Politecnico di Torino

Operating Systems

Fault injection

Report

Referents: Maurizio Rebaudengo, Masoud Hemmatpour

Authors: Cesarano Giuseppe, Fogliato Martina, Simili Michele

Contents

Aim of the project i
Application Program i
Interrupt Service Routine ii
Fault injection iii
Experimental session iv

Aim of the project

The aim of this project was writing a run time application program, to be loaded on **STM32f303** Discovery Board, which simulates a fault injection through the implementation of an interrupt service routine which simply flips a bit of a specific register. In particular, the consequences of modifying a register of a component of the Real Time Operating System **FreeRTOS**.
In our case the component to be targeted was semaphore management.

Application Program

For the development of our application program we chose as environment **Eclipse Neon**, selecting the Ac6 STM32 MCU Project option and including, as third party utility, FreeRTOS.
Our application basically behaves as a compass: it consists of two tasks in which one of them is reading from the magnetometer (**lsm303dlhc** sensor of the board, including both magnetometer and accelerometer) and the other one switches on and off the LEDs in a circular fashion.
In main.c, after configuring the LEDs and the sensor (220 Hz of sampling frequency, 1.3 Gauss of range and Continuous Conversion Mode) we declared the two tasks:

```
xTaskCreate(MAG_read, (const char *) "MAG", configMINIMAL_STACK_SIZE,
    NULL, 1, NULL);
xTaskCreate(compass, (const char *) "LED", configMINIMAL_STACK_SIZE,
    NULL, 1, NULL);
```

It can be seen that we set both tasks to the same priority.
A semaphore has also been added in order to synchronize the tasks; ideally whenever one of the two tasks finishes its execution (for instance, a sensor reading), it gives the semaphore which will then be taken by the other task. This binary semaphore is created by the MAG_read task and we specified the time in ticks to wait for the semaphore to become available:

```
xSemaphoreTake(Semaphore, (portTickType)10);
```

The magnetometer task runs as an infinite loop (for(;;)); each time it performs six readings from the registers of the sensor (3 coordinates, which are splitted in two 8-bits registers each) filling a buffer. Then, it computes x,y and z concatenating the two registers and eventually computes the direction of the horizontal magnetic field vector with the arctangent among y and x coordinates.
Variables x, y, z and angle are all global and can be seen by both tasks.
In turn, the LEDs task, reading the angle value stored by the MAG_read task, switches on only the LED which corresponds to the North direction.
Whenever one of the two tasks terminates one execution it requests a context switch using function taskYIELD().
We declared two arrays (opMag and opLed) in order to count the number of operations performed by each task. Every 0.1 seconds a new count is performed and every time a task executes it increments the count. The number of entries in the arrays is saturated to 40 since we run our application for 4 seconds only.

Interrupt Service Routine

We decided to use two timers, so that one of them can be used to count the number of operations performed by each task in a given amount of time (1/10 of second in our case), while the other one sets a fault once (after 1, 2 or 3 seconds after the scheduler starts).
For our ISR we used **TIM2** and **TIM3** of the Board, which run at a frequency of 8 MHz.
In a separate file (tim.c) we configured the registers of the two timers for our purposes:

```
void TIM2_init()
{
  RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
    NVIC->ISER[0] |= (1<<28);
    TIM2->CR1 = 0;
    TIM2->CR2 = 0;
    TIM2->DIER = TIM_DIER_UIE; //enable interrupt
    TIM2->ARR = 10000; // 1/10 of second
    TIM2->PSC = 79; //prescaler at 80
    TIM2->CR1 |= 1;
}

void TIM3_init()
{
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
    NVIC->ISER[0] |= (1<<29);
    TIM3->CR1 = 0;
    TIM3->CR2 = 0;
    TIM3->DIER = TIM_DIER_UIE; //enable interrupt
    TIM3->ARR = 20000; // number of seconds for fault injection (1, 2 or
                          3), 10000 ticks = 1 sec
    TIM3->PSC = 799; // prescaler at 800
    TIM3->CR1 |= 1;
```

```
}
```

The interrupts given by these timers are served by two ISR.

Fault injection

The aim of the fault injection was to modify run time the value of a variable related to semaphores, in particular those used by the kernel of the FreeRTOS itself, and monitor the results in different conditions.
At first we had an accurate look at the source files of FreeRTOS, in particular task.c and queue.c (since semaphores are actually queues). In particular, we concentrated on the two variables which were suggested: **pxMutexHolder** and **uxMutexesHeld**.
In order to get some help we also wrote on FreeRTOS forum, but they answered that the kernel does not use itself any mutex.
We found out that pxMutexHolder is defined as a pointer to **pcTail** which points to the byte at the end of the queue storage area (or to mutex holder, if any) and is used inside a structure called xQUEUE in queue.c.
Unfortunately we could not access to it or modify it.
As for uxMutexesHeld, we found out that this variable belongs to the Thread Control Block of each task. Therefore we added a function directly in file task.c which returns a pointer to the TCB of the currently running task:

```
TCB_t * GetReadyTasksHandle( void )
{
  return pxCurrentTCB;
}
```

We called this function in the ISR related to TIM3 in order to get the TCB of the task which is running when the interrupt is sent (we called it Fault).

```
TCB_t *Fault;
Fault = GetReadyTasksHandle();
```

In this way we could access all the variables which constitute the TCB, among which uxMutexesHeld.
However, whichever value we try to assign to this variable we could see no changes in the behavior of our program.
Instead, we managed to modify another variable, again belonging to the TCB and called **uxPriority**, which influences much more the behavior of the application, since it represents the priority of the running task.
Example:

```
(Fault->uxPriority) = (1<<0);
```

Experimental session

Once found the target variable for our fault injection, we started the experimental session.
In order to monitor the changes in all the global variables (like x, y, z and angle) and the number of operations performed by each task (opMag and opLed arrays), we used a tool called STM Studio.
It is a software by STMicroelectronics which helps debugging applications while they are running by reading and displaying their variables, preserving the real-time behavior of programs.

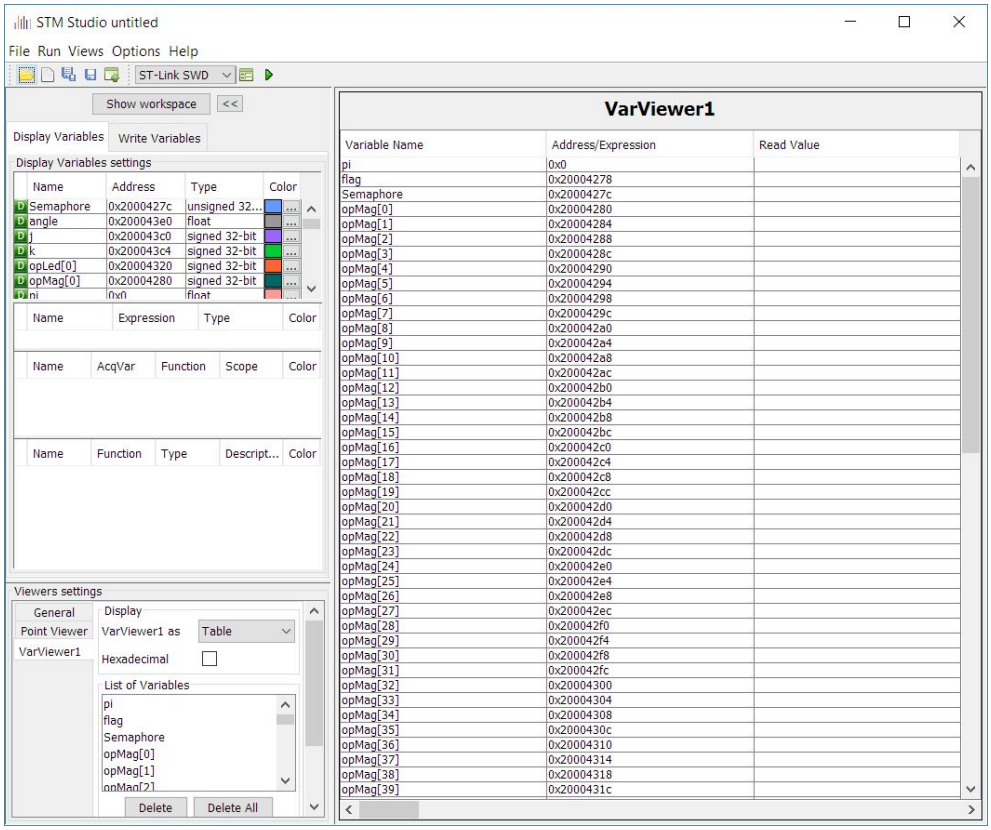

Figure 1: Graphical interface of STM Studio while running

In this way we could easily monitor the global variables of our application.
Overall we inserted 51 faults: we changed one bit at a time starting from bit 0 to bit 11, then bit 18, 25, 27 and 31. For each bit we tried to insert the same fault after 1 second of execution, then after 2 seconds and eventually after 3 seconds.
In the file fault_injection_results.pdf it is possible to see all results of our experiments, taking into account that during a normal run of the application each task performs between 35 and 40 operations .
We noticed some main patterns in our results. When the modified bit belongs to the TCB of MAG_read task, the LED task stops executing and its number of operations is 0; viceversa when the modified bit belongs to the TCB of LED task, the MAG task stops executing.
This behavior changes a little when modifying a more significant bit (ex. bit 5), when the task which has not been blocked starts performing an increasing number of operations in 1/10 second (getting to more than 1000 operations in some cases). An example can be seen in Figure 2.
Sometimes (ex. bit 11) both task block when the fault is injected.
Another interesting behavior we noticed, is that in some cases (ex. bit 0) when the bit is modified variable j (which is the index for opLed array) starts changing randomly and the number of operations

stored in the array increases consequently and is not reliable anymore.
More precise results relative to each fault injection can be seen in the following table:

Type of fault	**Interested thread**	**Comments**
FAULT1:1 << 0 at 1s	LedTCB	All variables are 0's
FAULT2:1 << 0 at 2s	LedTCB	unusual increase of j: more operations per ms
FAULT3:1 << 0 at 3s	LedTCB	unusual increase of j: more operations per ms
FAULT4:1 << 1 at 1s	MagTCB	Led thread stops after 1 second (Mag continues)
FAULT5:1 << 1 at 2s	MagTCB	Led thread stops after 2 seconds (Mag continues)
FAULT6:1 << 1 at 3s	MagTCB	Led thread stops after 3 second (Mag continues)
FAULT7:1 << 2 at 1s	MagTCB	Led thread stops after 1 second (Mag continues)
FAULT8:1 << 2 at 2s	LedTCB	Led thread stops after 2 seconds (Mag continues)
FAULT9:1 << 2 at 3s	MagTCB	Mag stops after 3 seconds; more ops per ms in Led after 3 seconds
FAULT10:1 << 3 at 1s	MagTCB	Led thread stops after 1 second (Mag continues)
FAULT11:1 << 3 at 2s	MagTCB	Led thread stops after 2 seconds (Mag continues)
FAULT12:1 << 3 at 3s	MagTCB	Led thread stops after 3 seconds (Mag continues)
FAULT13:1 << 4 at 1s	MagTCB	Both threads stop after 1 second
FAULT14:1 << 4 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT15:1 << 4 at 3s	LedTCB	Both threads stop after 3 seconds
FAULT16:1 << 5 at 1s	LedTCB	Mag stops after 1 second; more ops per ms in Led after 1 second
FAULT17:1 << 5 at 2s	LedTCB	Mag stops immediately; more ops in Led before 2 seconds, then it stops
FAULT18:1 << 5 at 3s	LedTCB	Mag stops immediately; more ops in Led before 1 second, then it stops
FAULT19:1 << 6 at 1s	MagTCB	Both threads stop after 1 second
FAULT20:1 << 6 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT21:1 << 6 at 3s	MagTCB	Both threads stop after 3 seconds
FAULT22:1 << 7 at 1s	MagTCB	Both threads stop after 1 second
FAULT23:1 << 7 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT24:1 << 7 at 3s	MagTCB	Both threads stop after 3 seconds
FAULT25:1 << 8 at 1s	MagTCB	Both threads stop after 1 second
FAULT26:1 << 8 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT27:1 << 8 at 3s	MagTCB	Both threads stop after 3 seconds
FAULT28:1 << 9 at 1s	MagTCB	Both threads stop after 1 second
FAULT29:1 << 9 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT30:1 << 9 at 3s	LedTCB	Both threads stop after 3 seconds
FAULT31:1 << 10 at 1s	MagTCB	Both threads stop after 1 second
FAULT32:1 << 10 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT33:1 << 10 at 3s	MagTCB	Both threads stop after 3 seconds
FAULT34:1 << 11 at 1s	MagTCB	Both threads stop after 1 second
FAULT35:1 << 11 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT36:1 << 11 at 3s	MagTCB	Both threads stop after 3 seconds
FAULT37:1 << 18 at 1s	MagTCB	Both threads stop after 1 second
FAULT38:1 << 18 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT39:1 << 18 at 3s	MagTCB	Both threads stop after 3 seconds

FAULT40:1 << 25 at 1s	MagTCB	Both threads stop after 1 second
FAULT41:1 << 25 at 2s	MagTCB	Both threads stop after 2 seconds
FAULT42:1 << 25 at 3s	MagTCB	Both threads stop after 3 seconds
FAULT43:1 << 27 at 1s	LedTCB	Both threads stop after 1 second
FAULT44:1 << 27 at 2s	LedTCB	Both threads stop after 2 seconds
FAULT45:1 << 27 at 3s	LedTCB	Both threads stop after 3 seconds
FAULT46:1 << 30 at 1s	LedTCB	Mag stops after 1 second; more ops per ms in Led after 1 second
FAULT47:1 << 30 at 2s	LedTCB	Mag stops after 2 seconds; more ops per ms in Led after 2 seconds
FAULT48:1 << 30 at 3s	MagTCB	Mag stops after 3 seconds; more ops per ms in Led after 3 seconds
FAULT49:1 << 31 at 1s	LedTCB	Mag stops after 1 second; more ops per ms in Led after 1 second
FAULT50:1 << 31 at 2s	LedTCB	Mag stops after 2 seconds; more ops per ms in Led after 2 seconds
FAULT51:1 << 31 at 3s	MagTCB	Mag stops after 3 seconds; more ops per ms in Led after 3 seconds

Table 1: Results of the fault injections

FAULT 46 : 1<<31 at 1 second

pi	0x0	1.0894961E-19
flag	0x20004278	1
Semaphore	0x2000427c	536874248
opMag[0]	0x20004280	0
opMag[1]	0x20004284	37
opMag[2]	0x20004288	37
opMag[3]	0x2000428c	37
opMag[4]	0x20004290	38
opMag[5]	0x20004294	37
opMag[6]	0x20004298	37
opMag[7]	0x2000429c	38
opMag[8]	0x200042a0	37
opMag[9]	0x200042a4	37
opMag[10]	0x200042a8	38
opMag[11]	0x200042ac	0
opMag[12]	0x200042b0	0
opMag[13]	0x200042b4	0
opMag[14]	0x200042b8	0
opMag[15]	0x200042bc	0
opMag[16]	0x200042c0	0
opMag[17]	0x200042c4	0
opMag[18]	0x200042c8	0
opMag[19]	0x200042cc	0
opMag[20]	0x200042d0	0
opMag[21]	0x200042d4	0
opMag[22]	0x200042d8	0
opMag[23]	0x200042dc	0
opMag[24]	0x200042e0	0
opMag[25]	0x200042e4	0
opMag[26]	0x200042e8	0
opMag[27]	0x200042ec	0
opMag[28]	0x200042f0	0
opMag[29]	0x200042f4	0
opMag[30]	0x200042f8	0
opMag[31]	0x200042fc	0
opMag[32]	0x20004300	0
opMag[33]	0x20004304	0
opMag[34]	0x20004308	0
opMag[35]	0x2000430c	0
opMag[36]	0x20004310	0
opMag[37]	0x20004314	0
opMag[38]	0x20004318	0
opMag[39]	0x2000431c	0

opLed[0]	0x20004320	0
opLed[1]	0x20004324	41
opLed[2]	0x20004328	43
opLed[3]	0x2000432c	44
opLed[4]	0x20004330	41
opLed[5]	0x20004334	39
opLed[6]	0x20004338	42
opLed[7]	0x2000433c	43
opLed[8]	0x20004340	43
opLed[9]	0x20004344	40
opLed[10]	0x20004348	47
opLed[11]	0x2000434c	452
opLed[12]	0x20004350	453
opLed[13]	0x20004354	452
opLed[14]	0x20004358	453
opLed[15]	0x2000435c	452
opLed[16]	0x20004360	453
opLed[17]	0x20004364	452
opLed[18]	0x20004368	453
opLed[19]	0x2000436c	452
opLed[20]	0x20004370	450
opLed[21]	0x20004374	444
opLed[22]	0x20004378	443
opLed[23]	0x2000437c	443
opLed[24]	0x20004380	444
opLed[25]	0x20004384	443
opLed[26]	0x20004388	443
opLed[27]	0x2000438c	443
opLed[28]	0x20004390	444
opLed[29]	0x20004394	443
opLed[30]	0x20004398	443
opLed[31]	0x2000439c	444
opLed[32]	0x200043a0	443
opLed[33]	0x200043a4	444
opLed[34]	0x200043a8	443
opLed[35]	0x200043ac	443
opLed[36]	0x200043b0	444
opLed[37]	0x200043b4	443
opLed[38]	0x200043b8	443
opLed[39]	0x200043bc	1057
j	0x200043c0	39
k	0x200043c4	39
i	0x200043cc	6
x	0x200043d0	-0.18454546
LedTCB	0x200043d4	536873520
Fault	0x200043d8	536873520
z	0x200043dc	-0.50306123
angle	0x200043e0	-170.21759
y	0x200043e4	-0.03181818
MainTCB	0x200043e8	0
MagTCB	0x200043ec	536872896

Figure 2: Fault injection result example (bit 31)