have their return values verified and all functions must verify their provided parameters. The only function in this code that returns a value is the 'compare' function, which takes in two values and returns the difference of the two. However, two functions require integer input values, 'tempmonitor' and 'deletenodes'. A function called 'isInt' was written to verify if an input value is an integer or not. The method takes in a reference to the value and will return a 0 if the input variable isn't an integer and 1 is it is. The method that was created is shown below.

```
int isInt(void *in)
{
  int ret = 0;
  if (isdigit(in) != 0)
  {
    ret = 1;
  }
  return ret;
}
```

The 'fire.c' code was adjusted and the method above was implemented in the aforementioned locations.

6.8 Rule 8

This rule required that the use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, variable argument lists (ellipses), and recursive macro calls are not allowed. The new fire alarm code passes this requirement, as the preprocessor is used for simple macros such as 'defines', and to include the 'sharedMemoryOperations' and 'carQueue' header files.

6.9 Rule 9

This rule states that the use of pointers must be restricted, this allows for the flow of the program to remain relatively simple and to allow for the use of tool-based analyzers. Pointers were kept to a minimum or not used at all when re-writing the new safe fire alarm.

6.10 Rule 10

This final rule states that the code must be compiled using all warnings enabled. This rule was followed and code was compiled and tested with the most pedantic settings. Additionally, 'VisualCodeGrepper' was used to analyse the code for any and all errors or warnings, which were promptly fixed.

6.11 Safety Critical Concerns

The first safety critical concern that was discovered was that the scope of some of the data is very large, in some cases the entire fire alarm system. However, due to the nature and structure of the project, I believe that the scope has been minimised to the smallest degree possible for as much of the code as posible.

7 References

Dinesh, T. (2021). Why to avoid goto in C. ecomputernotes.
`https://ecomputernotes.com/what-is-c/control-structures/why-to-avoid-goto-in-c`

Holzmann. G. J. (2006). The Power of 10: Rules for Developing Safety-Critical Code. comuter.org
`https://web.eecs.umich.edu/~imarkov/10rules.pdf`

8 Demonstration Video

`https://youtu.be/0BXs5Av7DLM`