CAB403 Assignment 2 Report

Reuben Visser - n10478337

Patrick Daly - n11179856

Brandon Falconer - n10468722

Brent Morgan – n10215662

Table of Contents

[Statement of Completeness 2](#_Toc86611103)

[Statement of Contribution 3](#_Toc86611104)

[Assessment of the safety-critical fire alarm system 4](#_Toc86611105)

[Analysis of the provided fire alarm 4](#_Toc86611106)

[Approach to fixing the fire alarm 6](#_Toc86611107)

[Potential safety-critical concerns and reservations of updated fire alarm 6](#_Toc86611108)

[Reference List 7](#_Toc86611109)

[Appendix 8](#_Toc86611110)

[Appendix A: Original firealarm.c 8](#_Toc86611111)

[Appendix B: Updated firealarm.c 13](#_Toc86611112)

# Statement of Completeness

A statement of completeness describing how much of the assignment has been completed and any issues (e.g. known bugs)

# Statement of Contribution

To avoid multiple people acting on the same sections of code at the same time, the sections were divided up so that most of the team members independently worked on individual parts; these parts being the manager, simulator, hash table, fire alarm and then finally the report. Then once all the group members were completed their parts, they were all put together and the process of debugging was undertaken to make all the components work together. Below a list of these components with the students who worked on it.

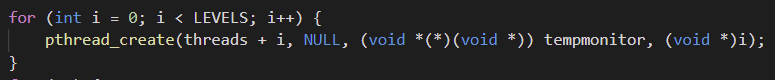
* Manager: Patrick Daly
* Simulator: Reuben Visser, Brandon Falconer
* Shared Memory: Patrick Daly
* Fire Alarm: Brent Morgan, Brandon Falconer
* Hash Table: Brent Morgan
* Report: Brent Morgan, Brandon Falconer, **WHOEVER** **DOES** **STATEMENT** **OF** **COMPLETENESS**

# Assessment of the safety-critical fire alarm system

## Analysis of the provided fire alarm

The original firealarm.c did not comply with various MISRA C rules and directives as well as other general code safety rules such as the ones in NASA’s “The power of 10”. This resulted in the code to be deemed unsafe and in need of a change or rewrite. To decide which is the best approach, each safety flaw needs to be analyzed to see exactly what parts of this code isn’t safely written.

**Use of Threads**Throughout the fire alarm code there is usage of thread libraries, its purpose in this code is so that each of the 5 levels in the carpark are allocated to a thread each and therefore they can then all run at the same time. However, as per MISCRA C rules and directives, no threads are allowed to be used in a safety-critical code, therefore these need to be removed/altered if possible. A clear example of this can be seen in line 155 of the original firealarm.c where the threads for each level are created as seen below.



**Multiple Variable Declarations on the Same Line**Another important MISRA C guideline is that variables are to be declared on separate lines to achieve a safely written code. There are however parts of code in the fire alarm that conflict this rule such as the one shown below on lines 58 and 59 which declare several variables on just 2 lines.



**MISRA Directive 4.12**In the MISRA C guidelines directive 4.12 states that “Dynamic memory allocation shall not be used” (Synopsys 2012), but in 5 different sections, the fire alarm code calls the malloc function which is a method of doing just that. An example of this is in line 152 where the threads pointer gets dynamically allocated memory for each level using the malloc function as seen in the snippet below.



**MISRA C Rule 11.6**Rule 11.6 of the MISRA C guidelines state that “A cast shall not be performed form pointer to void and an arithmetic type” (Synopsys 2012); however, parts of the code in the fire alarm conflict this. A prime example is the one below which is a compare function (line 51) that takes an input called first in the form of a void pointer, but then uses it in the return as an integer pointer.

Text

Description automatically generated

**MISRA C Rule 17.4**Rule 17.4 states that “All exit paths from a function with non*‑void* return type shall have an explicit *return* statement with an expression” (Synopsys 2012) which is true for all the functions in the fire alarm except for the main. The main function should return an integer, generally 1 or 0 as an exit code so that the output can show if the code fully ran but this was not the case in this code as there is no return statements in any of the paths in main. This can be seen in Appendix A showing the original fire alarm code.

**MISRA C Rule 18.3**Rule 18.3 States that “The +, -, += and -= operators should not be applied to an expression of pointer type” (Synopsys 2012). This rule is not followed in several parts of the fire alarm code, an example of which is on line 155 where it tries to add the integer value of “i” onto the pointer variable threads initialised a few lines above in line 152. This code is shown below.

Text

Description automatically generated

**MISRA C Rule 21.3**Rule 21.3 of the MISRA C guidelines states that “The memory allocation and deallocation functions of <stdlib.h> shall not be used”, however in this code it is. An example can be seen below where this can be seen clearly is in line 46 inside the “deletenodes” function which calls the free function from the standard library to free the memory of the variable.

Graphical user interface, text

Description automatically generated

**MISRA C Rule 21.9**Rule 21.9 states that “The library functions *bsearch* and *qsort* of <stdlib.h> shall not be used”. This is not followed in line 87 of the fire alarm as it calls the qsort function to sort the 5 recorded temperature samples and find the median as seen in the code below.

Text

Description automatically generated

**NASA “The Power of 10” Rule 1 & MISRA Rule 15.1**Another important guide to follow to make sure your code is safety-critical is a set of 10 rules called NASA’s The power of 10. Number 1 of which states to “Avoid complex flow constructs, such as goto and recursion” (Holzmann 2006) which is very similar to MISRA’s rule 15.1 which states “The goto statement should not be used” (Synopsys 2012). The fire alarm again fails this criterion by on line 159 having a “goto” call to trigger the emergency mode state of the alarm. This code is seen below.

Text

Description automatically generated

**NASA “The Power of 10” Rule 2**Number 2 of NASA’s The power of 10 rules states that “All loops must have fixed bounds. This prevents runaway code” (Holzmann 2006). This criterion is not met in the original fire alarm as on 4 different parts of the code an infinite for loop is called to run a set of code indefinitely as seen in the code snippet below from line 157. While this is an easy way to constantly check temperatures on all the levels, it is not up to the safety standard the code needs to be.

Text

Description automatically generated

## Approach to fixing the fire alarm

Due to the large number of rule breaches when checking the safety of the original fire alarm, it was thought best to completely rewrite it. One of the main causes of this was that the provided fire alarm relied heavily on the use of threads to constantly check the temperatures on all the levels. However, as per MISCRA C guidelines the use of threads isn’t allowed, meaning this whole process would need to be redesigned. While the use of threads might not be able to get completely removed from the application, they can be used in a much safer way than what it is in the current fire alarm. This along with the several other rule violations not just in MISCRA C but also NASA’s The Power of 10 rules meant that fixing the code to make it safety-critical would be more complicated and time consuming than just rewriting the fire alarm with the MISCRA C and NASA rules in mind.

## Potential safety-critical concerns and reservations of updated fire alarm

For the most part, when creating the new fire alarm, the MISRA C rules were kept in mind to ensure that wherever possible the fire alarm followed them. However, there was one rule that was not able to be followed; thus, meaning that the fire alarm is considerably safe, but not fully compliant with what it should be.

**Use of Threads**It was determined when re-writing the fire alarm that without the use of threads, the desired temperature collection intervals of 20ms apart will not be possible. This because if you were to do the fire alarm without the use of threads, the alternative would be to run it in a big for loop as one large sequential program. If it were to be coded as a sequential program, the processing time it would take to go through all the temp sensors on all the levels, it would most likely take more than 20ms. This timing process without threads would be getting constantly further from the desired amount with every level it has to check. For this reason, the use of threads was incorporated so that all the levels temperature sensors can be checked at the same time. Without the use of threads, due to the timing intervals being longer and more unpredictable during measurements; the fire alarm would not be able to detect a fire as fast and thus making the fire alarm itself unsafe for the people inside the carpark. This implementation can be seen through the include call to the “pthread.h” header on like 9 as seen in the code snippet taken from the updated fire alarm code in Appendix B.

Text

Description automatically generated

Another part of the code that shows this is in the main function where it creates a thread array as seen in the code snippet below.

Text

Description automatically generated with medium confidence

Apart from this one altercation, the new fire alarm implementation follows all the MISRA C and NASA Power of 10 guidelines, deeming it mostly safe, and an improvement on the original.

# Reference List

Synopsys. (2012). *MISRA C:2012: Coding Standard.* California, US: Synopsys.

Gerard J. Holzmann. (2006). *The Power of 10: Rules for Developing Safety-Critical Code.* California, US: NASA.

# Appendix

## Appendix A: Original firealarm.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <pthread.h>

#include <sys/mman.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int shm\_fd;

volatile void\* shm;

int alarm\_active = 0;

pthread\_mutex\_t alarm\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t alarm\_condvar = PTHREAD\_COND\_INITIALIZER;

#define LEVELS 5

#define ENTRANCES 5

#define EXITS 5

#define MEDIAN\_WINDOW 5

#define TEMPCHANGE\_WINDOW 30

struct boomgate {

    pthread\_mutex\_t m;

    pthread\_cond\_t c;

    char s;

};

struct parkingsign {

    pthread\_mutex\_t m;

    pthread\_cond\_t c;

    char display;

};

struct tempnode {

    int temperature;

    struct tempnode\* next;

};

struct tempnode\* deletenodes(struct tempnode\* templist, int after)

{

    if (templist->next) {

        templist->next = deletenodes(templist->next, after - 1);

    }

    if (after <= 0) {

        free(templist);

        return NULL;

    }

    return templist;

}

int compare(const void\* first, const void\* second)

{

    return \*((const int\*)first) - \*((const int\*)second);

}

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);

    }

}

void\* openboomgate(void\* arg)

{

    struct boomgate\* bg = arg;

    pthread\_mutex\_lock(&bg->m);

    for (;;) {

        if (bg->s == 'C') {

            bg->s = 'R';

            pthread\_cond\_broadcast(&bg->c);

        }

        if (bg->s == 'O') {

        }

        pthread\_cond\_wait(&bg->c, &bg->m);

    }

    pthread\_mutex\_unlock(&bg->m);

}

int main()

{

    shm\_fd = shm\_open("PARKING", O\_RDWR, 0);

    shm = (volatile void\*)mmap(0, 2920, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm\_fd, 0);

    pthread\_t\* threads = malloc(sizeof(pthread\_t) \* LEVELS);

    for (int i = 0; i < LEVELS; i++) {

        pthread\_create(threads + i, NULL, (void\* (\*)(void\*)) tempmonitor, (void\*)i);

    }

    for (;;) {

        if (alarm\_active) {

            goto emergency\_mode;

        }

        usleep(1000);

    }

emergency\_mode:

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

    // Handle the alarm system and open boom gates

    // Activate alarms on all levels

    for (int i = 0; i < LEVELS; i++) {

        int addr = 0150 \* i + 2498;

        char\* alarm\_trigger = (char\*)shm + addr;

        \*alarm\_trigger = 1;

    }

    // Open up all boom gates

    pthread\_t\* boomgatethreads = malloc(sizeof(pthread\_t) \* (ENTRANCES + EXITS));

    for (int i = 0; i < ENTRANCES; i++) {

        int addr = 288 \* i + 96;

        volatile struct boomgate\* bg = shm + addr;

        pthread\_create(boomgatethreads + i, NULL, openboomgate, bg);

    }

    for (int i = 0; i < EXITS; i++) {

        int addr = 192 \* i + 1536;

        volatile struct boomgate\* bg = shm + addr;

        pthread\_create(boomgatethreads + ENTRANCES + i, NULL, openboomgate, bg);

    }

    // Show evacuation message on an endless loop

    for (;;) {

        char\* evacmessage = "EVACUATE ";

        for (char\* p = evacmessage; \*p != '\0'; p++) {

            for (int i = 0; i < ENTRANCES; i++) {

                int addr = 288 \* i + 192;

                volatile struct parkingsign\* sign = shm + addr;

                pthread\_mutex\_lock(&sign->m);

                sign->display = \*p;

                pthread\_cond\_broadcast(&sign->c);

                pthread\_mutex\_unlock(&sign->m);

            }

            usleep(20000);

        }

    }

    for (int i = 0; i < LEVELS; i++) {

        pthread\_join(threads[i], NULL);

    }

    munmap((void\*)shm, 2920);

    close(shm\_fd);

}

## Appendix B: Updated firealarm.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <pthread.h>

#include <sys/mman.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <pthread.h>

#include "../shared\_memory/sharedMemory.h"

int shm\_fd;

const char\* key = "PARKING";

shm\* shared\_mem;

size\_t shmSize = 2920;

bool alarm\_active = false;

int num\_levels = 5;

void\* tempmonitor(void\* ptr) {

    int thread = \*((int\*)ptr);

    int temp;

    temp = shared\_mem->levels[thread].tempSen1;

    /\* Monitor Temperatures \*/

    while(temp != 0) {

        int tempList[35];

        int medianList[30];

        int count = 0;

        int addr;

        int medianTemp;

        int hightemps;

        int fixedTempCount;

        bool first = true;

        int iterations = 0;

        /\* Evaluate the first five temperatures before smoothing\*/

        for (int i = 0; i < num\_levels; i++) {

            temp = shared\_mem->levels[thread].tempSen1;

            tempList[i] = temp;

        }

        /\* Evaluate temps 5-35 for smoothing\*/

        for (count = 5; count < 35; count++) {

            temp = shared\_mem->levels[thread].tempSen1;

            tempList[count] = temp;

            int temporaryList[5];

            for (int i = 0; i < 5; i++) {

                temporaryList[i] = tempList[count - 5 + i];

            }

            /\* Sort temp list using selection sort \*/

            int n = sizeof(temporaryList) / sizeof(temporaryList[0]);

            int idx, jdx, min\_idx;

            for (idx = 0; idx < n - 1; idx++)

            {

                // Find the minimum element in unsorted array

                min\_idx = idx;

                for (jdx = idx + 1; jdx < n; jdx++)

                    if (temporaryList[jdx] < temporaryList[min\_idx])

                        min\_idx = jdx;

                // Swap the found minimum element with the first element

                int temp = tempList[min\_idx];

                temporaryList[min\_idx] = temporaryList[idx];

                temporaryList[idx] = temp;

            }

            /\* Find median \*/

            medianTemp = temporaryList[2];

            medianList[iterations] = medianTemp;

            iterations++;

        }

        /\* Calculate fixed temperature fire detection \*/

        fixedTempCount = 0;

        for (int i = 0; i < 30; i++) {

            //printf("median temp: %d \n", medianList[i]);

            if (medianList[i] >= 58) {

                fixedTempCount++;

            }

        }

        if (fixedTempCount >= 27) {

            alarm\_active = true;

        }

        /\* Calculate rate of rise fire detection \*/

        if ((medianList[30] - medianList[0]) > 8) {

            alarm\_active = true;

        }

        usleep(2000);

    }

}

int main()

{

    /\* Locate shared memory segment and attach the segment to the data space\*/

    if ((shm\_fd = shm\_open(key, O\_RDWR, 0)) < 0)

    {

        perror("shm\_open");

        return 1;

    }

    if ((shared\_mem = (shm\*)mmap(0, shmSize, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm\_fd, 0)) == (void\*)-1)

    {

        perror("mmap");

        return 1;

    }

    /\* Create a thread for each level \*/

    pthread\_t threads[num\_levels];

    int level[num\_levels];

    for (int i = 0; i < num\_levels; i++) {

        level[i] = i;

        pthread\_create(&threads[i], NULL, tempmonitor, &level[i]);

    }

    while(shared\_mem->levels[0].tempSen1 >= 0) {

        /\* Activate Alarm \*/

        if (alarm\_active) {

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

            /\* Handle the alarm system and open boom gates

               Activate alarms on all levels \*/

            for (int i = 0; i < num\_levels; i++) {

                shared\_mem->levels[i].alarm1 = true;

            }

            /\* Show evacuation message \*/

            char\* evacmessage = "EVACUATE ";

            for (char\* p = evacmessage; \*p != '\0'; p++) {

                for (int i = 0; i < num\_levels; i++) {

                    pthread\_mutex\_lock(&shared\_mem->entrances[i].SIGN.lock);

                    shared\_mem->entrances[i].SIGN.display = \*p;

                    pthread\_mutex\_unlock(&shared\_mem->entrances[i].SIGN.lock);

                }

                usleep(20000);

            }

            alarm\_active = false;

        }

        usleep(1000);

    }

    munmap(shared\_mem, 2920);

    close(shm\_fd);

}