Politecnico di Torino

Operating Systems

Fault injection on STM32f303

Final Project Report

Master degree in Electronics Engineering

Prof. Maurizio Rebaudengo
Masoud Hemmatpour

Casalino	Andrea	242094
Gualco	Andrea	240039
Inglese	Pietro	243243

October 31, 2017

Contents

Abstract

The purpose of the project is to inject different faults into the scheduler of the STMicroelectronics™ STM32f303 board running FreeRTOS™ and to study the Real Time Operating System behaviour after the injection.

CHAPTER 1

Introduction

The board that has been used is the STMicroelectronics^{TM} STM32F3DISCOVERY, running the Real Time Operating System FreeRTOS^{TM}.

1.1 STM32f303 Discovery

The STMicroelectronics^{TM} STM32F3DISCOVERY is a discovery kit designed to start developing very quickly.

The key features are:

- STM32F303VCT6 microcontroller featuring 256-Kbyte Flash memory, 48-Kbyte RAM in an LQFP100 package;
- On-board ST-LINK/V2 for PCB version A or B or ST-LINK/V2-B for PCB version C and newer;
- USB ST-LINK functions (Debug port, Virtual COM port with ST-LINK/V2-B only, Mass storage with ST-LINK/2-B only);
- Board power supply: through USB bus or from an external 3 V or 5 V supply voltage;
- External application power supply: 3 V and 5 V;
- L3GD20, ST MEMS motion sensor, 3-axis digital output gyroscope;
- LSM303DLHC, ST MEMS system-in-package featuring a 3D digital linear acceleration sensor and a 3D digital magnetic sensor;
- Ten LEDs (LD1 (red) for 3.3 V power on, LD2 (red/green) for USB communication, Eight user LEDs: LD3/10 (red), LD4/9 (blue), LD5/8 (orange) and LD6/7 (green));
- Two push-buttons (user and reset);
- USB USER with Mini-B connector;
- Extension header for all LQFP100 I/Os for quick connection to prototype board and easy probing;
- Comprehensive free software including a variety of examples, part of STM32CubeF3 package or STSW-STM32118 for legacy Standard Library usage.

The features that have been directly exploited in the project are the gyroscope, the 8 user LEDs, a page of the flash memory (in order to store informations), the RAM (in order to inject the faults) and the push buttons (especially the user one, used to switch from tasks to do).

1.2 FreeRTOS™

Over the board has been made run an Open Source Real time Operating System, FreeRTOS™. This is a very light operating system, mainly used for embedded applications. The core of the kernel is contained in only 3 C files:

- tasks.c
- queue.c
- list.c

The kernel part that is the most interesting wrt the scheduler is the one contained into the file *tasks.c*

Task

An application running on a RTOS can be structured as a set of different independent tasks. Each of them is executed within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. The processor of the STM32f303 can have only one task running at a time (it is single core) and the real time RTOS scheduler is responsible for deciding which task this should be. The RTOS scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes. As a task has no knowledge of the RTOS scheduler activity it is the responsibility of the real time RTOS scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is exactly that as when the same task was swapped out.
To achieve this, each task is provided with its own stack. When the task is swapped out the execution context is saved to the stack of that task so it can also be exactly restored when the same task is later swapped back in.

Task states

A task, once created, can have different states

- **Running**
  When a task is actually executing it is said to be in the Running state. It is currently utilising the processor. Since the STM32f303 processor (ARM® Cortex®-M4) is single core, there can only be one task in the Running state at any given time.
- **Ready**
  Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.
- **Blocked**
  A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls `vTaskDelay()` it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event. Tasks in the Blocked state normally have a “timeout” period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred. Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

- **Suspended**
  Similarly to the tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the `vTaskSuspend()` and `xTaskResume()` API calls respectively.

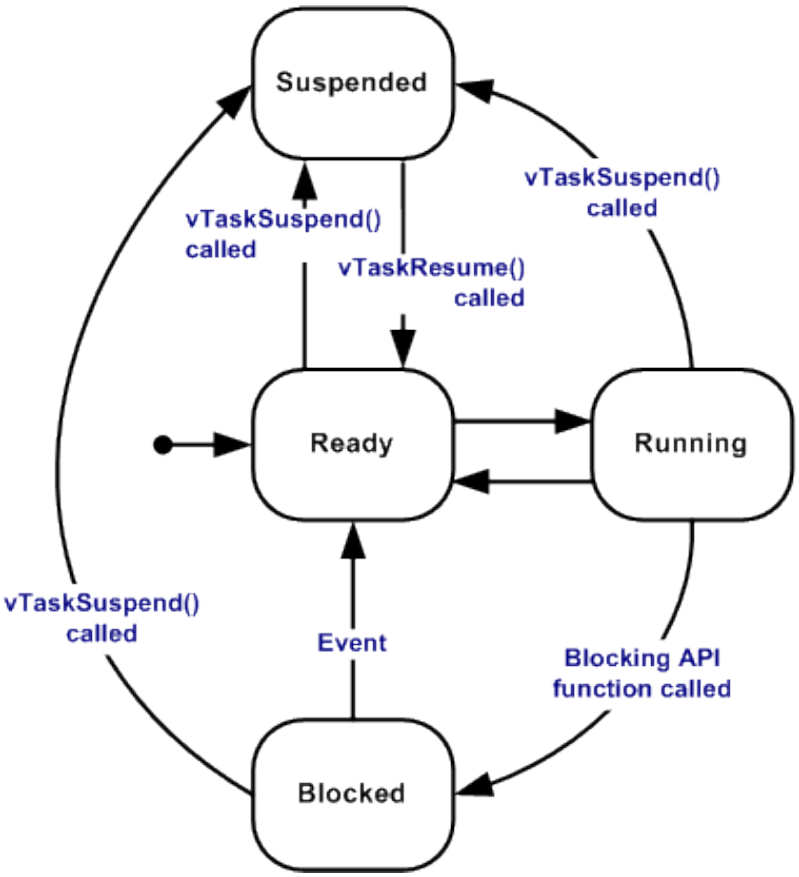

Figure 1.1: Task state transitions

Task priorities

Each task is assigned a priority from 0 to (`configMAX_PRIORITIES` - 1), where `configMAX_PRIORITIES` is defined within *FreeRTOSConfig.h*. In the project the number of priorities is 7.
Low priority numbers denote low priority tasks. The idle task has priority zero (`tskIDLE_PRIORITY`). The FreeRTOS scheduler ensures that tasks in the Ready or Running state will always be given processor (CPU) time in preference to tasks of a lower priority that are also in the ready state. In other words, the task placed into the Running state is always the highest priority task that is able to run.
Any number of tasks can share the same priority. If `configUSE_TIME_SLICING` is not defined, or if `configUSE_TIME_SLICING` is set to 1, then Ready state tasks of equal priority will share the available processing time using a time sliced round robin scheduling scheme.

CHAPTER 2

Demo Code

In order to have the fault injection and to use some of the features of the board, it has been written a "Demo code", containing different tasks with different purposes and assigning (not always) different priorities.

The Demo is written to have a directly visible effect of the fault and a written one, to check what happened. For this reason have been used both LEDs and writes on flash memory.

2.1 Task assignments

The program is divided in 5 different tasks:

- **Main Task**
  The purpose of the Main Task is to check the number of times the user button has been pressed and therefore kill or create the other tasks.
- **Task1**
  The purpose of Task1 is to activate the gyroscope, switching on the LED corresponding to the grade of the board. The grade is also saved in the flash memory, in a specific page.
- **Task2**
  The Task2 is always running alternating with Task1 and has the purpose to store in the flash memory a certain number (obviously different by the numbers saved in the Task1). It is used to see how the different priorities are handled by the scheduler.
- **Task3**
  The Task3 is the so called "Idle Task": it simply switches on all the LEDs and then kills himself. It is used to wait for the user input. It starts at the beginning of the demo and it is called again after Task1 and Task2.
- **Task4**
  The Task4 is in charge of reading the flash memory that has been written by Task1 and Task2 and it switches on the LEDs corresponding to the number that was written before.

In figure 2.1 can be seen the task execution order. The program starts with Task3 and Main Task with user button pressures equal to 0. Then, at the first button pressure, the Main Task creates both Task1 and Task2 and, in order to save in the flash memory their actions, it resets the memory page used to store data. At the end of the writing the user LEDs will be in idle state waiting for the next user input. In fact, the next button pressure will make the Main task kill Task1 and Task2 and create the Idle Task. The third button pressure will make the Main Task create Task4. The 4th button

pressure will make the Main Task kill Task4 and create Task3 again. This is very similar to the action of button = 0. In fact when the counter of button pressed reaches 4, it is set again to 1 in order to loop the normal execution as many times is needed.

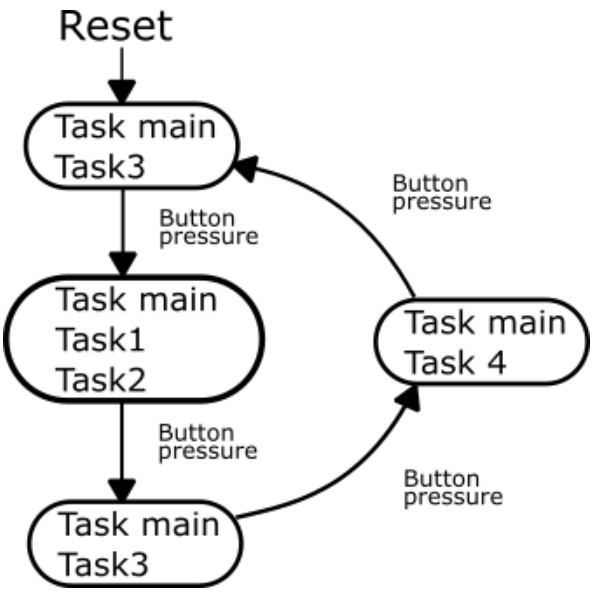

Figure 2.1: Task Execution Order

2.2 Tasks priorities

In order to test the behavior of the scheduler, the priorities of the demo code (and even the tasks) have been modified: 3 different configurations of the priorities have been used.

To understand the following memory dump consider that the value 0001 is written by Task1, while 0002 is written by Task2.

2.2.1 Test configuration 1

During **TEST 1** the different tasks have the same priority. The amount of Flash where the program tasks can write is limited in order to simplify the result reading. In case of correct functioning the flash memory will display the alternation of the writes of Task1 and Task2, as shown in figure 2.3.

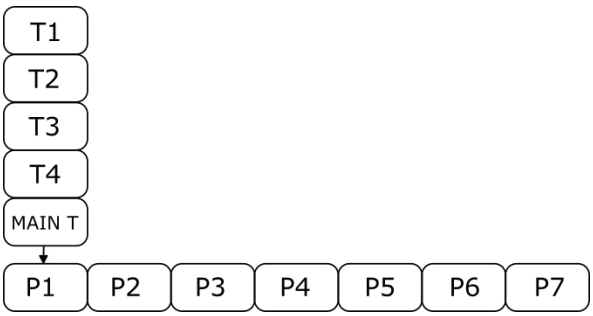

Figure 2.2: Test configuration 1

2.2.2 Test configuration 2

During **TEST 2** tasks do not have the same priority. In particular Main Task and Task4 have the priority 1, Task1 priority 2, Task2 priority 5 and Task3 priority 3. In case of correct functioning, the flash memory page where the program writes will be equal as Task1's one (figure 2.3).

Task	**Priority** [1-7]	
	Test1	Test2
Main Task	1	6
Task1	1	2
Task2	1	5
Idle Task	1	3
Task4	1	1

Address	0	2	4	6	8	A	C	E
0x08032000	0002	0001	0001	0001	0001	0001	0002	0001
0x08032010	0001	0001	0001	0002	0001	0001	0001	0001
0x08032020	0001	0002	0001	0001	0001	0001	0001	0002
0x08032030	0001	0001	0001	0001	0002	0001	0001	0001
0x08032040	0001	0001	0002	0001	0001	0001	0001	0002
0x08032050	0001	0001	0001	0001	0001	0002	0001	0001
0x08032060	0001	0001	0002	0001	0001	0001	0001	0001
0x08032070	0002	0001	0001	0001	0001	0002	0001	0001
0x08032080	0001	0001	0001	0002	0001	0001	0001	0001
0x08032090	0002	0001	0001	0001	0001	0001	0002	0001
0x080320A0	0001	0001	0001	0002	0001	0001	0001	0001
0x080320B0	0001	0002	0001	0001	0001	0001	0002	0001
0x080320C0	0001	0001	0001	0001	FFFF	FFFF	FFFF	FFFF

Figure 2.3: Memory Dump of Task1 and Task2

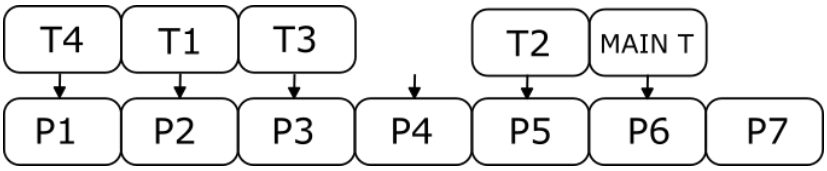

Figure 2.4: Test configuration 2

2.2.3 Test configuration 3

The **TEST 3** purpose is to put the scheduler under pressure: several task doing the same job are created.

Multiple instances of Task1 and Task2 are in fact created, giving them different priority combinations. While on the visual inspection of the LEDs it was impossible to understand the differences between

the tasks, for the flash memory has been used an offset (passed as parameter at the creation of the task) to understand which task was writing onto the memory.
The priorities of the Main Task, Idle Task and Task4 are the same as wtextbfTEST2, while there are 5 instances of Task1 and 5 instances of Task2, having priorities from 1 to 5, as show in the table.

Task	**Priority [1-7]**
`Main Task`	6
`Task1_0`	1
`Task1_1`	2
`Task1_2`	3
`Task1_3`	4
`Task1_4`	5
`Task2_0`	1
`Task2_1`	2
`Task2_2`	3
`Task2_3`	4
`Task2_4`	5
`Idle Task`	3
`Task4`	2

In the memory dump of figure 2.5 is shown the alternation between the various tasks. The 2 right digits indicate Task1 (01) or Task2 (02), while the 2 digits on the left indicate the priority (01 - 05).

Address	0	2	4	6	8	A	C	E
0x08032000	0502	0501	0402	0401	0302	0301	0202	0201
0x08032010	0102	0101	0501	0401	0301	0201	0101	0501
0x08032020	0401	0301	0201	0101	0501	0401	0301	0201
0x08032030	0101	0501	0401	0301	0201	0101	0502	0402
0x08032040	0302	0202	0102	0501	0401	0301	0201	0101
0x08032050	0501	0401	0301	0201	0101	0501	0401	0301
0x08032060	0201	0101	0501	0401	0301	0201	0101	0502
0x08032070	0501	0402	0401	0302	0301	0202	0201	0102
0x08032080	0101	0501	0401	0301	0201	0101	0501	0401
0x08032090	0301	0201	0101	0501	0401	0301	0201	0101
0x080320A0	0501	0401	0301	0201	0101	0502	0402	0302
0x080320B0	0202	0102	0501	0401	0301	0201	0101	0501
0x080320C0	0401	0301	0201	0101	FFFF	FFFF	FFFF	FFFF

Figure 2.5: Memory Dump of Task3

CHAPTER 3

Faults description

In order to test different parts of the scheduler, it has been decided to inject six different faults. To properly inject a fault is needed to bypass the protection of the Privileged data which are handling the scheduler and tasks informations. To access to this part of code, a new function has been added to *task.c*, declaring it into the header file, which let the *main.c* program modify the values.

```
 /* INJECTION FUNCTION */
Pointer_to_SRAM GET_SRAM_scheduler_pointers( void ) {
  Pointer_to_SRAM SRAM;
  SRAM.point_pxReadyTasksLists  = &pxReadyTasksLists  ;
  SRAM.point_pxDelayedTaskList  = pxDelayedTaskList   ;
  SRAM.point_xPendingReadyList  = &xPendingReadyList  ;
  SRAM.point_uxTopReadyPriority = &uxTopReadyPriority ;
  SRAM.point_xSchedulerRunning  = &xSchedulerRunning  ;

  return SRAM;
}
```

The return of this function is simply a *struct* containing the SRAM addresses of the variable we are interested in. This is possible just because the code of FreeRTOS is open source, otherwise we could not obtain those addresses.
Once the addresses are obtained (in the file *main.c*), the variable or the struct is accessed and the LSB of the target variable is flipped doing a bitwise XOR operation with a 1.
In order to have more than one fault, in the *main.c*, has been done a *switch statement* to decide the target variable. The code is listed below.

```
switch (FAULT) {
  case 1:
    /* Ready Tasks List
     * Change the number of Prioritized ready tasks. */
    ((List_t *)
      (data_addr.point_pxReadyTasksLists))->uxNumberOfItems  ^= 0x01;
    break;
  case 2:
    /* Delayed Task List <= Points to the delayed task list currently being
        used.
     * Change the number of Delayed tasks. */
    ((List_t *)
      (data_addr.point_pxDelayedTaskList))->uxNumberOfItems  ^= 0x01;
    break;
```

```
  case 3:
    /* Pending Ready List <= Tasks that have been readied while the
        scheduler was suspended.
     * They will be moved to the ready list when the scheduler is resumed.
     * Change the number of Pending ready tasks. */
    ((List_t *)
      (data_addr.point_xPendingReadyList))->uxNumberOfItems ^= 0x01;
    break;
  case 4:
    /* Top Ready Priority */
     * Change the priority of the top ready task. */
    *((UBaseType_t *)
      (data_addr.point_uxTopReadyPriority)) ^= 0x01;
    break;
  case 5:
    /* Scheduler Running */
    /* Change the state of the scheduler */
    *((UBaseType_t *)
      (data_addr.point_xSchedulerRunning)) ^= 0x01;
    break;
  case 6:
    /* Current task Priority */
    *((UBaseType_t *)
      (data_addr.point_uXPriority)) ^= 0x01;
    break;
  default:
    break;
}
```

The following part of this chapter covers the different variables where the faults have been injected. The choice of the target variables has been done studying the different functions and structures contained into the file *tasks.c*, going to isolate the scheduler related ones and choosing among them the ones with the highest probability to have major effects on the scheduler behavior.

Ready Tasks List

The target of the Ready task list is the number of prioritized ready tasks. This is actually done modifying the length of the queue from which the scheduler is going to pick the tasks.
This fault injection is expected to strongly affect the Scheduler behavior.

Delayed Task List

The target is the number of delayed task: this fault injection is similar to the previous one, but the modification happens to the delayed task list, that should be empty. A change in the order of execution is expected.

Pending Ready List

This variable points to the list containing info about the Tasks that have been readied while the scheduler was suspended. They will be moved to the ready list when the scheduler is resumed. Once again, the number of items is modified.

Top Ready Priority

This variable contains the priority of the top ready task: a change in the value could have strong effects in the scheduling.

Scheduler Running

This variable contains the scheduler state (true or false): it states if the scheduler is on or off. This could probably be the strongest fault that could affect the Scheduler.

Current Task Control Block Priority

This variable contains the priority of the current task. The behavior of the scheduler could be strongly affected when the priority of the current task is changed while it is still running.

3.1 Fault Injection

To inject the fault and to have a regular pattern to check in case of errors or simply to have a better understanding of the memory writes, there has been the necessity to inject the fault exploiting the timer interrupt of the system.

Timer

In order to be able to properly simulate a bit flip behavior, the event had to be triggered using a method external to the OS. This means that the systick timer couldn't be used, but an actual physical one. The used board has several timers with a big variety of functions. After a comparison of the different timers, *timer7* has been chosen, because of its simplicity and therefore low power consumption. The declaration and activation of the timer is managed in the function `TIM7_Configuration` which receives as arguments the prescaler value and the number of ticks to count.

```
/**
 * @brief  Initialize TIM7
 * @note   Set TIM7 in up-count mode
 * @param  @param1 Value that will be counted
 *         @param2 Ideal Prescaler value
 * @retval None
 */
void TIM7_Configuration (uint32_t value_to_count, uint32_t Prescaler_ideal) {

  __TIM7_CLK_ENABLE();
  timer7.Instance               = TIM7;
  timer7.Init.Prescaler         = Prescaler_ideal - 1;
  timer7.Init.CounterMode       = TIM_COUNTERMODE_UP;
  timer7.Init.Period            = value_to_count;
  timer7.Init.ClockDivision     = TIM_CLOCKDIVISION_DIV1; // no division
  timer7.Init.RepetitionCounter = 0; // IRQ must happen only once
  timer7.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;

  HAL_NVIC_SetPriority(TIM7_IRQn, 0x05, 0x00); // NOT ROBUST: if you put
      more than 15 @arg2 error
  /* Enable the TIM7 global Interrupt*/
  HAL_NVIC_EnableIRQ(TIM7_IRQn);
```

```
  HAL_TIM_Base_Init     (&timer7);
  HAL_TIM_Base_Start_IT (&timer7);
  return;
}
```

The clock source has been manually programmed in another function. In order to know at which point in the flash memory the interrupt executed the fault injection, a write on the flash memory has been made by the interrupt handling function.

CHAPTER 4

Injection results

4.1 Expected Results

In this part are described the expected behaviors of the scheduler after the fault injection. Since the fault injection is done with a bitwise XOR with the target variable, the possible actions done by this fault are the incrementing or the decrementing of the variable.
The difference between the different test could be that, in the case the running tasks are more and with a different priority, the program avoids the complete crash, but continues with a not regular behavior (i.e. one of the tasks doesn't run anymore).

1. **Ready Task List**: in case it is decremented, a ready task is removed from the queue, while if it is incremented the queue will think to have a ready task which is actually not existing. In both cases it is expected a scheduler fail, meaning that the program will crash and won't be able to go on.
2. **Delayed Task List**: it is expected a similar behavior to the previous case.
3. **Pending Ready List**: it is expected a similar behavior to the previous case.
4. **Top Ready List**: it is expected a similar behavior to the previous case.
5. **Scheduler Running**: the program should block.
6. **Current TCB priority**: The behavior of the program is strongly depending on the priority of the different tasks. It could happen anything or the program could change behavior very significantly.

4.2 Obtained Results

Hereinafter are reported the different behaviors of the scheduler after the fault injection, along with the memory dumps.
In the final part of the previous chapter has been described the method used to understand at which point of the writes on the flash memory was the fault injected. In both TEST1 and TES2 the writes happened in the same position where the working faults make the scheduler stop working. In TEST3 the write happened in the same position where the case 1 and case 4 stop.

4.2.1 TEST1

1. **Ready Task List**: the program stops working. The memory dump is shown in figure 4.1.

Address	0	2	4	6	8	A	C	E
0x08032000	0002	0001	0001	0001	0001	0001	0002	0001
0x08032010	0001	0001	0001	0002	0001	FFFF	FFFF	FFFF
0x08032020	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032030	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032040	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032050	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032060	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032070	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032080	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032090	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320A0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320B0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320C0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

Figure 4.1: Memory Dump of TEST1 and TEST2 after the program stopped working

2. **Delayed Task List**: the program runs without any problem. **No fault**.
3. **Pending Ready List**: the program stops working in the same way as case 1. See figure 4.1.
4. **Top Ready List**: the program stops working in the same way as case 1. See figure 4.1.
5. **Scheduler Running**: the program stops working in the same way as case 1. See figure 4.1.
6. **Current TCB priority**: the program runs without any problem. **No fault**.

4.2.2 TEST2

1. **Ready Task List**: the program stops working, as case 1 of TEST1. Same memory dump, See figure 4.1.
2. **Delayed Task List**: the program runs without any problem. **No fault**.
3. **Pending Ready List**: the program stops working in the same way as case 1. See figure 4.1.
4. **Top Ready List**: the program stops working in the same way as case 1. See figure 4.1.

5. **Scheduler Running**: the program stops working in the same way as case 1. See figure 4.1..
6. **Current TCB priority**: the program runs without any problem. **No fault**.

4.2.3 TEST3

1. **Ready Task List**: the program stops working. The memory dump is shown in figure 4.2.

Address	0	2	4	6	8	A	C	E
0x08032000	0502	0501	0402	0401	0302	0301	0202	0201
0x08032010	0102	0101	0501	0401	0301	0201	0101	0501
0x08032020	0401	0301	0201	0101	0501	0401	0301	0201
0x08032030	0101	0501	0401	0301	0201	0101	0502	0402
0x08032040	0302	0202	0102	0501	0401	0301	0201	0101
0x08032050	0501	0401	0301	0201	0101	0501	0401	0301
0x08032060	0201	0101	0501	0401	0301	0201	0101	0502
0x08032070	0501	0402	0401	0302	0301	0202	0201	0102
0x08032080	0101	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032090	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320A0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320B0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320C0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

Figure 4.2: Memory Dump of TEST3 after the program stopped working

2. **Delayed Task List**: the program runs without any problem. **No fault**.
3. **Pending Ready List**: the program stops working later than the case 1. The memory dump is shown in figure 4.3.

Address	0	2	4	6	8	A	C	E
0x08032000	0502	0501	0402	0401	0302	0301	0202	0201
0x08032010	0102	0101	0501	0401	0301	0201	0101	0501
0x08032020	0401	0301	0201	0101	0501	0401	0301	0201
0x08032030	0101	0501	0401	0301	0201	0101	0502	0402
0x08032040	0302	0202	0102	0501	0401	0301	0201	0101
0x08032050	0501	0401	0301	0201	0101	0501	0401	0301
0x08032060	0201	0101	0501	0401	0301	0201	0101	0502
0x08032070	0501	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032080	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x08032090	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320A0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320B0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0x080320C0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

Figure 4.3: Memory Dump of TEST3 after the program stopped working

4. **Top Ready List**: the program stops working in the same way as case 1. See figure 4.2.
5. **Scheduler Running**: the program stops working in the same way as case 3. See figure 4.3.
6. **Current TCB priority**: the program runs without any problem. **No fault**.

CHAPTER 5

Conclusion

The fault injection has been successful: the program, when the fault timer interrupt injected the fault, crashed and it wasn't possible to return in an operative state.