Fault Tolerance in FreeRTOS with triple modular redudancy

Guido Miglierina and Sara Piazza

July 30, 2025

Contents

1 Project data

- Project supervisor: Davide Baroffio
- Group members:

Last and first name	Person code	Email address
Piazza Sara	10782998	sara.piazza@mail.polimi.it
Miglierina Guido	10785492	guido.miglierina@mail.polimi.it

- The project was developed jointly between group members. There is no clear distinction on the task subdivision because each design decision was taken together.
- https://github.com/Miglie/ESProject

2 Project description

The goal of our project was to implement triple modular redundancy of the tasks in FreeRTOS and compare their output with voting to improve the fault tolerance of the operating system (e.g. fault tolerance to bitflips). We were able to deepen our knowledge of the topics studied in the AOS course like OS kernel and Real Time operating systems.

2.1 Design and implementation

2.1.1 xTaskCreate_TMR(): Generating three instances of each task

The first step to implement a new version of the kernel with triple modular redundancy was to add a new function to create tasks. The new function is called *xTaskCreate_TMR()* and every time it is called generates three instances of the task instead of one. Those task are then inserted in the scheduler queues to be executed. We implemented *xTaskCreate_TMR()* in an "all or nothing" way: if there is not enough space to create all three tasks, no tasks are created at all. To enforce this behaviour we had to add a slightly more complex error handling mechanism than the original function. In *xTaskCreate_TMR()* when a group of tasks is created they are associated to a Group Control Block (*GCB*), a data structure that saves the data of a group of three tasks. To do so we changed the implementation of the *TCB*, adding a new field with the pointer to the *GCB* (it was also necessary to modify the static version of TCB present in FreeRTOS.h). The arguments passed to function are the same of the original *xTaskCreate()* (see FreeRTOS documentation).

```
BaseType_t xTaskCreate_TMR(TaskFunction_t pxTaskCode,
                           const char * const pcName,
                           const configSTACK_DEPTH_TYPE usStackDepth,
                           void * const pvParameters,
                           UBaseType_t uxPriority, TaskHandle_t * const pxCreatedTask)
                           { ... }
```

2.1.2 GCB: A data structure to handle three instances

Every group of three tasks has a Group Control Block (*GCB*). *GCB* is a data structure that is accessed directly from the *TCB* of each task. In the *GCB* are saved the pointers to the three outputs generated by the tasks in the group and the *TaskHandle_t()* of the first two task terminating their execution. This bookkeeping is necessary because when the first two tasks save their output, they are suspended and they are woken up by the third task. To do so it needs to have the *TaskHandle_t* of the first two tasks. The last field of the *GCB* is an integer counter that keeps trace of the number of instances that have terminated.

```
struct GCB{
            void * output[3];
            TaskHandle_t task[2];
            int counter;
        };

typedef struct GCB * GroupHandle;
```

2.1.3 What a user needs to do: taskTerminated(), compare(), commit()

To signal that a task has terminated a round of execution and to save its output, the kernel function *taskTerminated()* has to be called at the end of the task code.

```
BaseType_t taskTerminated(void * output, int deallocate_memory,
                          int(*compare)(void * result1, void * result2),
                          void(*commit)(void * result),
                          TickType_t xDelay)
```

```
                                { ... }
```

To handle the various data types employed by the user, the output (void * output) is passed to this function via a void pointer. *taskTerminated()* saves the output in the GCB. If the task calling *taskTerminated()* is the first or the second of the group, the task will then suspend itself. If instead it is the third instance, the voting system is called. A flag (*int deallocate_memory*) is passed to this function to signal to the kernel if the memory has to be deallocated or not. For example a struct that is no more useful can be safely deallocated, while the outputs of task producing integers should never be deallocated. Deallocation happens inside the *taskVoting()* function. The user has also to implement two functions called *commit()* and *compare()*, which are passed as arguments through function pointer.

```
int compare (void * first, void * second){
/*Returns
1 if the element pointed to by first is equal to the element pointed to by second,
0 otherwise*/
}

void commit (void * result){
//Apply the desired actions using the correct output
}
```

The user can choose a delay to pass as argument. The delay takes place when the voting is done. The kernel delays the task and then wakes up the whole group (The delay should never be 0 to avoid starvation of other groups).

2.1.4 taskVoting(): the voting system

The function *taskVoting()* is called when the three outputs have been saved in the *GCB*. This function compares the outputs with the function provided by the user, and if a majority is found, the corresponding output is passed to the user defined commit function to finalize the task.

```
BaseType_t taskVoting(GroupHandle handle,
                      int deallocate_memory,
                      int(*compare)(void * result1, void * result2),
                      void(*commit)(void * result))
                      { ... }
```

If all three outputs are different the value of the return type *BaseType_t* is set to *pdFail* and then the error is propagated to the user by *taskTerminated()*. If the flag *deallocate_memory* is 1 memory is then deallocated, otherwise not. When the majority test (and eventually the commit section) has been done the task is delayed by a user defined time. After that it wakes up the whole group which is then rescheduled.

2.1.5 Assumptions

The whole project has been developed and tested using the common configuration of the FreeRTOS.h file, with the exception of preemption being disabled.

3 Project outcomes

3.1 Concrete outcomes

3.1.1 Modified kernel

The artifact we have produced is a modified version of the FreeRTOS kernel, that is available on our repository. Specifically we extended tasks.c, task.h, FreeRTOS.h. This version provides an API to use FreeRTOS with Triple Modular Redundancy.

3.1.2 Testing

We have also produced a series of tests that can be found in the file TestCollection.c, to extensively test the newly added functions. To test the system for bitflips we have manually changed the values of some variables during the execution using GDB. The first few test are simple because they were developed in the early stages of the development of the project, while the last test tries to put everything together.

3.2 Learning outcomes

- Even if we had already experience with C, we greatly improved our coding skills and understanding of basic concepts like pointers.
- We learned how to debug with GDB and how to use it for testing.
- We have learned how microcontrollers work and how to program them.
- We looked for the first time at the code of a real time OS and to modify it we had to understand deeply its functioning.

3.3 Existing knowledge

During the AOS course we learned the basic structure of an operating system. We found particularly useful the knowledge about tasks, task control blocks and how they are implemented in a OS. We also followed the Embedded Systems course, where we learned how fault tolerance is crucial, especially in real time applications. It would have been easier to approach this project if we had seen how FreeRTOS works during the AOS course.

3.4 Problems encountered

- Deciding which code had to be implemented in the kernel and which was acceptable to be left to the user.
- Managing multiple groups of tasks.
- Handling the diversity in user-defined data types and commit actions.
- Difficult first approach in understanding FreeRTOS kernel.

4 Honor Pledge

We pledge that this work was fully and wholly completed within the criteria established for academic integrity by Politecnico di Milano (Code of Ethics and Conduct) and represents our original production, unless otherwise cited.

We also understand that this project, if successfully graded, will fulfill part B requirement of the Advanced Operating System course and that it will be considered valid up until the AOS exam of Sept. 2025.

Guido Miglierina and Sara Piazza