ECE 560: EMB. SYS. ARCHITECTURES

PROJECT 2: SHIELDS UP! REPORT

Cristian Hellmer

Crhellme

INTRODUCTION

This report details the efforts made to make a particular embedded system more reliable. The starter program flashes an LED and features a graphical UI on an LCD touchscreen display. The modifications made to the program protect the system from many different faults, including stack overflow, interference, and abnormal inputs. Protection measures implemented include resetting the system, reverting values in the case of unexplained changes, and reconfiguring components of the system which should not change.

FAULT TEST 1 - TR_DISABLE_ALL_IRQS

Fault Analysis

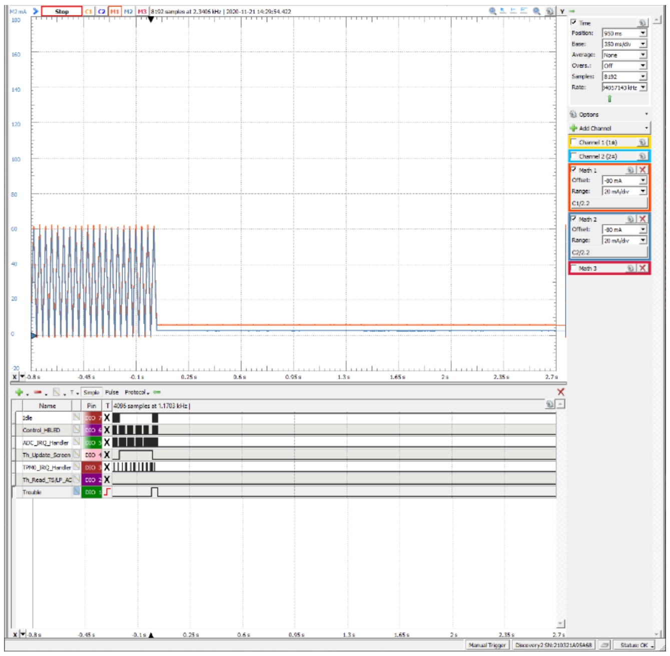

Position: 950 ms
Base: 350 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 4057143 kHz
Math 1 Offset: -80 mA Range: 20 mA/div C1/2.2
Math 2 Offset: -80 mA Range: 20 mA/div C2/2.2

Figure 1.1. The system stops working after the fault disables all interrupts.

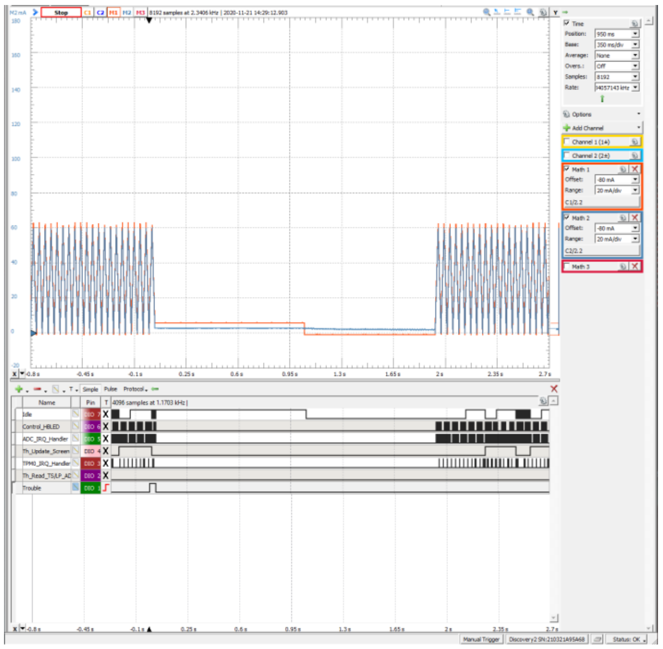

Position: 950 ms
Base: 350 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 4057143 kHz
Math 1 Offset: -80 mA Range: 20 mA/div C1/2.2
Math 2 Offset: -80 mA Range: 20 mA/div C2/2.2

Figure 1.2. Fault detection and response code resets the system in a working state.

The fault calls the function "__disable_irq()" which disables all interrupts used in the system. Figure 1.1 shows that this causes the system to become unresponsive.

Fault Management Approach

The fault is detected using a watchdog timer which is configured to trigger if there is a period of 1024 ms where the timer is not serviced. The watchdog timer is initialized using the function **Init_COP_WDT** which sets the time-out divider. **Service_COP_WDT** writes 0x55 and then 0xAA to SRVCOP to service the watchdog timer in the **Thread_Update_Screen** thread. Also, to enable the watchdog timer, the **DISABLE_WDOG** macro was changed to 0 for the watchdog timer to be able to be used.

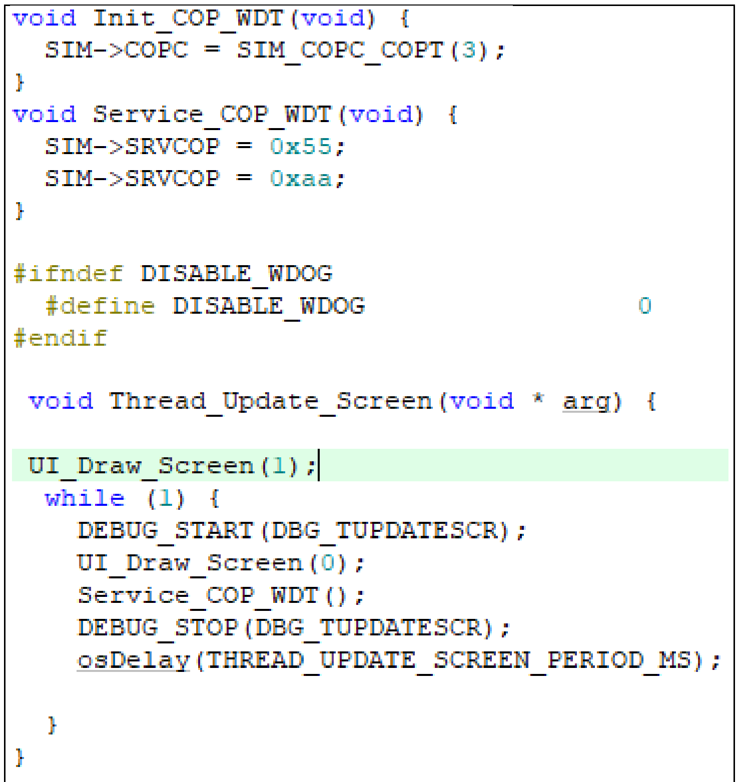

```
void Init_COP_WDT(void) {
  SIM->COPC = SIM_COPC_COPT(3);
}
void Service_COP_WDT(void) {
  SIM->SRVCOP = 0x55;
  SIM->SRVCOP = 0xaa;
}

#ifndef DISABLE_WDOG
  #define DISABLE_WDOG                 0
#endif

 void Thread_Update_Screen(void * arg) {

 UI_Draw_Screen(1);
  while (1) {
    DEBUG_START(DBG_TUPDATESCR);
    UI_Draw_Screen(0);
    Service_COP_WDT();
    DEBUG_STOP(DBG_TUPDATESCR);
    osDelay(THREAD_UPDATE_SCREEN_PERIOD_MS);

  }
}
```

Figure 1.3. Key source code additions.

Evaluation of Effectiveness

As shown in figure 1.2, the system resets approximately 1 second after the system stops working. This is a reasonable duration considering an error like this should be rare and since the error disrupts much of the functionality of the system. This change makes the system more robust in cases where the functionality of important parts of the code are suspended.

FAULT TEST 2 - TR_SETPOINT_HIGH

Fault Analysis

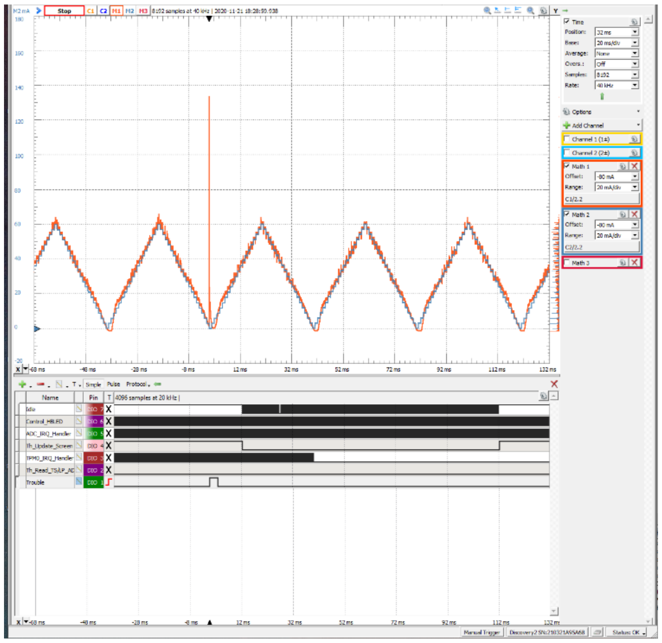


Figure 2.1. The system experiences a spike in current due to the fault.

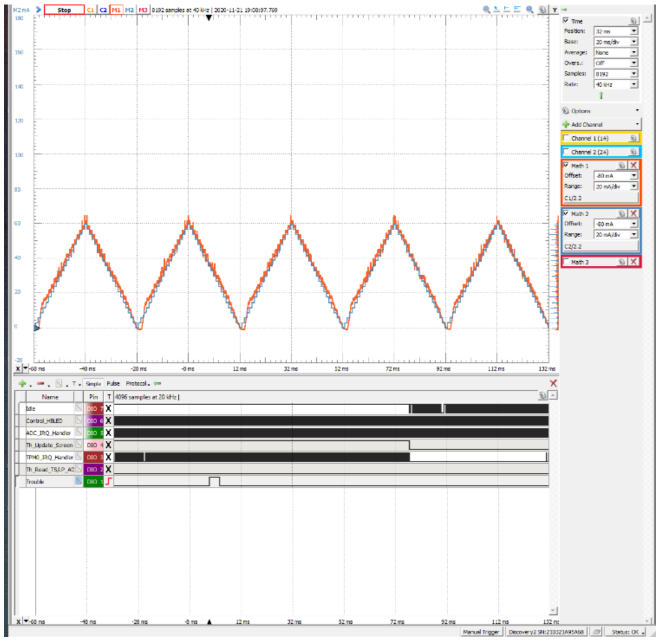


Figure 2.2. Fault detection and response code changes g_set_current to a reasonable value at the time of the fault.

The fault manually sets **g_set_current** to 1000. Figure 2.1 shows that this causes the system experiences a temporary spike to in current due to this change.

Fault Management Approach

To detect the fault, a conditional statement was added to the controller used in the system which checks for a large change in current. To make this happen, the **g_set_current** which was previously passed to the controller is recorded in the variable **previous_set_current**. The conditional statement looks to see if the new **g_set_current** is over 7 units away from the previous current to detect a fault. 7 was chosen since the function **Update_Set_Current** changes **g_set_current** by 3 units at a time and the fact that when a fault occurs, the recovered value will be, at most, 6 units away from the next value. If a fault is detected, the value of **g_set_current** is changed to equal the previous value.

Evaluation of Effectiveness

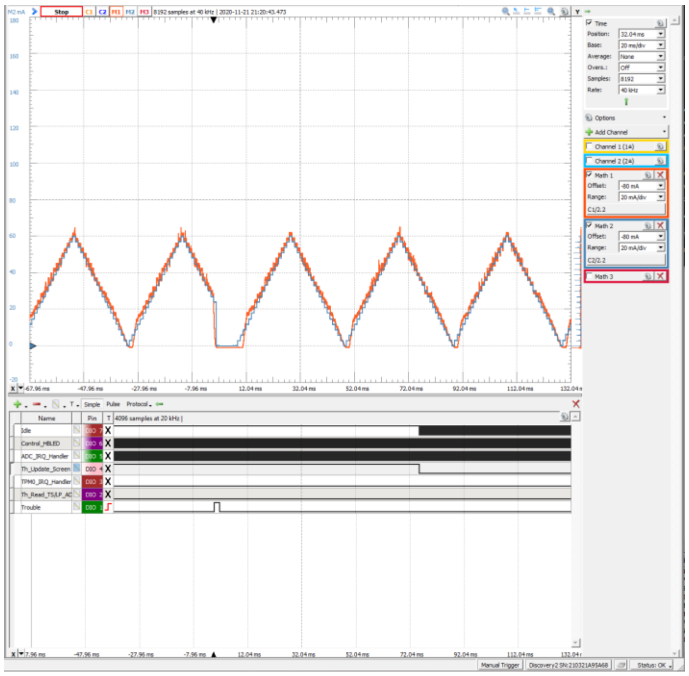
As shown in Figure 2.2, the system returns to normal operation in a negligible amount of time since the fault is detected as soon as the system tries to update the current driven to the LED. This change makes the system more reliable in cases where **g_set_current** is changed unintentionally. Managing of this fault causes the code to run slightly slower due to the conditional statements needing to be checked.

```
volatile int previous_set_current=0;

//Maximum change allowed in g_set_current
#define MAX_CURRENT_CHANGE (7)

case PID_FX:
  if(((g_set_current - previous_set_current) > MAX_CURRENT_CHANGE) |
      ((previous_set_current - g_set_current) > MAX_CURRENT_CHANGE)) {
    g_set_current = g_set_current;
  }
```

Figure 2.3. Key source code additions.

FAULT TEST 3 - TR_SETPOINT_ZERO

Fault Analysis



8192 samples at 40 kHz | 2020-11-21 21:20:43.473

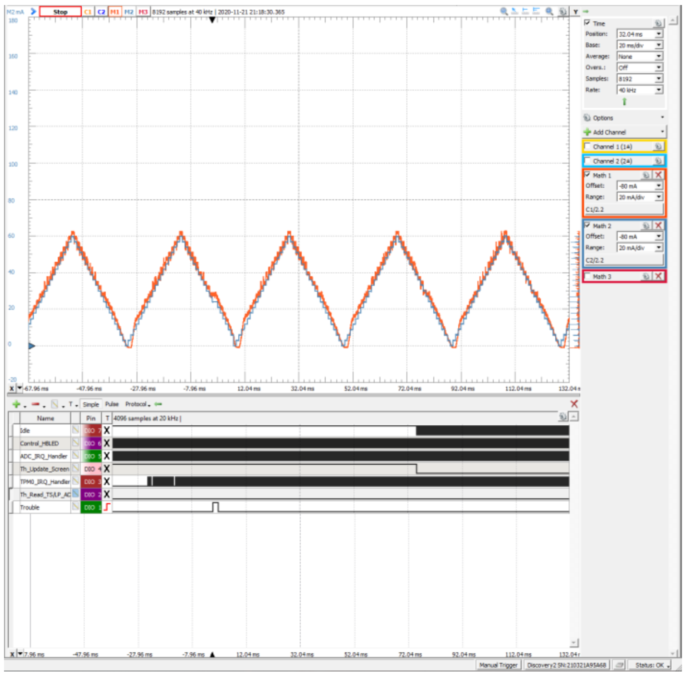
Time — Position: 32.04 ms; Base: 20 ms/div; Average: None; Overs.: Off; Samples: 8192; Rate: 40 kHz

Math 1 — Offset: -80 mA; Range: 20 mA/div

Math 2 — Offset: -80 mA; Range: 20 mA/div

4096 samples at 20 kHz |

Figure 3.1. The system flashing is interrupted by the fault by changing the current to 0.



8192 samples at 40 kHz | 2020-11-21 21:18:30.365

Time — Position: 32.04 ms; Base: 20 ms/div; Average: None; Overs.: Off; Samples: 8192; Rate: 40 kHz

Math 1 — Offset: -80 mA; Range: 20 mA/div

Math 2 — Offset: -80 mA; Range: 20 mA/div

4096 samples at 20 kHz |

Figure 3.2. Fault detection and response code changes g_set_current to a reasonable value at the time of the fault.

The fault manually sets **g_set_current** to 0. Figure 3.1 shows that this causes the system current to change to 0 until the next flash begins.

Fault Management Approach

See Fault Test 2.

Evaluation of Effectiveness

See Fault Test 2.

```
volatile int previous_set_current=0;

//Maximum change allowed in g_set_current
#define MAX_CURRENT_CHANGE (7)

case PID_FX:
  if(((g_set_current - previous_set_current) > MAX_CURRENT_CHANGE) |
      ((previous_set_current - g_set_current) > MAX_CURRENT_CHANGE)) {
    g_set_current = g_set_current;
  }
```

Figure 3.3. Key source code additions.

FAULT TEST 4 - TR_PID_FX_GAINS

Fault Analysis

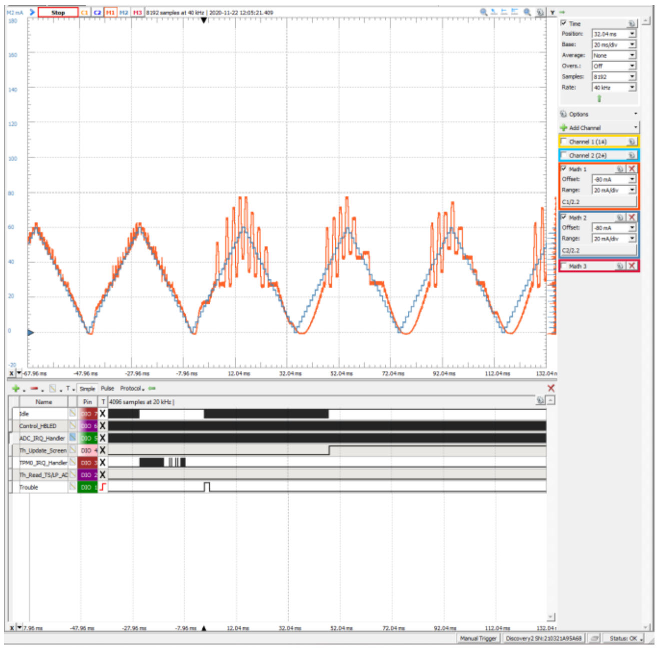


Figure 4.1. The system becomes erroneous after the fault is injected into the system.

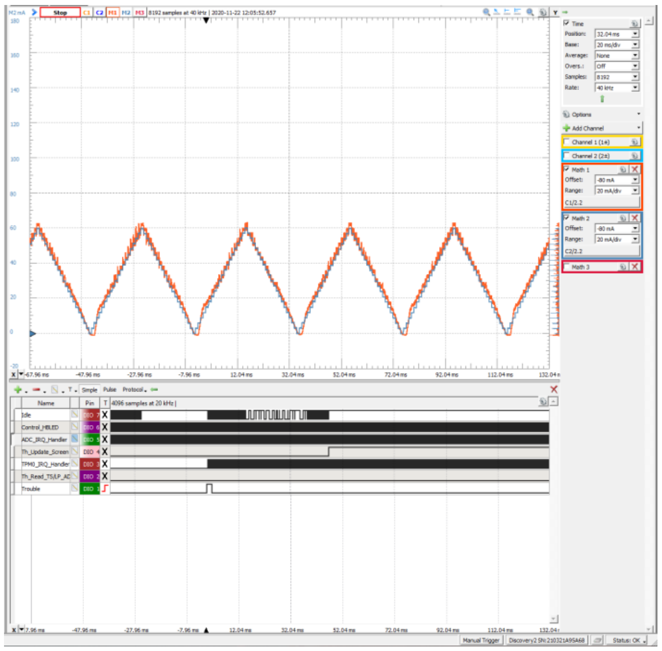


Figure 4.2. Fault detection and response code returns the system to the state it was before the fault.

The fault changes the variables **plantPID_FX.pGain**, **plantPID_FX.iGain**, **plantPID_FX.dGain** to values which cause the controller to not work well. As shown in figure 4.1, the current driven to the LED travels back and forth between way too high and way too low during flashes.

Fault Management Approach

The fault is detected using 3 conditional statements which check the **plantPID_FX.pGain**, **plantPID_FX.iGain**, **plantPID_FX.dGain** variables individually to see if they have changed from the values which make the system run properly. These values are defined using the macros **P_GAIN_FX**, **I_GAIN_FX**, and **D_GAIN_FX**. If they have changed, the variables are changed to equal the macros defines at the start of the program.

Evaluation of Effectiveness

As shown in figure 4.2, the system returns to normal operation in a negligible amount of time since the fault is detected as soon as the controller tries to update the duty cycle of the LED PWM signal. Managing of this fault causes the code to run slightly slower due to the conditional statements needing to be checked.

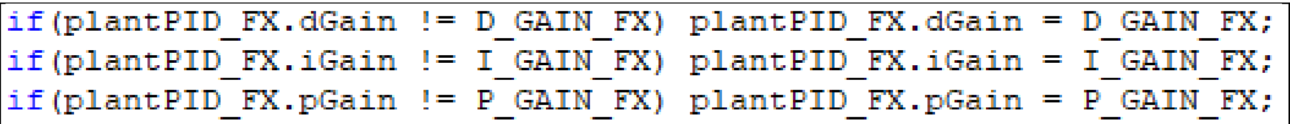

```
if(plantPID_FX.dGain != D_GAIN_FX) plantPID_FX.dGain = D_GAIN_FX;
if(plantPID_FX.iGain != I_GAIN_FX) plantPID_FX.iGain = I_GAIN_FX;
if(plantPID_FX.pGain != P_GAIN_FX) plantPID_FX.pGain = P_GAIN_FX;
```

Figure 4.3. Key source code additions.

FAULT TEST 5 - TR_LCD_MUTEX

Fault Analysis

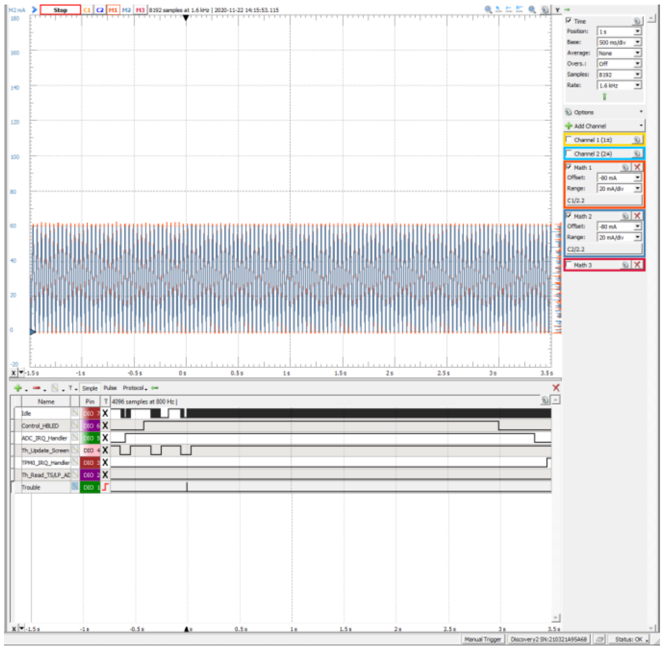


Figure 5.1. Thread_Update_Screen stops running after the fault is injected into the system.

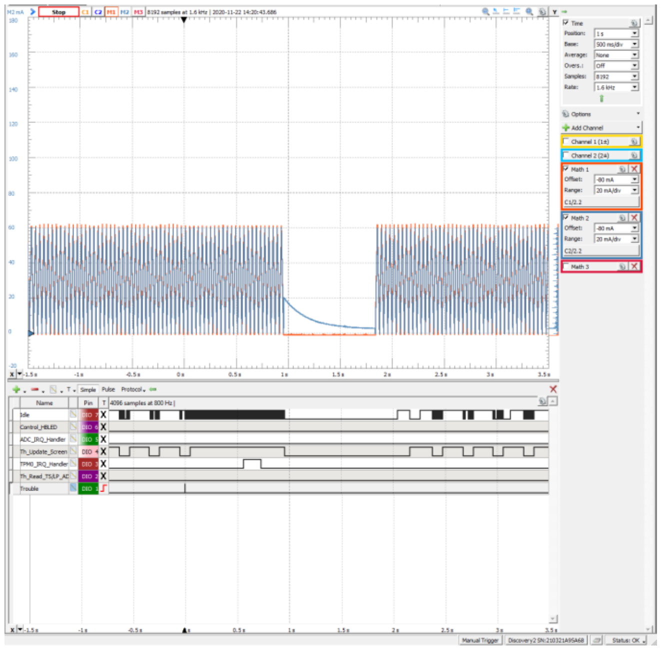


Figure 5.2. Fault detection and response code resets the system after the fault is detected and Thread_Update_Screen continues running.

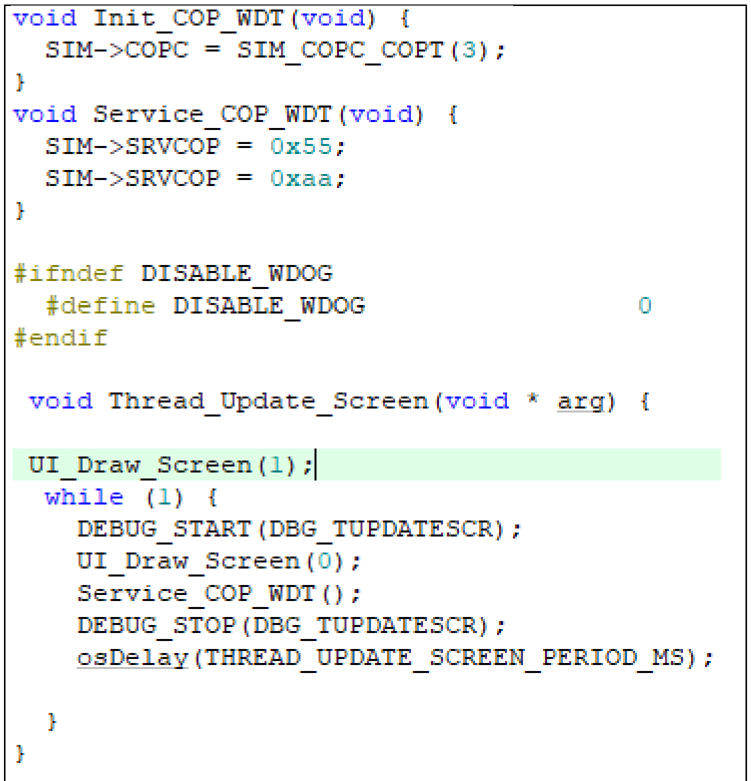
The fault acquires the **LCD_mutex** used by the LCD using **osMutexAcquire** and does not release the mutex afterwards. When the LCD functions attempt to acquire the **LCD_mutex**, they cannot. As shown in figure 5.1, this causes these functions to not execute the code in **Thread_Update_Screen**, which updates the screen.

Fault Management Approach

See Fault Test 1.

Evaluation of Effectiveness

See Fault Test 1.



```
void Init_COP_WDT(void) {
  SIM->COPC = SIM_COPC_COPT(3);
}
void Service_COP_WDT(void) {
  SIM->SRVCOP = 0x55;
  SIM->SRVCOP = 0xaa;
}

#ifndef DISABLE_WDOG
  #define DISABLE_WDOG              0
#endif

 void Thread_Update_Screen(void * arg) {

 UI_Draw_Screen(1);
  while (1) {
    DEBUG_START(DBG_TUPDATESCR);
    UI_Draw_Screen(0);
    Service_COP_WDT();
    DEBUG_STOP(DBG_TUPDATESCR);
    osDelay(THREAD_UPDATE_SCREEN_PERIOD_MS);

  }
}
```

Figure 5.3. Key source code additions.

FAULT TEST 6 - TR_DISABLE_ADC_IRQ

Fault Analysis

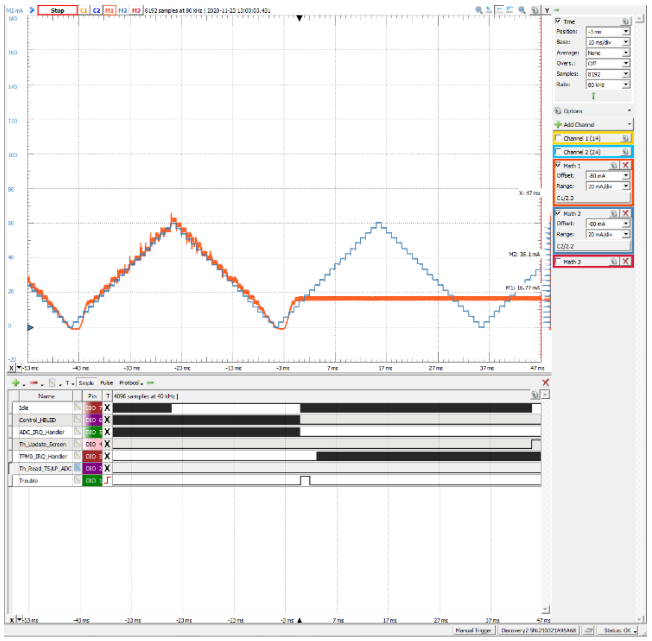


Figure 6.1. The current stops updating after the fault is injected into the system.

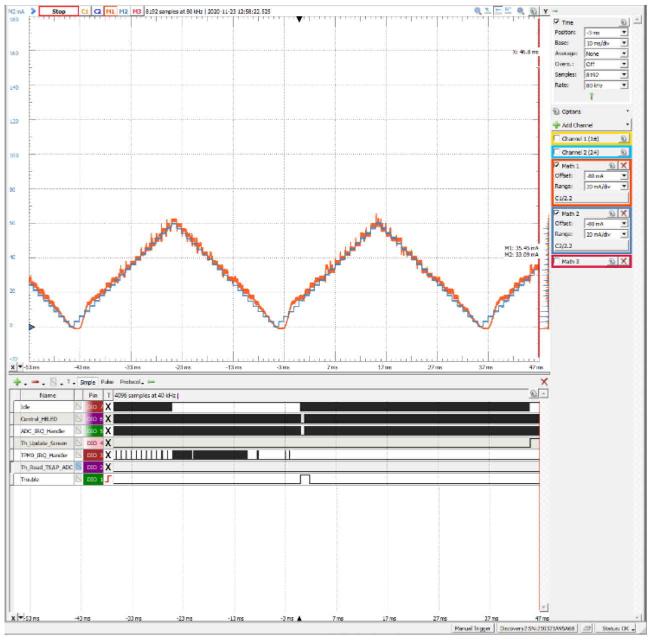
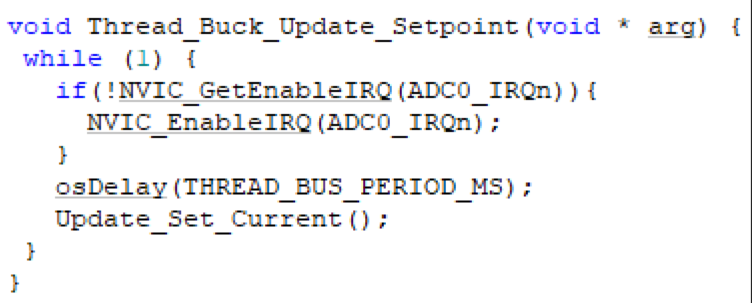


Figure 6.2. Fault detection and response code returns the system to the state it was before the fault.

The fault disables ADC interrupts by passing **ADC0_IRQn** to the function **NVIC_DisableIRQ**. As shown in figure 6.1, this causes the system to stop updating the current driven to the LED, leaving it at the value it was when the fault was originally injected.

Fault Management Approach

The fault is detected a conditional statement in **Thread_Buck_Update_Setpoint** which checks to make sure the ADC is enabled by passing **ADC0_IRQn** to the function **NVIC_GetEnableIRQ**. If the ADC is not enabled, then the ADC gets enabled by passing **ADC0_IRQn** to the function **NVIC_EnableIRQ**. The ADC is checked in this thread because correct operation of **Thread_Buck_Update_Setpoint** is dependent on correct ADC operation to use the setpoint update the LED current.



```
void Thread_Buck_Update_Setpoint(void * arg) {
 while (1) {
   if(!NVIC_GetEnableIRQ(ADC0_IRQn)){
     NVIC_EnableIRQ(ADC0_IRQn);
   }
   osDelay(THREAD_BUS_PERIOD_MS);
   Update_Set_Current();
 }
}
```

Figure 6.3. Key source code additions.

Evaluation of Effectiveness

As shown in figure 6.2, the system returns to normal operation in less than 1 ms since the fault is checked every time **Thread_Buck_Update_Setpoint** is ran. This is because **THREAD_BUS_PERIOD_MS** is 1. Managing of this fault causes the code to run slightly slower due to the conditional statement needing to be checked.

FAULT TEST 7 - TR_STACK_OVERFLOW

Fault Analysis

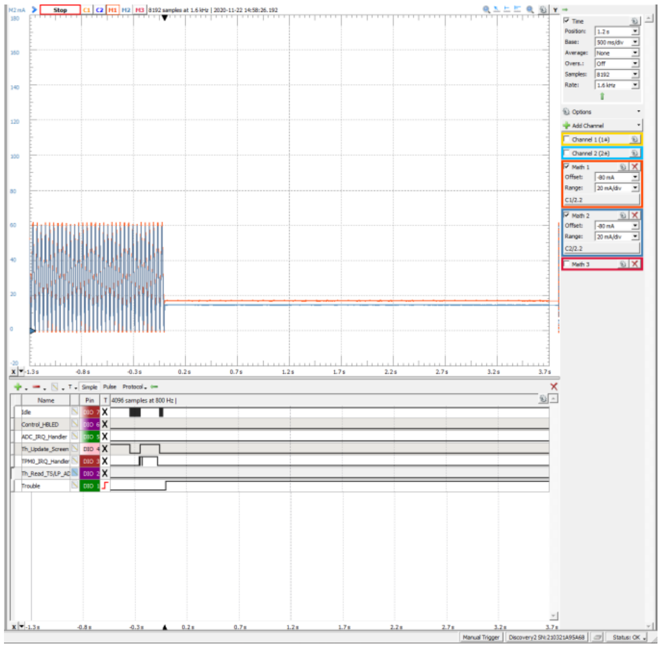


Figure 7.1. The system stops functioning after the fault is injected into the system.

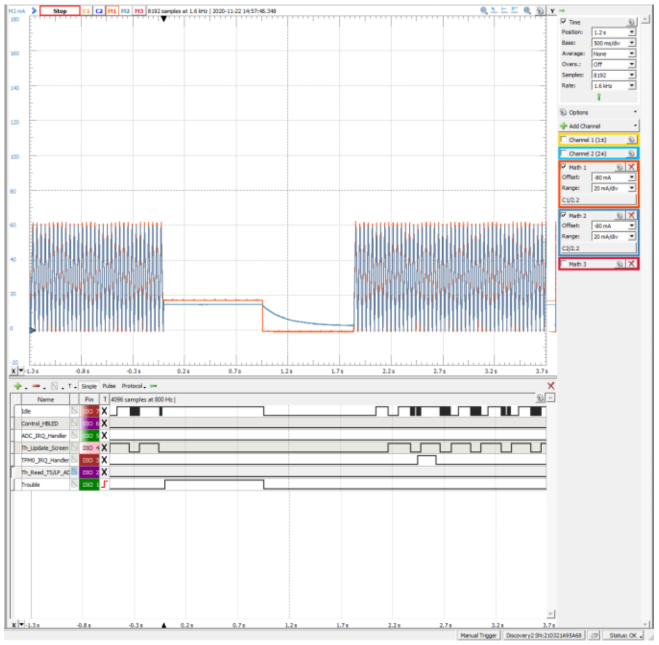


Figure 7.2. Fault detection and response code resets the system in a working state.

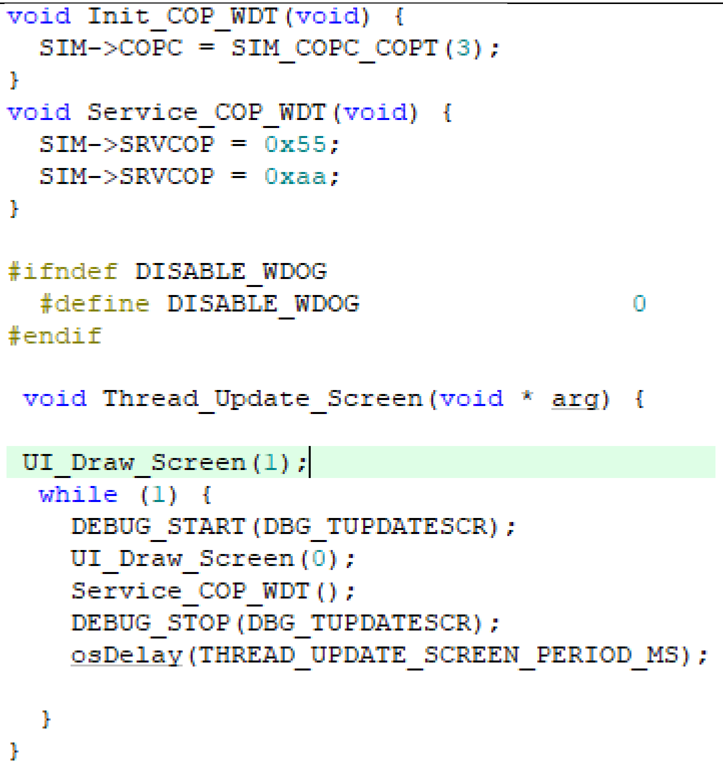
The fault calls the function **Fault_Recursion_Test** which calls itself repeatedly until the system's stack get overflowed. This causes the system to cease working.

Fault Management Approach

See Fault Test 1.

Evaluation of Effectiveness

See Fault Test 1.



```
void Init_COP_WDT(void) {
  SIM->COPC = SIM_COPC_COPT(3);
}
void Service_COP_WDT(void) {
  SIM->SRVCOP = 0x55;
  SIM->SRVCOP = 0xaa;
}

#ifndef DISABLE_WDOG
  #define DISABLE_WDOG                    0
#endif

 void Thread_Update_Screen(void * arg) {

 UI_Draw_Screen(1);
  while (1) {
    DEBUG_START(DBG_TUPDATESCR);
    UI_Draw_Screen(0);
    Service_COP_WDT();
    DEBUG_STOP(DBG_TUPDATESCR);
    osDelay(THREAD_UPDATE_SCREEN_PERIOD_MS);

  }
}
```

Figure 7.3. Key source code additions.

FAULT TEST 8 - TR_DISABLE_PERIPHCLOCKS

Fault Analysis

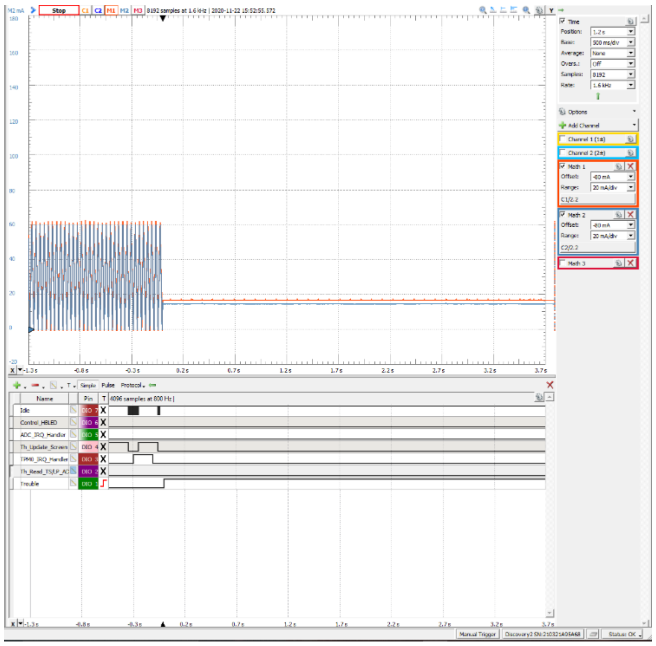


Figure 8.1. The system stops functioning after the fault is injected into the system.

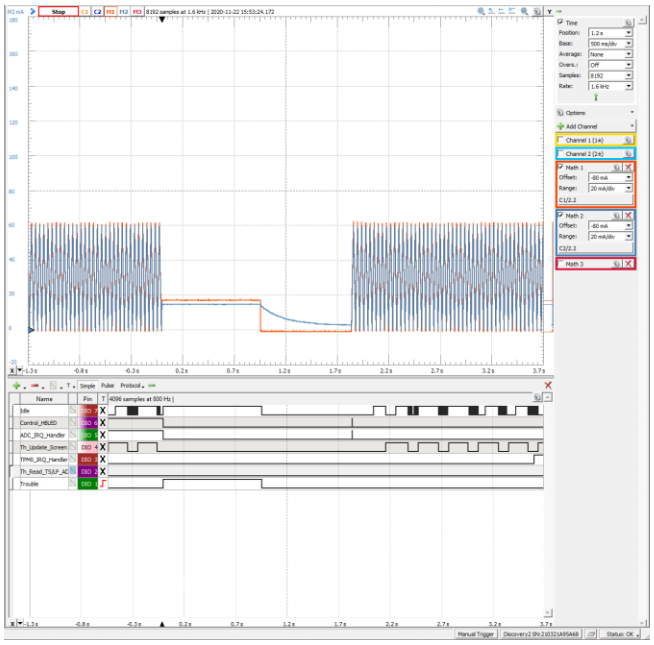


Figure 8.2. Fault detection and response code resets the system in a working state.

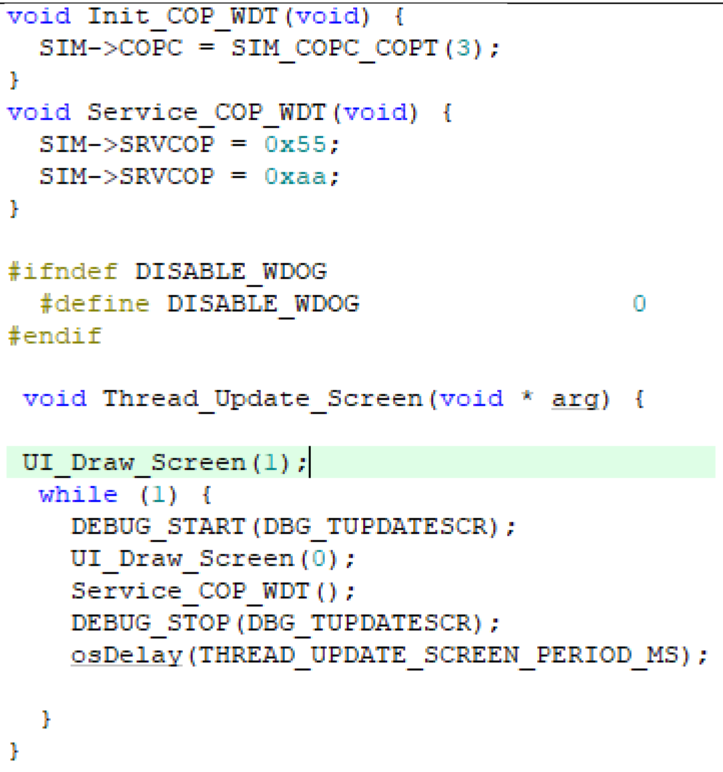
The fault changes the **SIM->SCGC6** variable to 0. This disables the clock source of the peripherals used in the system, causing the system to stop working.

Fault Management Approach

See Fault Test 1.

Evaluation of Effectiveness

See Fault Test 1.



```
void Init_COP_WDT(void) {
  SIM->COPC = SIM_COPC_COPT(3);
}
void Service_COP_WDT(void) {
  SIM->SRVCOP = 0x55;
  SIM->SRVCOP = 0xaa;
}

#ifndef DISABLE_WDOG
  #define DISABLE_WDOG                  0
#endif

 void Thread_Update_Screen(void * arg) {

 UI_Draw_Screen(1);
  while (1) {
    DEBUG_START(DBG_TUPDATESCR);
    UI_Draw_Screen(0);
    Service_COP_WDT();
    DEBUG_STOP(DBG_TUPDATESCR);
    osDelay(THREAD_UPDATE_SCREEN_PERIOD_MS);

  }
}
```

Figure 8.3. Key source code additions.

FAULT TEST 9 - TR_OSKERNELLOCK

Fault Analysis

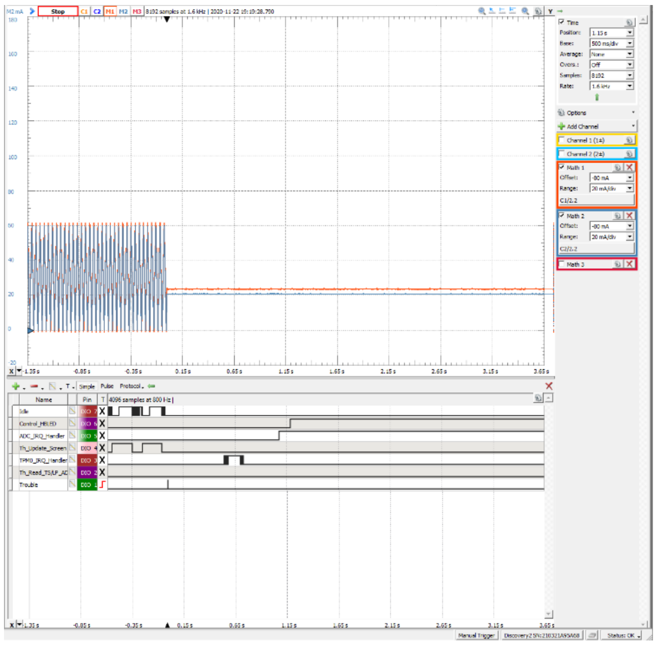


Figure 9.1. The system stops functioning after the fault is injected into the system.

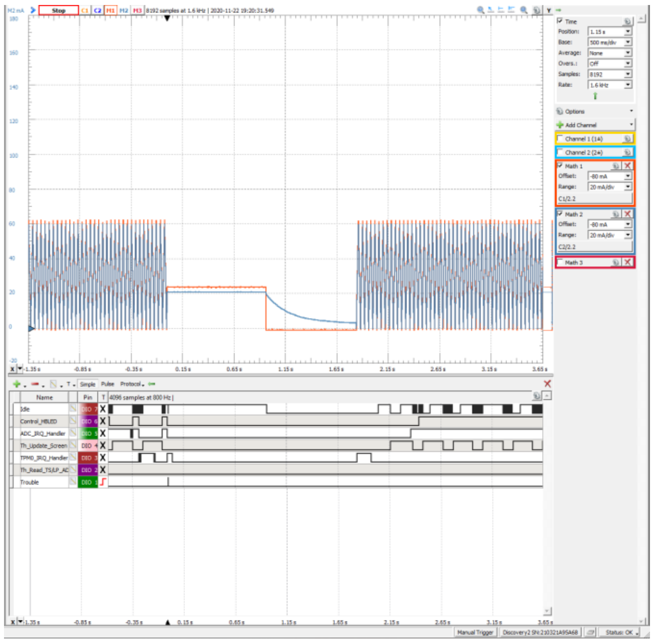


Figure 9.2. Fault detection and response code resets the system in a working state.

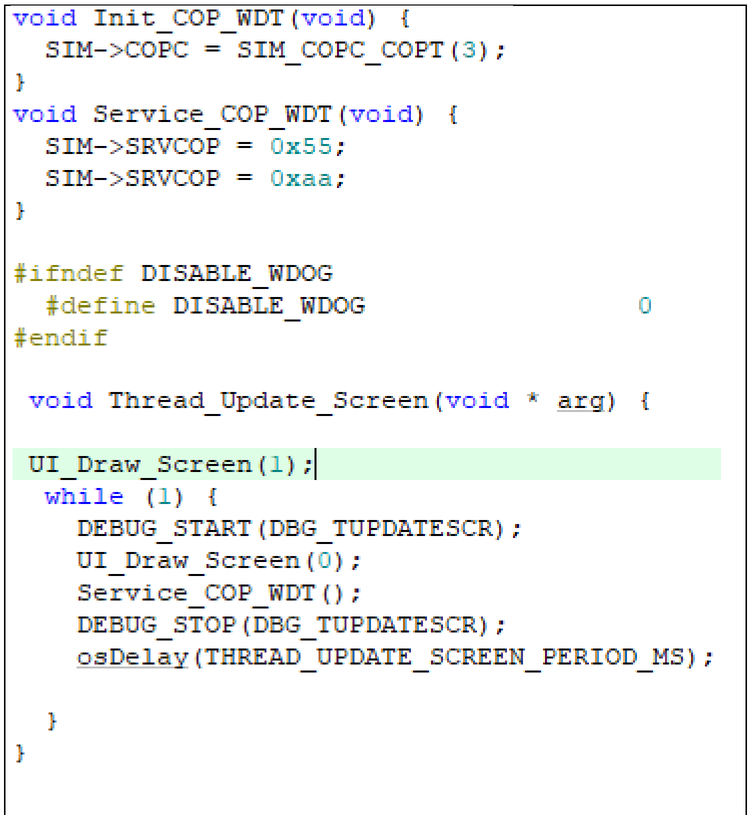
The fault calls the function **osKernelLock** which locks all task switches. This causes the system to stop working.

Fault Management Approach

See Fault Test 1.

Evaluation of Effectiveness

See Fault Test 1.



```
void Init_COP_WDT(void) {
  SIM->COPC = SIM_COPC_COPT(3);
}
void Service_COP_WDT(void) {
  SIM->SRVCOP = 0x55;
  SIM->SRVCOP = 0xaa;
}

#ifndef DISABLE_WDOG
  #define DISABLE_WDOG                0
#endif

 void Thread_Update_Screen(void * arg) {

 UI_Draw_Screen(1);
  while (1) {
    DEBUG_START(DBG_TUPDATESCR);
    UI_Draw_Screen(0);
    Service_COP_WDT();
    DEBUG_STOP(DBG_TUPDATESCR);
    osDelay(THREAD_UPDATE_SCREEN_PERIOD_MS);

  }
}
```

Figure 9.3. Key source code additions.

FAULT TEST 10 - TR_HIGH_PRIORITY_THREAD

Fault Analysis

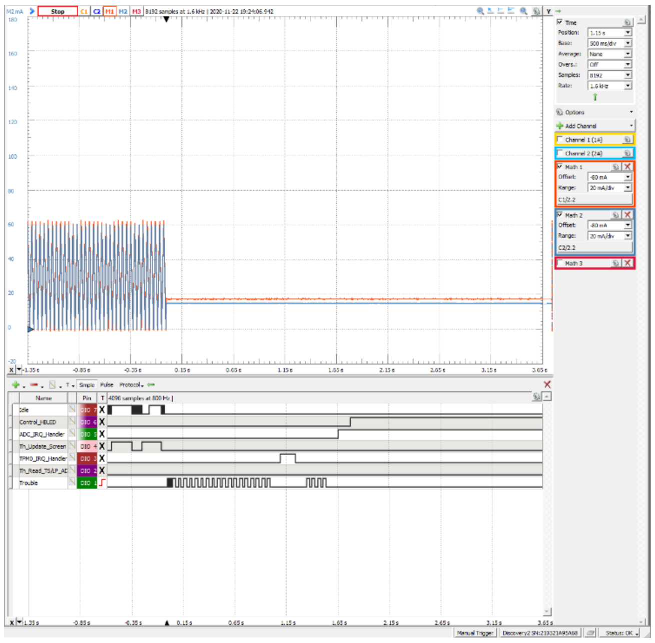


Figure 10.1. The system stops functioning after the fault is injected into the system.

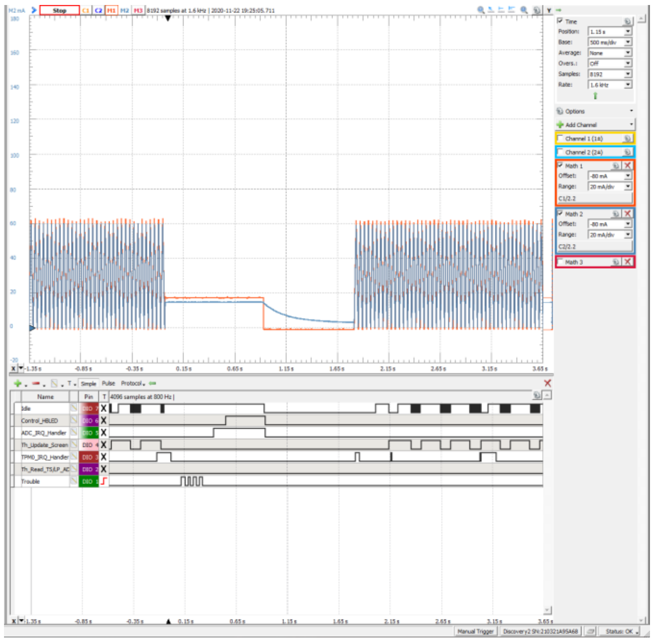


Figure 10.2. Fault detection and response code resets the system in a working state.

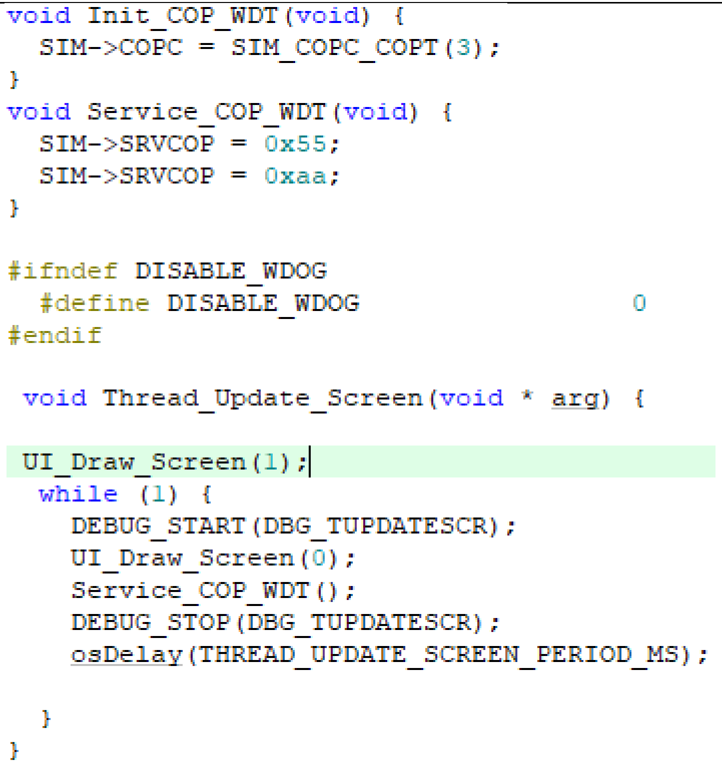
The fault changes the priority of **Thread_Fault_Injector** to **osPriorityRealtime**. This makes the thread higher priority than the other threads in the system. The fault also begins a while loop which does not end. Since no other threads can preempt the thread, it runs forever. This causes the system to stop working.

Fault Management Approach

See Fault Test 1.

Evaluation of Effectiveness

See Fault Test 1.



```
void Init_COP_WDT(void) {
  SIM->COPC = SIM_COPC_COPT(3);
}
void Service_COP_WDT(void) {
  SIM->SRVCOP = 0x55;
  SIM->SRVCOP = 0xaa;
}

#ifndef DISABLE_WDOG
  #define DISABLE_WDOG                    0
#endif

 void Thread_Update_Screen(void * arg) {

 UI_Draw_Screen(1);
  while (1) {
    DEBUG_START(DBG_TUPDATESCR);
    UI_Draw_Screen(0);
    Service_COP_WDT();
    DEBUG_STOP(DBG_TUPDATESCR);
    osDelay(THREAD_UPDATE_SCREEN_PERIOD_MS);

  }
}
```

Figure 10.3. Key source code additions.

RETROSPECTIVE

- During this project, I learned that reliable embedded systems should feature ways to deal with unexpected occurrences which could destroy the system’s functionality. Specifically, I learned how valuable the watchdog timer is in making a robust embedded system because of its versatility in detecting faults which disrupt critical processes. In the future, I plan on implementing watchdog timers and other failure mechanisms in my designs to ensure that issues can be dealt with.