Memory management and error handling in FreeRTOS for a CubeSat project

Diaa Jadaan

Supervisor: Prof. Kjetil Svarstad

NUTS Project Manager: Roger Birkeland

Contents

List of Figures 3
1 Introduction 4
1.1 Problem description 4
1.2 Project Overview 4
1.3 The CubeSat platform 5
1.4 NUTS design 5
2 Memory management 7
2.1 Memory management schemes in FreeRTOS 7
2.2 Suggesting a memory management scheme 9
2.3 Detecting memory leaks 10
2.3.1 Measuring memory usage 10
2.3.2 Analyzing memory usage 12
2.3.3 Tracking where memory is allocated 12
3 Stack overflow 14
3.1 Manual checking 14
3.2 Runtime stack checking 14
3.3 Suggesting a checking method 15
4 Lock-free circular buffer 16
4.1 Introduction 16
4.2 How does it work 16
4.2.1 Inserting elements into the queue 16
4.2.2 Removing elements from the queue 18
5 Exception handling 19
5.1 The usual way 19
5.2 Using an exception handling framework 20
5.3 Documenting exceptions 21
5.4 Summary 21
5.5 Further work 22
6 Using RAID 4 techniques for error detection in embedded systems 23
6.1 Detection using redundancy 23
6.2 A more manageable way 24
6.3 Operation 26

6.4 Benchmarks 27
7 Future work 29
8 References: 31

List of Figures

Figure 1: Abstract overview of the satellite modules and their connections 6
Figure 2: Transformation rule targeting errors affecting data 23
Figure 3: Transformation targeting errors affecting control instructions 24
Figure 4: Error detection and correction procedure 27
Figure 5: Error detection - worst case scenario 28
Figure 6: Error detection - best case scenario 28

1 Introduction

1.1 Problem description

Over the past years, the main concern of CubeSats was to achieve a simple design that produces a satellite with maximum cost efficiency. Recently, a growing number of researchers started to look into the matters of reliability as an equally important factor for the success of the satellite mission. An example of that is the collaboration between researchers from TU Delft and TU Munich that focusses on general CubeSat reliability aspects as well as performance and reliability aspects of a CubeSat bus architecture. [11] shows a survey that they set up to collect some statistics about the current trends among CubeSat projects in terms of reliability orientation of satellite designs.

Electronic components are vulnerable to a number of effects when exposed to cosmic rays. The collective term for the different failure mode occurrences is Single Event Phenomena (SEP) or Single Event Effects (SEE). The expected SEU error rate for commercial off the shelf (COTS) electronic components in LEO is 10^{-5} error/bit-day. This might appear to be a minuscule number, but with 128kB of RAM it amounts to over 10 errors accumulating per day in orbit [10].

Dealing with errors includes detecting, reporting and correcting these errors. When choosing a detection method there is often a compromise between the selected method's efficiency, its required overhead and how much intervention does it require from the programmer to keep track of the errors. Although the last point is often overlooked, it's not less important than the others. A complicated tracking method can lead a complex software structure which usually makes bugs harder to avoid in addition to producing a code that's hard to maintain.

Apart from error handling, an efficient management of the different tasks running on the satellite's processor is also a critical matter. NUTS software is being designed to operate on top of a real time operating system – FreeRTOS. For the operating system to work in an optimal manner, it should be configured to reflect the design needs, and the correct components (e.g. the .h files) should be chosen.

1.2 Project Overview

The focus of this work has been to explore some of the architectural aspects of FreeRTOS and make some suggestions regarding the use of different options and features of this operating system. This includes the use of an appropriate memory management scheme and stack overflow detection mechanism. In addition to that, a method for detecting memory leaks is introduced and integrated within the FreeRTOS source files.

In addition to that, in chapter 4 we address the need of having an efficient circular buffer implementation that expresses a non-blocking behavior. This feature will allow us to eliminate the need of using semaphores and mutexes, and therefore get rid of some hazards we may face otherwise such as dead locks (e.g. in case of a buggy task).

As mentioned earlier, dealing with errors includes detecting, reporting and correcting them. Chapter 5 deals with the issue of reporting errors in an efficient way that gives the programmer the convenience of working in the way he/she is used to, and yet producing a high quality code. The other common alternative for error reporting depends on setting a convention that requires returning an error code with each function call and passing back the function result as a parameter. This will make it harder to reuse previously written code and introduces the factor of human error.

Chapter 6 uses the technique discussed in chapter 5 to introduce an error detection and correction mechanism that produces a much cleaner code with the same reliability and some extra overhead penalty compared to the other used methods.

1.3 The CubeSat platform

Usually, the term satellite technology is associated with great complexity and very high cost. Recent year's development of small, inexpensive satellites known as pico and nano satellites has in a way changed this by considerably lowering both the price point of satellite construction and launch.

An interesting development along these lines has been the introduction of the CubeSat platform. To help universities worldwide perform space research the CubeSat platform was developed in 1999 aiming to provide practical, cost-effective and reliable launch opportunities for small satellites and their payloads through a standardized platform measuring from 10 x 10 x 10 cm to 10 x 10 x 30 cm [9].

By investigating what is possible to accomplished with low cost components in terms of reliability, two important things can be achieved. The cost of using space platforms for research can be reduced while maintaining the required level of reliability and at the same time the number of non-functional satellites (i.e. problematic space junk) can be reduced.

As a part of the national student satellite program, the NTNU Test Satellite (NUTS) project was initiated in 2010. The project's goal is to launch a double CubeSat giving hands-on experience of satellite technology to students within different fields, such as electronics, communications, space technology, physics, cybernetics and computer science. This includes planning, specification, design, construction, launch and operation of a satellite.

1.4 NUTS design

The NUTS internal design depends on a bus structure. The main parts of the satellite bus are:

- Power system (solar cells and batteries)
- Attitude and orbit control
- Mechanical system (structures and mechanisms)
- TT&C (telemetry, tracking and command)
- Thermal system

- On-board data handling
- Payload and experiment

The NUTS CubeSat is using the AVR32 UC3 MCU as main processor. There are two processors on board, one for the main computer and one for the communications system. The system is running the FreeRTOS operating system.

The next figure shows an overview of the satellite modules and their connections to the backplane.

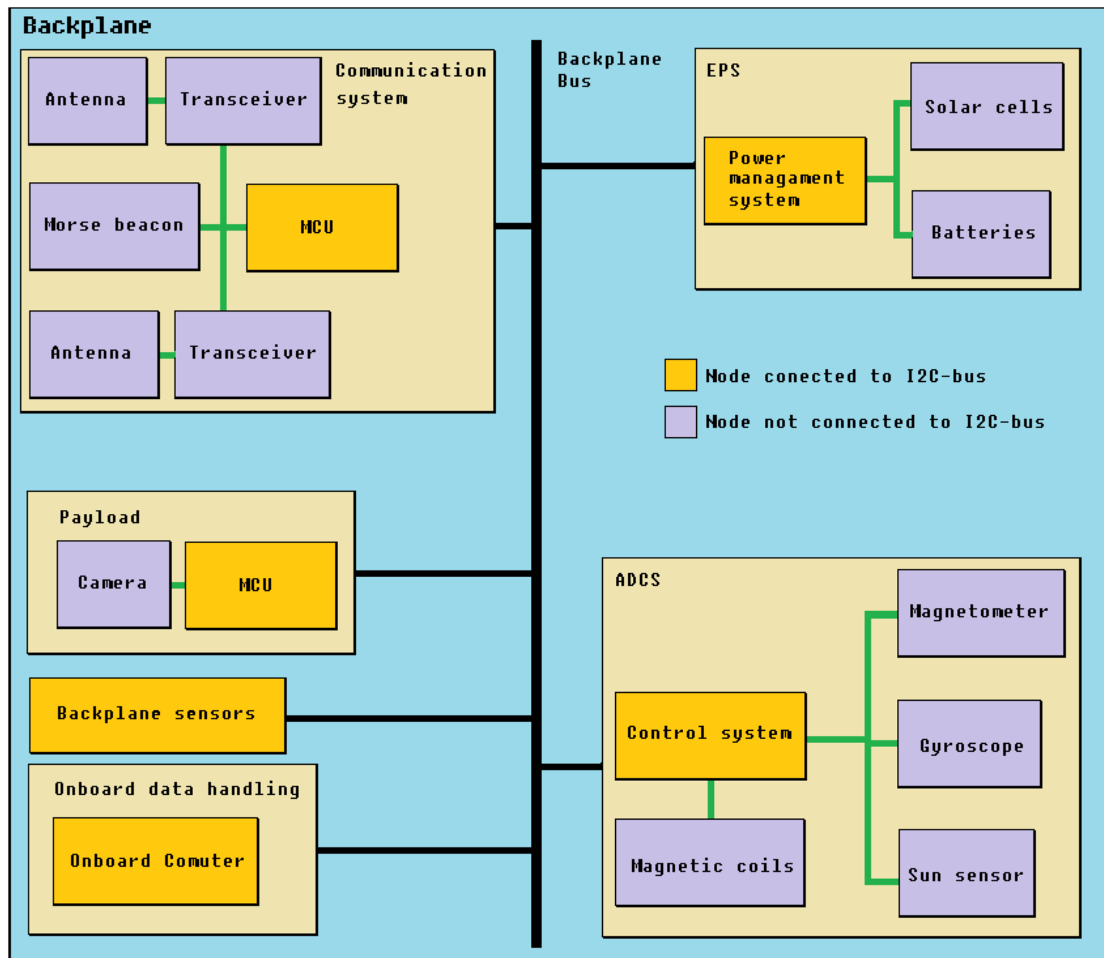


Figure 1: Abstract overview of the satellite modules and their connections

2 Memory management

The C programming language gives the programmer control over dynamic memory allocation. Reckless use of this control can lead to memory management problems, which cause performance degradation, unpredictable execution or crashes.

Some of the problems that cause memory leaks are writing or reading beyond an allocated memory segment or trying to free memory that has already been freed. A memory leak occurs when memory is allocated and not freed after use, or when the pointer to a memory allocation is deleted, rendering the memory no longer usable. Memory leaks degrade performance and, over time, cause a program to run out of memory and crash. Access errors lead to data corruption, which causes a program to behave incorrectly or crash [2].

In the next section we will have an overview of memory management in FreeRTOS comparing different available implementations. After that we will discuss some methods to detect memory leaks.

2.1 Memory management schemes in FreeRTOS

A real time operating system kernel has to allocate RAM dynamically each time a task, queue, or semaphore is created. The use of the standard `malloc()` and `free()` library functions can by accompanied with some undesirable side effects for one or more of the following reasons [1]:

- Their implementation can be relatively large, taking up valuable code space.
- They are rarely thread-safe.
- They are not deterministic; the amount of time taken to execute the functions will differ from call to call.
- They can suffer from memory fragmentation.
- They can complicate the linker configuration.

To avoid some of these problems, an alternative memory allocation implementation is required. FreeRTOS defines two new functions to do that. Memory allocation can be done by calling `pvPortMalloc()` instead of calling `malloc()` directly. Also when RAM is being freed, instead of calling `free()` directly, the function `vPortFree()` is used. `pvPortMalloc()` has the same prototype as `malloc()`, and `vPortFree()` has the same prototype.

FreeRTOS contains five implementations of vPortFree() and pvPortMalloc(), each of them is designed to address different requirements depending on the system design.

Next, each of these implementations will be described briefly with focusing on the main differences and when each of them is most suitable to use [3].

Heap_1

This is the simplest implementation of all. It does not permit memory to be freed once it has been allocated. It is most suitable for systems where the application creates all the tasks, queues, semaphores, etc. when the system boots, and then uses all of these objects for the lifetime of program.

Heap_2

Unlike scheme 1, it allows previously allocated blocks to be freed and uses a first fit algorithm to allocate new blocks. However, it does not combine adjacent free blocks into a single large block. This scheme is suitable when the size of allocated blocks is always the same. Otherwise the available free memory might become fragmented into many small blocks, eventually resulting in allocation failures. This type is also not deterministic - but is much more efficient that most standard C library malloc implementations.

Heap_3

This implements a simple wrapper for the standard C library malloc() and free() functions. The wrapper simply makes the malloc() and free() functions thread safe. This scheme is also not deterministic and will probably considerably increase the RTOS kernel code size.

Heap_4

This scheme uses a first fit algorithm and, unlike scheme 2, does combine adjacent free memory blocks into a single large block (it does include a coalescence algorithm). This implementation can be used even when the application repeatedly deletes tasks, queues, semaphores, mutexes, etc. and when it allocates variable block sizes.

Therefore, this scheme will most likely result in a less fragmented heap space compared to heap_2, even when frequent operations of allocation/deallocation are required.

Although this scheme is not deterministic, it is much more efficient than most standard C library malloc implementations.

Heap_5

This scheme uses the same first fit and memory coalescence algorithms as heap_4, and allows the heap to span multiple non adjacent (non-contiguous) memory regions.

The next table shows a summary of each implementation features:

	Deterministic	**Code size**	**Segmentation**
heap_1	Yes	Small	No
heap_2	No	Small	High
heap_3	No	Large	Target dependent
heap_4	No	Moderate	Moderate
heap_5	No	Moderate	Low

2.2 Suggesting a memory management scheme

Generally speaking, it is not recommended to use dynamic memory in embedded systems unless there is a very good reason for that. So in most cases the heap_1 implementation would be favored.

If that's not the case then we are force to consider the other 4 options. Heap_1 plus a `vPortFree()` function will give us heap_2. Anyway, heap_2 is not that attractive to use since it doesn't support combining adjacent memory blocks even if they are free, leading to these blocks being eventually useless, or even worse, leading to allocation failure.

Heap_3 will give us the traditional C library functions which we avoided in the first place due to the previously mentioned reasons.

Therefore, from my point of view, an appropriate choice for our application would be heap_4 since it provides a good compromise between the required overhead and segmentation level. Heap_5 might be desired in some cases for its additional convenience, but the additional code size and operation overhead should be evaluated against the actual usability.

Also we should keep in mind that our choice of a memory management scheme will determine what options we would have when choosing a method for stack overflow detection. More on this topic in the stack overflow section.

2.3 Detecting memory leaks

One of the problems with developing embedded systems is the detection of memory leaks. Some tools exist for detecting memory leaks, but most of them are either Linux specific or targeted to a certain compiler.

In [4], a couple of methods for detecting memory leaks are discussed. We will review these methods briefly and then select one of them and try to integrate it in one of FreeRTOS implementations of `pvPortMalloc()` and `vPortFree()`.

What makes memory leaks particularly harder to detect than other kind of bugs is that the misbehavior they cause may occur far from where the bug was caused. As a result, these problems are difficult to resolve by stepping through the code with a debugger. Therefore, code inspections sometimes catch these problems more quickly than any technical solution. Adding debug code to generate output is often a better alternative than a source code debugger, but it can change the shape of the memory layout. Another drawback is that if the debugging code consumes memory, we may run out of RAM sooner in the debug version than we would in the production version. This is specifically important in microcontroller based systems where a very little amount of RAM is available. Still, the leak should remain detectable regardless of these side effects of the debug code.

For simplicity, simple wrappers of `malloc()` and `free()` functions will be used in the coming example codes (not the actual function defined in `heap.h`).

2.3.1 Measuring memory usage

The first thing we might think of to measure memory usage is to simply add up how much is allocated and subtract any memory freed. For `malloc()`, this is trivial. We can simply have a static value and track allocations using the following code:

```
void *pvPortMalloc(size_t size)
{
   size_inUse += size;
   return malloc(size);
}
```

Deallocating is a bit trickier, since `free()` is not passed a size. Instead, the `free()` function is passed a pointer to the block. Usually the size is hidden in a header just before the block pointed to, so something like this might work:

```
void vPortFree(void *p)
{
   size_t *sizePtr=((size_t *) p)-1;
```

```
   size_inUse -= *sizePtr;
   free(p);
}
```

This would not be portable since another `free()` implementation might not use this convention, or may store it at a slightly different offset.

A better option can be using an array to keep a record of each block of allocated memory. When the block is freed, we remove that record from our array. Ideally, we would sort these blocks by type, but calls to `malloc()` and `free()` do not contain any type information. The best indication available is the size of the allocations. Therefore, the array `counter` holds how many block we have of a certain size. Also, the array `blocks` is used to hold pointers to the allocated blocks.

```
void *pvPortMalloc(size_t size)
{
   void *newAllocation = malloc(size);

   for (int i = 0; i < NUM_BLOCKS; i++)
      if (blocks[i].addr == 0)
      {
         // found empty entry; use it
         blocks[i].addr = newAllocation;
         blocks[i].size = size;
         incrementCountForSize(size);
         break;
      }
   FS_totalAllocated += size;
   return newAllocation;
}
```

`vPortFree()` can be written in a similar way:

```
void vPortFree(void *blockToFree)
{
   for (int i = 0; i < NUM_BLOCKS; i++)
      if (blocks[i].addr == blockToFree)
      {
         // found block being released
         decrementCountForSize(blocks[i].size);
         FS_totalAllocated -= blocks[i].size;
         blocks[i].addr = 0;
         blocks[i].size = 0;
         break;
      }
   free(blockToFree);
}
```

The proposed arrays are defined as:

```
typedef struct
{
   void * address;
   size_t size;
} BlockEntry

typedef struct
{
   int count;
   size_t size;
} Counter;
```

```
static BlockEntry blocks[NUM_BLOCKS];
static Counter counters[NUM_SIZES];
```

It is also worth mentioning that `NUM_SIZES` and `NUM_BLOCKS` should be chosen to be large enough to cope with the number of allocations in the system, but not so large that we use up all of the RAM before we begin.

2.3.2 Analyzing memory usage

A simple way to analyze memory usage over time to set up a simple function or task that is called periodically to print (or save to a file) the number of allocated blocks of each size. The output may be similar to this:

End of Iteration n		End of Iteration n+1		…
Size	Allocated blocks	Size	Allocated blocks	
20	1	20	1	
24	5	24	10	
44	5	44	6	

By monitoring the printed information over time we can detect if the number of blocks of a certain size is growing in an abnormal way. However, having a lot of object with similar sizes will make it more difficult to pin point the leak source.

2.3.3 Tracking where memory is allocated

To have a better chance hunting a suspicious memory leak source, we can collect information about where memory is allocated in a program. If we used a macro definition we can pick up this information without changing our function signature:

```
#define pvPortMalloc(size_t size) pvPortMallocLine(size, __LINE__, __FILE__)
```

Now the `__LINE__` and `__FILE__` variables will be automatically passed to our function. What's left now is to make some small changes in our allocation function:

```
void *pvPortMallocLine(size_t size , int line, char *file)
{
   void *newAllocation = malloc(size);

   for (int i = 0; i < NUM_BLOCKS; i++)
      if (blocks[i].addr == 0)
      {
         // found empty entry; use it
         blocks[i].addr = newAllocation;
         blocks[i].size = size;
```

```
         blocks[i].line = line;
         blocks[i].file = file;
         incrementCountForSize(size);
         break;
      }
   FS_totalAllocated += size;
   return newAllocation;
}
```

Also our `struct` definition will need to be changed a bit:

```
typedef struct
{
   void * addr;
   size_t size;
   int line;
   char * file;
} BlockEntry;
```

The `__FILE__` string has a non-trivial overhead, so if we don't have a lot of files in our project it's recommended to depend only on the line number.

3 Stack overflow

In an RTOS, there is a separate stack for each task, and each task might have drastically different stack size needs. Making things even more challenging, stack overflows often affect a somewhat unrelated memory area – global variables, allocated memory, or another task's stack – and thus the subsequent problem does not manifest itself until much later than when the overflow occurred making it more difficult to detect.

FreeRTOS provides the user with the option to check for overflow situations manually using an API, or at run time using an OS functionality. The option used is configured using the `configCHECK_FOR_STACK_OVERFLOW` configuration constant.

3.1 Manual checking

The first option is to do the detection manually. To select this method we set `configCHECK_FOR_STACK_OVERFLOW` to 0. The manual checking can be done by using the API function:

```
uxTaskGetStackHighWaterMark()
```

This function returns the amount of remaining stack space that has been available since the task started executing. The closer the high water mark is to zero, the closer the task has come to overflowing its stack.

3.2 Runtime stack checking

FreeRTOS provides two mechanisms that can be used to assist in the detection and correction of a stack overflow situation. It should be taken into consideration that these options are only available on architectures where the memory map is not segmented. This should be put in mind when choosing one of the memory management schemes discussed earlier.

When a stack overflow is detected, the stack overflow hook function is called. The user should provide the hook function which has the following prototype:

```
void vApplicationStackOverflowHook( TaskHandle_t xTask,
                                    signed char *pcTaskName );
```

The `xTask` and `pcTaskName` parameters pass to the hook function the handle and name of the offending task respectively.

Method 1

This method is selected when `configCHECK_FOR_STACK_OVERFLOW` is set to 1. At every context switching, the kernel checks that the stack pointer remains within the valid stack space after the context has been saved. If it's found outside the valid range the stack overflow hook is called. This method is quick to execute but can miss stack overflows that occur between context saves.

Method 2

This method is selected when `configCHECK_FOR_STACK_OVERFLOW` is set to 2. In addition to what's done in method 1 extra checks are performed. When a task is created, its stack is filled with a known pattern. Method 2 tests the last valid 16 bytes of the task stack space to verify that this pattern has not been overwritten. The stack overflow hook function is called if any of the 16 bytes have changed from their expected values.

Method 2 is a bit slower than method 1 but it will most likely catch all stack overflows.

3.3 Suggesting a checking method

The decision here strongly depends on our choice for the memory management scheme. If we have chosen heap_1 then we are free to pick whatever method we want. A good suggestion would be to pick a runtime checking method at first, and if we can afford paying some extra overhead (especially if this is only for debug phase) that would be method 2. And since this method checks the last 16 bytes of the stack, it can give us some insight whether an overflow is soon to happen. After that, if an overflow is detected, we can give the task some extra stack size or make some structural changes in the task. In the latter case the manual watermark check can be useful to determine which events or cases the overflow is most likely related to, and therefore having a better idea what changes can improve the behavior.

In the case when a memory management scheme other than heap_1 is chosen, there is no much place for a choice and we are tied to the manual method.

The manual method is generally a good practice when we suspect that some task will consume a lot of stack size in its life time, or when we are short on memory in our system and want a better estimation of our needs.

4 Lock-free circular buffer

4.1 Introduction

Some functionalities for the NUTS satellite require the usage of a circular buffer. A circular buffer is basically a queue that works as if it is connected end-to-end eliminating the need to shift its elements when it's consumed. Most of the circular buffers implementations out there are mainly dependent on semaphores to provide thread safety for the cases where multiple producers or multiple consumers exist.

The usage of semaphores based blocking to provide thread safety comes with a set of undesired hazards such as dead locks and priority inversion. Therefore, several circular buffer implementations were explored and eventually a lock-free implementation that doesn't depend on semaphores was chosen and ported to C with some modifications to be usable on our microcontroller based system.

4.2 How does it work

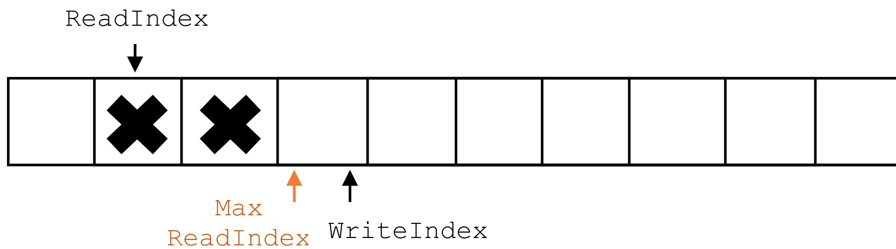
The queue is based on an array and 3 different indexes:

- `writeIndex`: Where a new element will be inserted to.
- `readIndex`: Where the next element where be extracted from.
- `maximumReadIndex`: It points to the place where the latest "committed" data has been inserted. If it's not the same as `writeIndex` it means there are writes pending to be "committed" to the queue, that means that the place for the data was reserved (the index in the array) but the data is still not in the queue, so the thread trying to read will have to wait for those other threads to save the data into the queue.

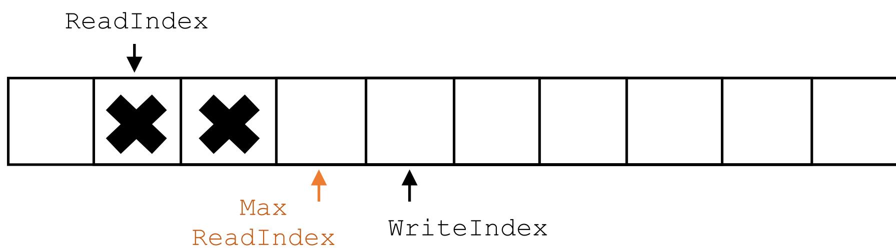
It is worth mentioning that 3 different indexes are required because the queue allows to set up as many producers and consumers as needed. In other words, this structure is what allows a multi-threaded behavior. This will be explained in more details later on.

4.2.1 Inserting elements into the queue

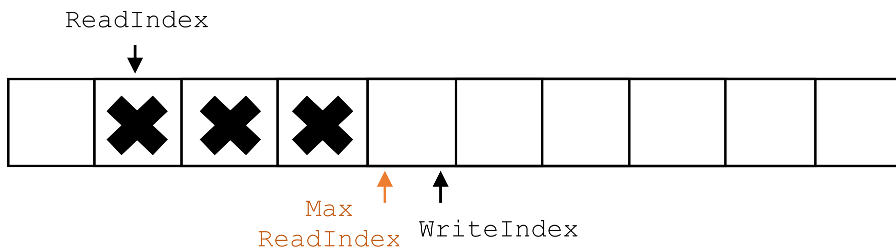
The following image describes the initial configuration of the queue. Each square describes a position in the queue. If it is marked with a big X, it contains some data. Blank squares are empty. In this particular case there are 2 elements currently inserted into the queue. `WriteIndex` points to the place where new data would be inserted. `ReadIndex` points to the slot which would be emptied in the next call to `pop()`.



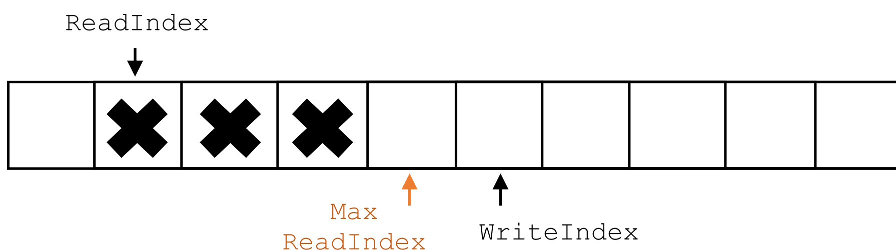
Basically when a new element is going to be written into the queue using the `push()` function, the producer thread "reserves" the space in the queue incrementing `WriteIndex`. `MaximumReadIndex` points to the last slot that contains valid (committed) data.



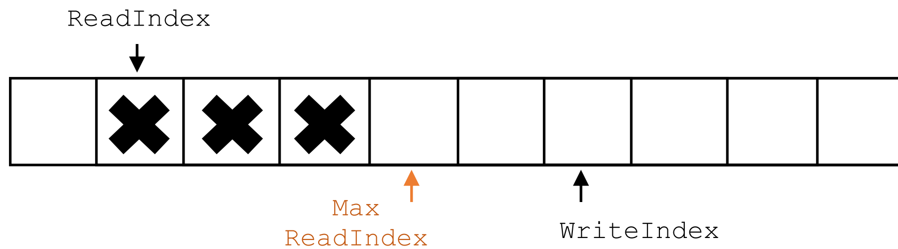
Once the new space is reserved, the current thread can take its time to copy the data into the queue. It then increments the `MaximumReadIndex`.



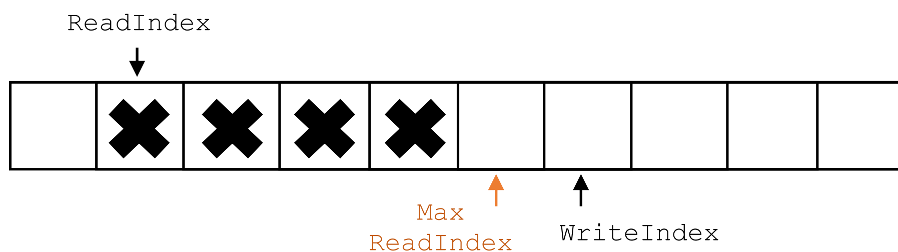
Now there are 3 elements fully inserted in the queue. In the next step another task tries to insert a new element into the queue



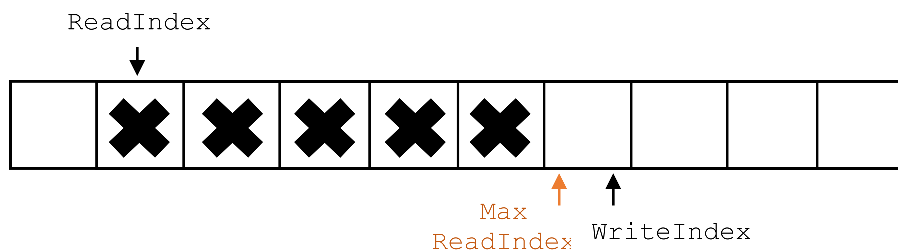
It has already reserved the space its data will occupy, but before this task can copy the new piece of data into the reserved slot a different thread reserves a new slot. There are 2 tasks inserting elements at the same time.



Threads now copy their data in the slot they reserved, but it must be done in a strict order: The 1st producer thread will increment `MaximumReadIndex`, followed by 2nd producer thread. The strict order constraint is important because we must ensure data saved in a slot is fully committed before allowing a consumer thread to pop it from the queue.



The first thread committed data into the slot. Now the 2nd thread is allowed to increment `MaximumReadIndex` as well.



The second thread also incremented `MaximumReadIndex`. Now there are 5 elements into the queue.

4.2.2 Removing elements from the queue

I will not explain the removing process in details. In some sense it is similar to the insertion process except the `ReadIndex` is the pointer that's moving in this case. The `pop()` function will return a failure when the `ReadIndex` and the `MaximumReadIndex` refer to the same place.

5 Exception handling

5.1 The usual way

As it's commonly known, C doesn't directly support exception handling. So how do we know that a certain function has failed? Let's take the following example function that performs popping a value from a stack.

```
int stack_pop(int_stack * stack);
```

The typical approach to express operation failure would be making the function return an error code. That seems like a neat way to do detect an error. But then `stack_pop` would need another argument in order to return the item and the function prototype would look like this:

```
enum stack_status stack_pop(int_stack * stack, int * item);
```

Comparing the two function prototypes we directly notice that the use of the later form repeatedly makes our code less readable and thus, most likely, less maintainable.

The basic function of exception handling is to transfer control to an exception-handler when an error occurs, where the handler resides somewhere higher up in the current function call hierarchy. [6] discusses the use of `setjmp()` and `longjmp()` to accomplish simple exception handling in C. Let's take a look at the following code snippet:

```
jmp_buf jumper;

int function(int a, int b)
{
   if (b == 0) // can't divide by 0
      longjmp(jumper, -3);
   return a / b;
}

void main(void)
{
   if (setjmp(jumper) == 0)
   {
      int Result = function(7, 0);
      // continue working with Result
   }
   else
      printf("an error occured\n");
}
```

After the `main()` function starts executing and `setjmp()` will return 0 and `function()` will be called. Let's suppose now that b equals to 0. In this case `longjmp()` will be executed causing the program to jump back to where `setjmp()` was called and returning this time '-3' indicating an error and printing the error

message. As we notice the division was never executed therefore fulfilling our purpose.

We may seem to have solved the problem. However, this example is a little too simple. It relies on a single global variable called "jumper," which contains the information where the exception handler is. However, we need many different exception handlers in different functions, so how will `function()` know which jumper to use? Another issue is multitasking. Each task will have its own call hierarchy and thus needs its own jumper or set of jumpers.

5.2 Using an exception handling framework

The framework proposed in [5] builds on the previous concept creating a dynamically linked list of exception handler records. Such a record will contain a `jmp_buf` structure and some supplemental information. Each function that defines an exception-handler adds such a record to the list and removes it from the list when it returns. These details are hidden from the user in a library called "`e4c.h`" and the function calls are formed as macros with the names `try`, `catch` *and* `throw` which are familiar to users who have dealt with other exception supporting languages such as C++.

Let's get back to the `stack_pop()` function and rewrite it to throw and exception in case of an empty stack:

```
int stack_pop(int_stack * stack){

    if( stack_is_empty(stack) ) throw(StackUnderflowException, NULL);

    ...
}
```

Now we write the `main()` to use the aforementioned library. First we need to make sure that we call the stack functions within an exception context that we establish by calling `e4c_using_context(E4C_TRUE){}`, and each exception related operation after that should be within the body of this context. Here is an example of the usage:

```
#include <stdio.h>
#include "e4c.h"
#include "stack.h"

int main(int argc, char * argv[]){

e4c_using_context(E4C_TRUE){

    int_stack stack;
    stack_construct(&stack, 3);
    int x;

    try{
        /* this will throw an exception since the stack is empty */
        stack_pop(&stack);
    }
```

```
    catch(StackException){
        printf("Error: %s\n", e4c_get_exception()->message);
    }
    finally{
        stack_destruct(&stack);
    }
}
}
```

Another way to set up the exception context is by calling `e4c_context_begin()` when the context starts, and `e4c_context_end()` when it ends. This may be more suitable when the context cover a large chunk of code.

5.3 Documenting exceptions

It is a good practice to document which exceptions a function can throw (and under what circumstances), so the caller can get ready to handle them. This can be done in the same way we document the parameters and return value of a certain function. This can be an example of good documentation:

```
/**
 * Returns (and removes) the item at the top of the stack.
 *
 * @param stack The stack to pop from
 * @return the topmost item
 *
 * @throws NullPointerException if stack is NULL
 * @throws StackUnderflowException if stack is empty
 */
int stack_pop(int_stack * stack);
```

5.4 Summary

Using exceptions allows us to design more elegant interfaces. Our functions don't need to return values just for the sake of error handling.

What has been done in this project (regarding exception handling) is reviewing different approaches to handle errors in the C language. Also several exception handling frameworks has be evaluated, e.g. the one used in OnTime ROTS, although they haven’t been mentioned in this report. Exceptions4C framework was chosen because it is well documented, easy to use and has a nice exceptions hierarchy.

5.5 Further work

Although the discussed framework works with FreeRTOS without problems when tested in a small project, it may cause further issues in the long run. One of the nice features of this framework is the support of multi-threading (can be activated using a certain macro). Anyway, the thread-safe implementation relies on `pthreads`, the POSIX API for writing multithreaded application. Therefore, for an optimum compatibility, this implementation should be ported to the FreeRTOS API.

I have actually started to work on porting it but giving the complexity of the task and the limited time available I couldn't fulfill anything major in this respect. So a possible future work can be porting the implementation from POSIX to FreeROTS API.

6 Using RAID 4 techniques for error detection in embedded systems

In this section I will describe a software technique for detecting soft errors occurring in digital systems. Before going into details of this technique we will take a look at one of the alternative methods used for error detection which is complete operational redundancy.

6.1 Detection using redundancy

[7] explains in details a number of software techniques for fault tolerance based mainly on redundancy. We will briefly review the methods used for detecting errors affecting data and conditional instructions. The idea is to define the interdependence relationships between the variables of the program and to classify them in two categories according to their role in the program: intermediary variables and final variables.

Once the variables relationships are drawn up, all the variables in the program are duplicated. For each operation carried out on an original variable, the operation is repeated for its replica.

The next figure shows an example of program transformation:

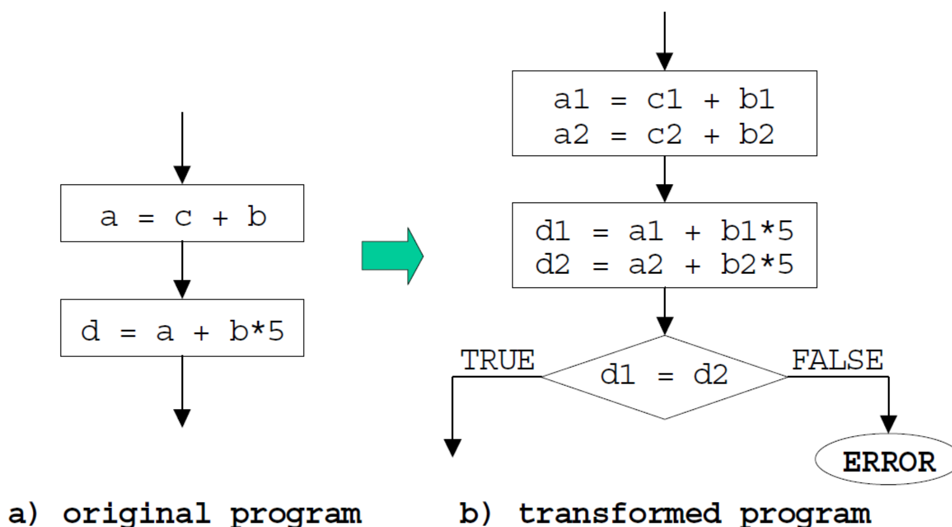


Figure 2: Transformation rule targeting errors affecting data

After each write operation on the final variables a consistency check between the values of the two variables (original and duplicated) is introduced. An error is signaled

if there is a difference between the value of the original variable and the one of the duplicated variable.

In addition to the errors affecting data, an error can affect a control instruction such as conditional instructions. In this case, extra checks should be added especially when this condition can cause a critical change in the execution flow. This can be illustrated in the following figure:

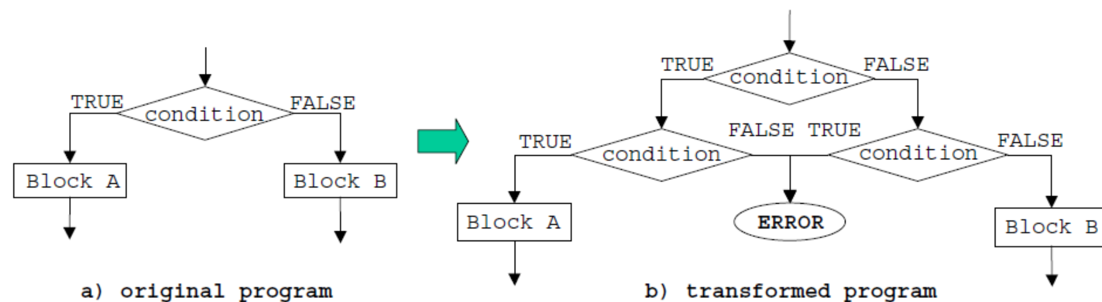


Figure 3: Transformation targeting errors affecting control instructions

6.2 A more manageable way

The previous method indeed achieves a very good error detection rate. However, the resulting code is often confusing, hard to read and difficult to maintain. This may ultimately have negative consequences, increasing the potential of human errors. For example, in a program that contains thousands of lines it is not very unlikely that a programmer mistakenly replaces a2 with a1. Depending on its place, this bug may not have direct consequences, causing a failure much later in the execution cycle. We can just imagine how painful would it be to look for this tiny mistake in a huge source code.

The method I propose is fairly simple and it's not a break through by any means. In the hard disks world, RAID4 technique is a very successful method for error detection and correction using the simple concept of parity. This, combined with the exception handling method discussed in the previous chapter, can give us a very elegant and effective error handling mechanism.

I have chosen to do my test implementation in C++ to make use of the operator overloading feature. I have also implemented a simple test bench with error injection function to simulate bit flipping situations. A new type has been defined, and called `intx`, to serve as the error safe interface between the programmer and the actual integer operation. An `intx` variable can be used exactly as a normal integer so the user can keep programing in the way that's comfortable to him without worrying about error checking. All that the user has to do is to put the critical operations inside a `try{}` block and catch any potential error.

```
intx arr[3];
try
{
	arr[0] = 0x0304;
	arr[1] = arr[0] * 13;
	error_injection();
	arr[2] = arr[1] * arr[0];
}
catch (const char* msg)
{
	errors_count++;
}
```

The `error_injection()` function will inject a single error in a random byte from a random element in the array. If an error is detected in any of the data that has been retrieved from the memory, an exception will directly be thrown and can be dealt with in the `catch{}` block.

Trying to implement the same example using the redundancy method and supposing all variable will be needed later we get the following code:

```
int arr1[3];
int arr2[3];
try
{
	arr1[0] = 0x0304;
	arr2[0] = 0x0304;
	arr1[1] = arr1[0] * 13;
	arr2[1] = arr2b[0] * 13;
	if (arr2[1] != arr1[1])
		throw "VariableFieldsMismatch";
	error_injection();
	arr1[2] = arr1[1] * arr1[0];
	arr2[2] = arr2[1] * arr2[0];
	if (arr2[2] != arr1[2])
		throw "VariableFieldsMismatch";
}
catch (const char* msg)
{
	errors_count++;
}
```

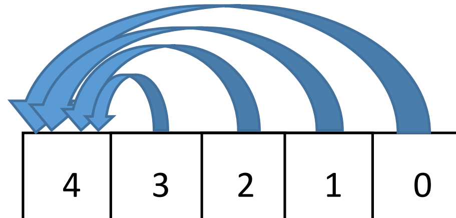
Instead of the 3 lines required in the first case 10 lines of code are needed in the second. This difference can be higher if a more complex example is used.

Under the hood, a union is used to represent the 4 bytes of an integer as 4+1 bytes to represent the parity. This is how the used union looks like:

```
union {
	uint32_t var;
	uint8_t c[5];
};
```

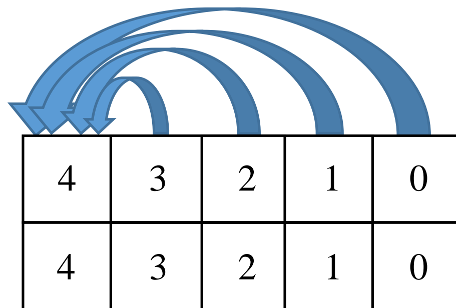
6.3 Operation

When we are just concerned with error detection, the procedure is quite simple. After an assignment the parity byte is set depending on the other 4 bytes:

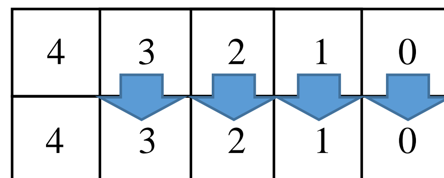


After that, whenever this variable is needed the parity is verified before doing any operation and an exception is thrown if something went wrong.

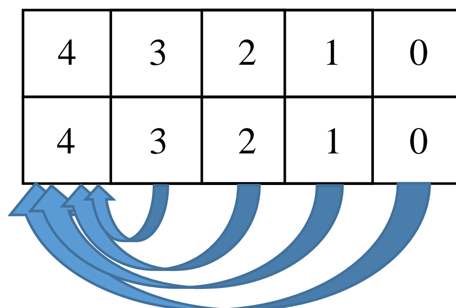
When it comes to error correction things get a bit tricky. For the sake of error correction another 5 bytes are added. They are declared using a union similar to the one shown earlier. Now suppose we have performed an assignment operation on some variable of the type `intx`. Initially, the first parity byte is calculated as before.



Then we copy the data bytes from the first union to the second:



Finally we calculate the parity for the second union:



Now, whenever this variable is needed the parity of the first union is verified before doing any operation. If the check gave an error, the parity of the second union is check.

If it was correct, we transfer the first 4 bytes to the first union. Otherwise, we throw an exception for variable fields mismatch.

The following flow graph shows this procedure:

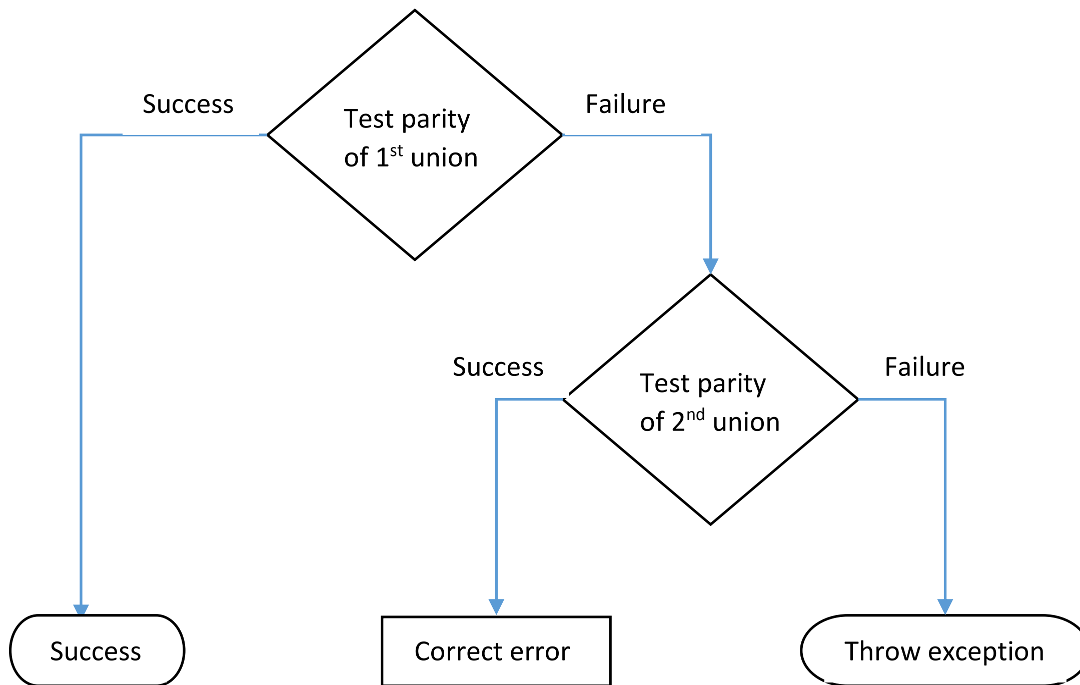


Figure 4: Error detection and correction procedure

6.4 Benchmarks

The proposed method was tested against the redundancy technique by measuring and comparing the execution time under different conditions. In terms of the number of detected errors, both methods were able to capture all the injected errors. Perhaps this is not very realistic because of the simplicity of the error injection method used, but this is at least an indication that both are fairly robust.

Regarding execution speed, the performance was very similar in an environment with many errors. The following chart shows a comparison of execution time in this test case for a number of runs:

Series1: Proposed method

Series2: Redundancy method

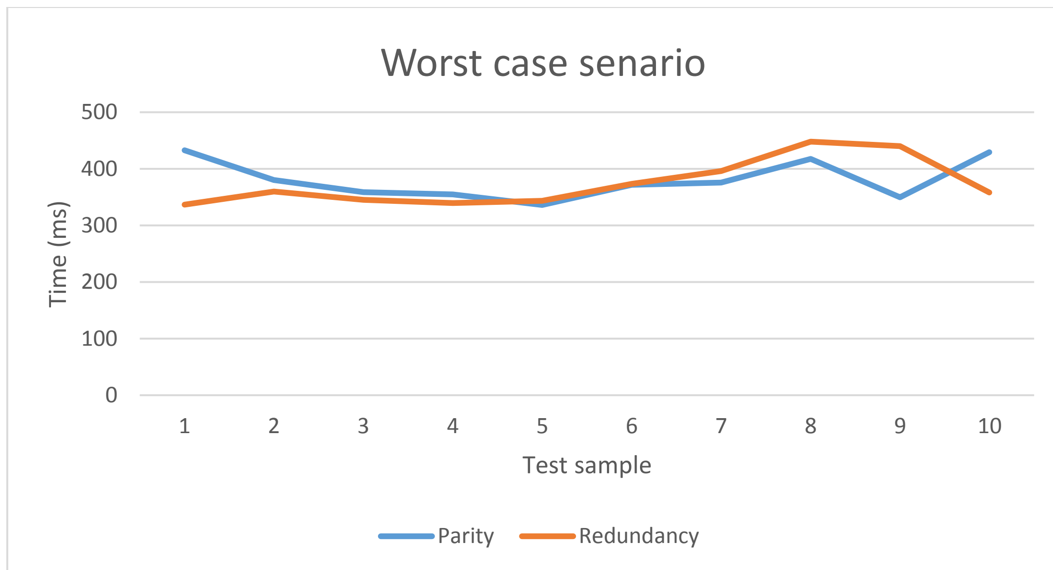


Figure 5: Error detection - worst case scenario

Unfortunately, in the best case where the environment produces no errors, the proposed method shows a remarkable drawback with a clear advantage of the redundancy method. The next chart shows the test case:

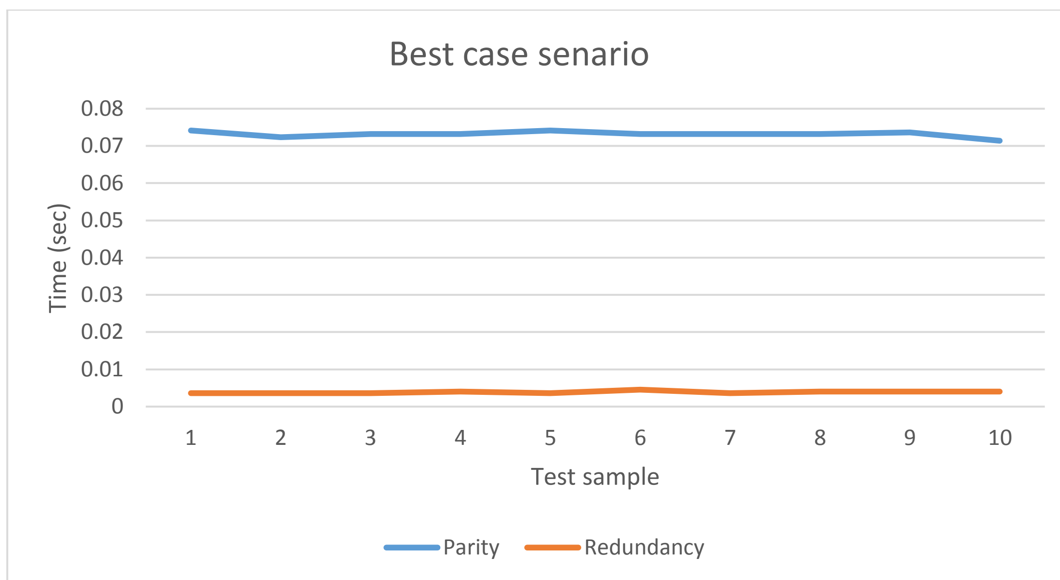


Figure 6: Error detection - best case scenario

Regardless of the relatively poor performance in the best case, we shouldn't directly consider this solution useless. In an error intensive environment, in my opinion, it will have preferable features such as programming convenience and increased software reuse and maintainability. Also we should forget about the error correction ability that the other method lacks. Even if we decided to perform error correction using redundancy, we will produce even larger code size with the multiple variables copies that will make the code almost impossible to manage.

7 Discussion and Conclusion

This report has discussed several topics, from some of operating systems matters such as memory management and stack overflow, to dealing with error handling, with an emphasis on error reporting. It also suggest a method for error detection and correction.

Most embedded systems try to avoid dynamic memory allocation and keep their memory management as simplistic as possible. And throughout this project we noticed that using the simplest memory management scheme won't just give assurance against some problems such as segmentation and memory leaks, but it will allow using more advanced techniques for stack overflow detection as well. That's why the simpler techniques are, in most cases, more preferable over the other schemes that may give us some more convenience in use. This is especially true for applications like CubeSates, where having less potential sources of failure is a priority.

Also, there has been a lot of emphasis in this report on the importance of an efficient error reporting mechanism. An efficient implementation of exception handling can reduce the amount of code necessary for error reporting and therefore making the code more manageable. Several exception handling frameworks exist. A feature rich framework was chosen and discussed in details.

The last chapter of the report discussed a mechanism for using some type overloading to introduce parity bytes to the common data types. These additional bytes can be used for error detection and correction without forcing the programmers to have some strict coding guidelines. Although this technique lacks some efficiency, there is a potential for improvement and further work can be done to enhance its features.

8 Future work

The field of error checking is very wide and there is a lot more to be done. The first potential for future work is the pointed mentioned earlier in chapter 5 to port the POSIX dependent multi-threaded implementation to FreeRTOS.

The second potential is continuing the work with the proposed error checking technique. It's true that it isn't the most sophisticated one out there, but I believe there is a potential for development. I still haven't fully tested the version with error correction, so this can be a good point to start from.

9 References:

[1] Using the FreeRTOS Real Time Kernel - A Practical Guide, Richard Barry

[2] Memory Leak Detection in Embedded Systems, Cal Erickson, Linux Journal, Issue #101

[3] FreeRTOS documentation, http://www.freertos.org

[4] How to Detect Memory Leaks, Niall Murphy, http://www.barrgroup.com/Embedded-Systems/How-To/Detecting-Memory-Leaks-C

[5] Exceptions4c - An exception handling framework for C (https://code.google.com/p/exceptions4c/)

[6] On Time RTOS documentation, http://www.on-time.com/

[7] Detecting soft errors by a purely software approach, B. Nicolescu, R. Velazco.

[8] Lock-free circular buffer implementation, Faustino Frechilla, CodeProject (Link)

[9] Specification, CubeSat Design, "Rev. 12," Cal Poly, August, 2009.

[10] Error Detection and Correction for Low-Cost Nano Satellites, Kjell Arne Ødegaard, Master thesis, NTNU

[11] Questionnaire on CubeSat Reliability, Jasper Bouwmeester, Martin Langer, TU Delft and TU Munich, http://tinyurl.com/njvuywz