ECE 460: EMB. SYS. ARCHITECTURES
PROJECT 2: SHIELDS UP!

Yuvini Velasquez

yhvelasq

INTRODUCTION

In creation and design of a robust Embedded System, there are many errors and faults that need to be prevented and managed. These can be internal or external faults such as exploit attacks, device failures, bad programming practices, or even change of bits because of solar flare. To manage and prevent a few of these cases, it was necessary to modify the code in Project 2.

To be able to solve any of these errors, there was a need to understand the architecture of the MCU and the implemented code. This allowed to have a better debugging approach.

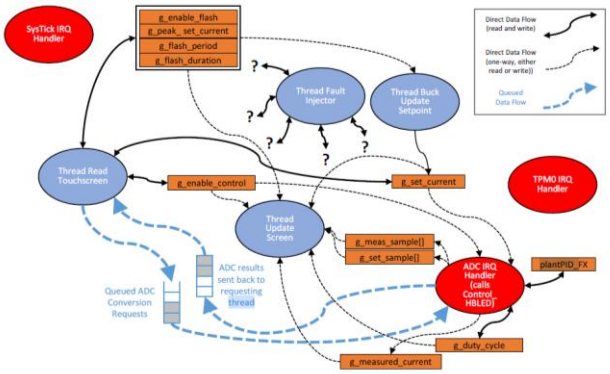


Figure 1. Software architecture for Project 2 program.

Some strategies to manage these faults included creating a system that stores and verifies previous values, adding maximum and minimum value verifiers to keep the threshold, storing and verifying configuration periodically, and adding a mechanism to restart the system such as a Watchdog timer. To view the behavior of the board, I used an oscilloscope by using the Analog Discovery 2 which allowed to view the enable and disable of some pins and the current through the LED. I also used the Keil uVision IDE and the debugging tools available for this platform where I could watch the Stack and having variables in the Watch List.

FAULT SETPOINT_HIGH

Fault Analysis

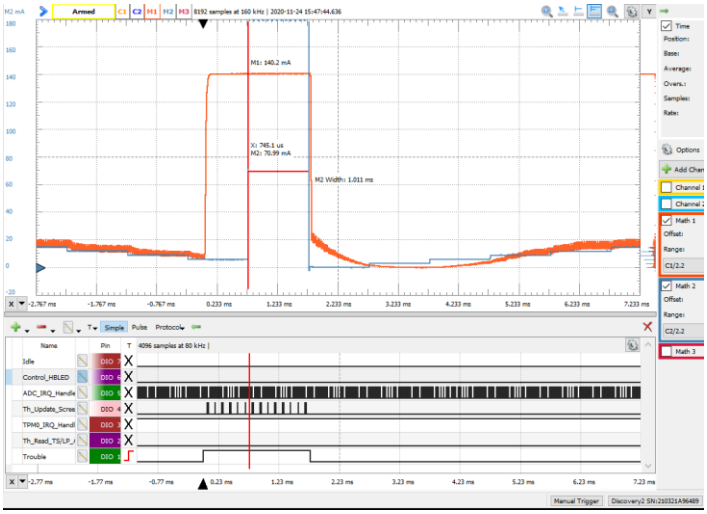


Figure 0.1. Current goes unexpectedly high, there is no verification process where the set current is verified with its previous value, leaving room for error.

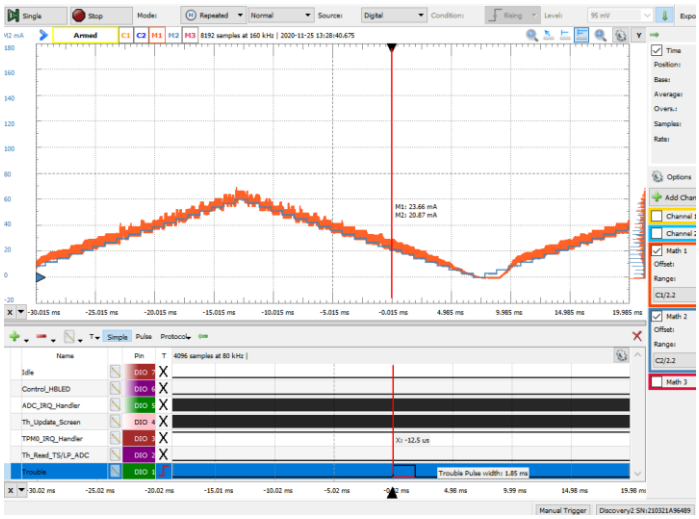


Figure 0.2. Code implemented to review and correct unexpected current changes by comparing them to a previous recorded current.

The fault set_point_high manually changes the current setpoint which make the current go very high. On Figure 0.1 we can see how the current goes high for 1.745 ms. This simulates a case when the current is set to high by an unexpected event. After some current comparison in the Control_HBLED we can correct an unexpected high value.

Fault Management Approach

To be able to solve this problem, I first needed to figure out the function that is updating the **g_set_current** value. This function happened to be **Update_Set_Current**. In here I created a copy of the set current called **g_set_current_copy** nonvolatile so it won't be modified unexpectedly. This will help in the comparison of the **g_set_current** with the saved copy. The function **Update_Set_Current** is called by the **Thread_Buck_Update_Setpoint** thread.

Thread_Buck_Update_Setpoint is updated every four times the **ADC0_IRQHandler** runs. So, it is smarter to add verification of the code in **ADC0_IRQHandler**, but specifically in **Control_HBLED.** This allows to find an unexpected current change faster which reduces the time where another handler can set the incorrect current.

Additionally, a backup check is added in Update_Set_Current function where there is a comparison between the current **g_set_current** and the recorded current from last function call named **g_set_current_copy** to verify if the current is the value previously recorded. If that is not the case, then the **g_set_current_copy** recorded value is copied instead of the **g_set_current** value.

```
volatile int g_set_current=0;          // Default starting LED current
int g_set_current_copy = 0;            // Copy of set current
void Update_Set_Current(void) {
  // Ramp curent up and down
  static volatile int t=0;
  if (g_enable_flash){
    t++;
    if(g_set_current != g_set_current_copy){
    if (t <= g_flash_duration/2) {
    if(g_peak_set_current < g_set_current){
    Set_DAC_mA(g_set_current);
    if (t >= g_flash_period)
      t = 0;
    g_set_current_copy = g_set_current; //Saving current in period
  }
}
void Control_HBLED(void) {
    uint16_t res;
    FX16_16 change_FX, error_FX;
    static int curr_sample=0;
    static uint16_t prev_set_current_sample=0;
    int buff_size, threshold;

    DEBUG_START(DBG_CONTROLLER);

  //verification if current has changed since last set
      if(g_set_current != g_set_current_copy){
        g_set_current = g_set_current_copy;
      }
```

Evaluation of Effectiveness

As shown in **Error! Reference source not found.** this process eliminates non expected current values and there isn't a spike in any of the currents. Other solutions were first tested but there was still a spike in the current. There is no visible signal on the oscilloscope of the high current output which shows that there is a good timing response for this solution. This is even with an oscilloscope visualization of 15uS per division.

FAULT SETPOINT_LOW

Fault Analysis

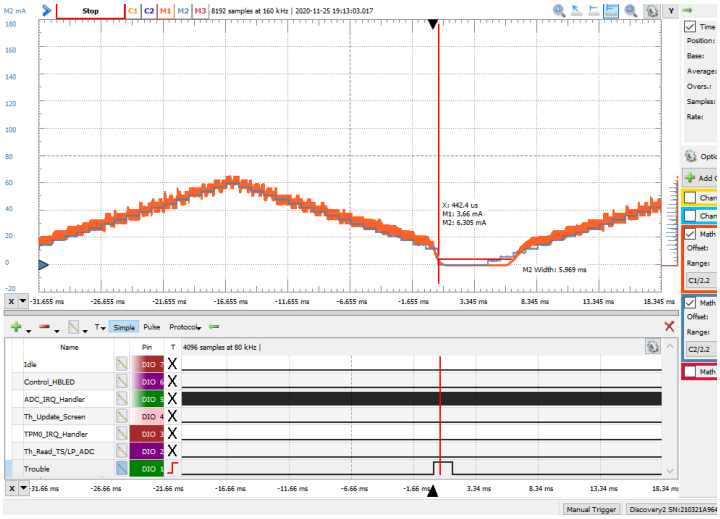


Figure 1.1. Current goes unexpectedly low, there is no verification process where the set current is verified with its previous value, leaving room for error.

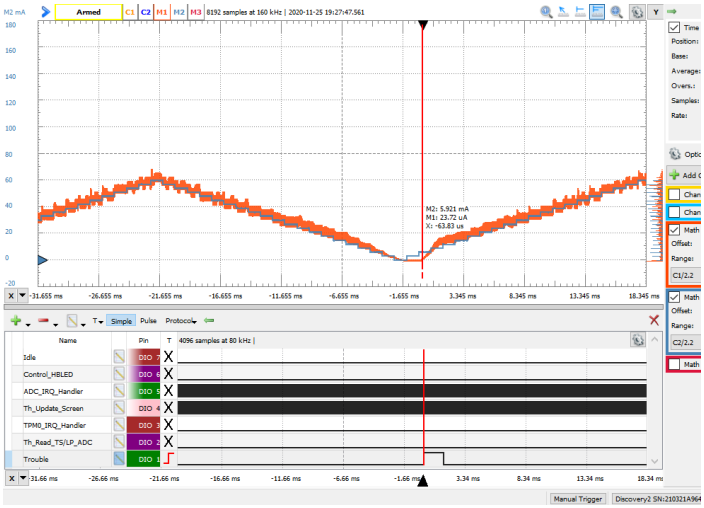


Figure 1.2. Fault detection and response code corrected where currents get verified and corrected if an unexpected value is seen.

The fault set_point_low manually changes the current setpoint which makes the current go very low. On Figure 1.1 we can see how the current goes low for 5.969 ms. This simulates a case when the current is set to low by an unexpected event. After some current comparison in the Control_HBLED we can correct an unexpected low value.

Note: The solution implemented for Fault 0 also corrected Fault 1. Which shows a solid solution for unexpected current problems.

Fault Management Approach

To be able to solve this problem, I first needed to figure out the function that is updating the **g_set_current** value. This function happened to be **Update_Set_Current**. In here I created a copy of the set current called **g_set_current_copy** nonvolatile so it won't be modified unexpectedly. This will help in the comparison of the **g_set_current** with the saved copy. The function **Update_Set_Current** is called by the **Thread_Buck_Update_Setpoint** thread.

Thread_Buck_Update_Setpoint is updated every four times the **ADC0_IRQHandler** runs. So, it is smarter to add verification of the code in **ADC0_IRQHandler**, but specifically in **Control_HBLED.** This allows to find an unexpected current change faster which reduces the time where another handler can set the incorrect current.

Additionally, a backup check is added in Update_Set_Current function where there is a comparison between the current **g_set_current** and the recorded current from the last function call named **g_set_current_copy** to verify if the current is the value previously recorded. If that is not the case, then the **g_set_current_copy** recorded value is copied instead of the **g_set_current** value.

```
volatile int g_set_current=0;          // Default starting LED current
int g_set_current_copy = 0;            // Copy of set current
void Update_Set_Current(void) {
  // Ramp curent up and down
  static volatile int t=0;
  if (g_enable_flash){
    t++;
    if(g_set_current != g_set_current_copy){
    if (t <= g_flash_duration/2) {
    if(g_peak_set_current < g_set_current){
    Set_DAC_mA(g_set_current);
    if (t >= g_flash_period)
      t = 0;
    g_set_current_copy = g_set_current; //Saving current in period
  }
}
void Control_HBLED(void) {
    uint16_t res;
    FX16_16 change_FX, error_FX;
    static int curr_sample=0;
    static uint16_t prev_set_current_sample=0;
    int buff_size, threshold;

    DEBUG_START(DBG_CONTROLLER);

  //verification if current has changed since last set
      if(g_set_current != g_set_current_copy){
        g_set_current = g_set_current_copy;
      }
```

Evaluation of Effectiveness

As shown in **Error! Reference source not found.** this process eliminates non expected current values and there isn't a spike in any of the currents. Other solutions were first tested but there was still a spike in the current. There is no visible signal on the oscilloscope of the high current output which shows that there is a good timing response for this solution. This is even with an oscilloscope visualization of 15uS per division.

FAULT PID_FX_GAINS

Fault Analysis

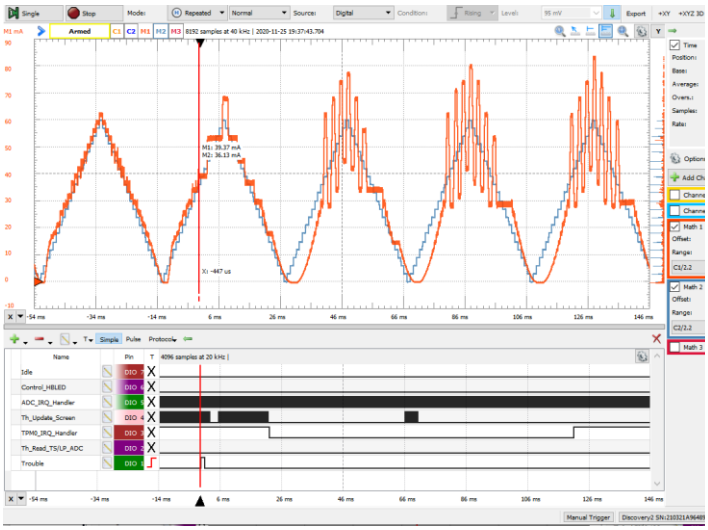


Figure 2.1. After the present fault, the gain of the current is off.

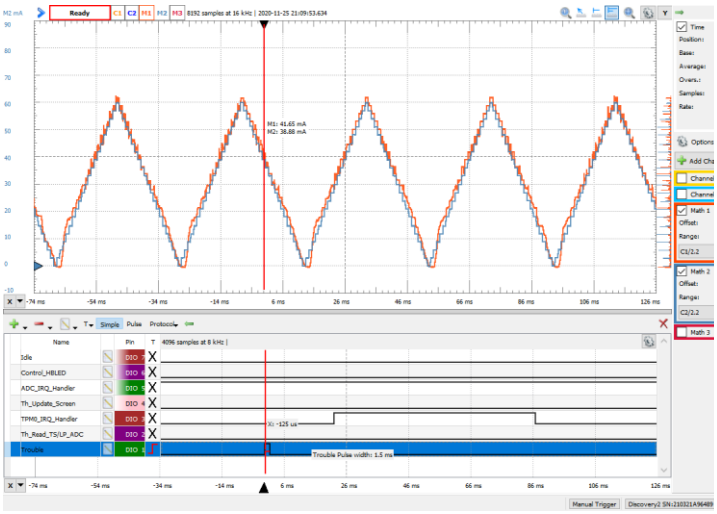


Figure 2.2 Implemented a verification method that compares a previous PID value used in the last setup with the current value.

The fault PID_FX_GAIN changes the current's gain. On Figure 2.1 we can see how the current starts incrementally changing its gain until the pattern is not as expected. This simulates a case when the **SPid**, which contains proportional gain, integral gain, and derivative gain, changes by an unexpected event.

Fault Management Approach

To be able to solve this problem, I first needed to figure out where the values for gain are stored. I found out that values are stored in a struct type **SPIDFX** instantiated as **plantPID_FX** which contains the proportional gain, integral gain, and derivative gain among other values. These values are used in the function **UpdatePID_FX** to calculate the proportional, integral, and derivative terms with respect with the **error_FX** which is the now current error of this term.

A verification statement has been set in UpdatePID_FX in case **plantPID_FX** is unexpectedly modified. First, we needed to have some variables that will store the previous set up **plantPID_FX** value after the values have been used in **UpdatePID_FX**. These values are stored at the end of the **UpdatePID_FX** function to be read in the next call function. When the program runs for the first time, these values used to compare are set to zero, therefore we need to set up a first-time update. This was done using an **IF** statement with three **OR** statements which check if values are not zero. If values are zero, then the function runs as it's supposed to for the first run, otherwise it verifies if the values stored are equal to the ones currently set in **plantPID_FX**. If not, then the stored values are given to **plantPID_FX**.

```
FX16_16 pGain_Store = 0;  //Storing proportional gain as reference
FX16_16 iGain_Store = 0;  //Storing integral gain as reference
FX16_16 dGain_Store = 0;  //Storing derivative gain as reference
FX16_16 UpdatePID_FX(SPidFX * pid, FX16_16 error_FX, FX16_16 position_FX){

  FX16_16 pTerm, dTerm, iTerm, diff, ret_val;
if(pGain_Store ||iGain_Store ||dGain_Store){
    if(pGain_Store != pid->pGain){
      pid->pGain = pGain_Store;
    }
    if(iGain_Store != pid->iGain){
      pid->iGain = iGain_Store;
    }
    if(dGain_Store != pid->dGain){
      pid->dGain = dGain_Store;
    }
  }

  ret_val = Add_FX(pTerm, iTerm);
  ret_val = Subtract_FX(ret_val, dTerm);
  pGain_Store = pid->pGain;
  iGain_Store = pid->iGain;
  dGain_Store = pid->dGain;
  return ret_val;
}
```

This modification is implemented every time **Control_HBLED** is called and control mode is set to **PID_FX**. In **PID_FX** the function **UpdatePID_FX** is called and the verification that was implemented is used.

Evaluation of Effectiveness

As shown in **Error! Reference source not found.** this process eliminates any unwanted change in the gain of current. This solution does not require a correction time as seen in the orange current. There is no visible signal on the oscilloscope of where the problem is, which shows that there is a good timing response for this solution. This is even with an oscilloscope visualization of 20mS per division.

FAULT LCD_MUTEX

Fault Analysis

Video link4.1: *https://youtu.be/-7olpZ1rwAU*

Getting into infinite loop because osMutexAdquire receives LCD_mutex where it gets stuck.

Video link 4.2: *https://youtu.be/ak2Ju7lhu5I*

Implemented a Watchdog timer code that detects the osMutexAdquire and MCU if they are refreshing correclty and if not it will reset it.

The fault **TR_LCD_mutex** acquires the **LDC_mutex** and does not return it. On Video link 1 we can see how the LCD gets stuck. This simulates a case when the mutex is stuck in an infinite loop.

Fault Management Approach

To be able to solve this problem, I first needed to figure out in which mutex the code will get stuck. I noticed that it was the LCD that was stuck, and I also analyzed the fault code where it was stuck in. A WatchDog timer was implemented to avoid getting the MCU stuck at the LCD mutex.

First, I included the code for a Watchdog timer in the program files called **COPT_WDT.c** and **COP_WDT.h** where the initialization and service of the WatchDog timer is located. After this, I disabled the function that by default disables the WatchDog by defining **DISABLE_WDOG** to 0.

To service the WatchDog timer, I found the function that was called the most by the LCD process by adding breakpoints to the functions used in the LCD. With this process I found that the function **LCD_Text_PrintChar** gets called more times than other LCD functions and I decided to service the timer there.

Evaluation of Effectiveness

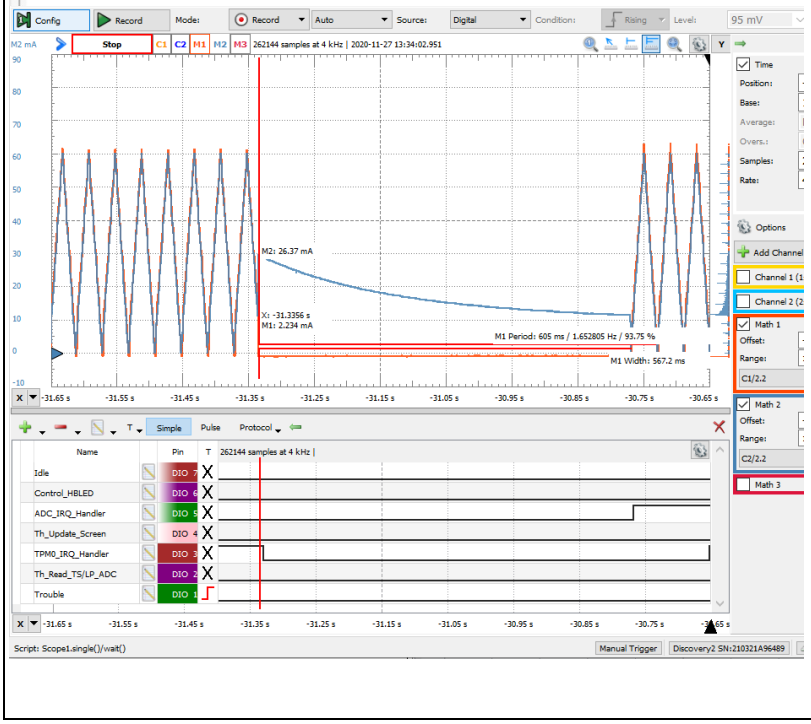
To visualize the reset, I set the oscilloscope which recorded the reset process. From the time the infinite loop starts to the reset there is a 605 ms time. This solution may reset and the MCU to which may cause the loss of current processes, but it is a great solution since most likely the OS was in a non-recoverable state.

```
#ifndef COP_WDT_H
#define COP_WDT_H

void Init_COP_WDT(void);
void Service_COP_WDT(void);
void Flash_Reset_Cause(void);

#endif // COP_WDT_H
```

```
void Init_COP_WDT(void) {
  // Select 1 kHz clock and 1024 cycle time-out
  SIM->COPC = SIM_COPC_COPT(3) & ~SIM_COPC_COPCLKS_MASK & ~SIM_COPC_COPW_MASK;
  }

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

```
#define DISABLE_WDOG 0
void SystemInit (void) {
#if (DISABLE_WDOG)
  /* SIM_COPC: COPT=0,COPCLKS=0,COPW=0 */
  SIM->COPC = (uint32_t)0x00u;
```

```
void LCD_Text_PrintChar(PT_T * pos, char ch) {
  Service_COP_WDT();
  uint8_t glyph_index_entry;
  const uint8_t * glyph_data; // start of the data
#if BITS_PER_PIXEL == 1        // Copy bitmap byte directly
  PT_T cur_pos;
#endif
```



FAULT DISABEL_ALL_IRQS

Fault Analysis

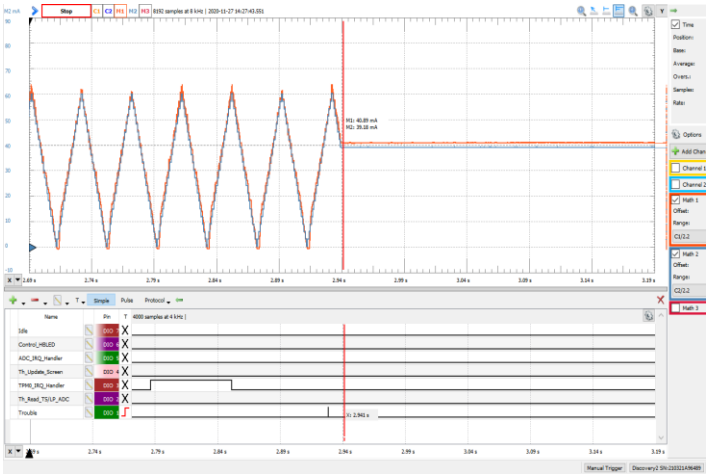


Figure 5.1. OS is stuck and IRQ is disabled.

Video 5.1: https://youtu.be/8j8-LPlmE9M

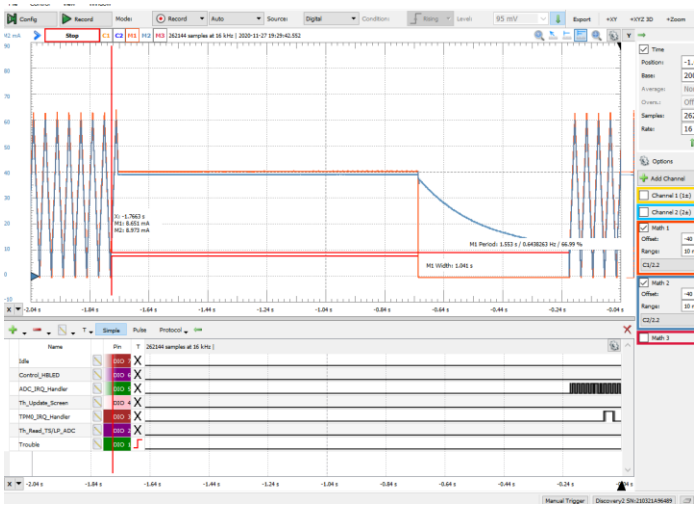


Figure 5.2. Implemented a WatchDog timer which recets the MCU if there are unwanted cases.

Video 5.2: https://youtu.be/KtlLumXY75I

The **Disable_All_IRQ** disables the interrupts. This creates an almost unrecoverable state. This simulates a case when all ISRs are disabled creating an infinite loop in a high-priority ISR.

Fault Management Approach

To be able to solve this problem, I first tried to create a method where I could store the current clock after all of the peripherals were set up and then recover them. The only problem is that as soon as the clocks were all disabled, it was hard to go to a function to verify if clocks were still implemented. Therefore, having a WatchDog timer was the best option.

I reused the WDT setup from the previous fault solution. First, I included the code for a WatchDog timer in the program in the files called **COPT_WDT.c** and **COP_WDT.h** where the initialization and service of the WatchDog timer is located. After this, I disabled the function that by default disables the WatchDog by defining **DISABLE_WDOG** to 0.

To service the WatchDog timer, I decided to service the timer in **main** function because this function has a wider coverage verses going to every specific function that sets up the clock for each handler.

```
#ifndef COP_WDT_H
#define COP_WDT_H

void Init_COP_WDT(void);
void Service_COP_WDT(void);
void Flash_Reset_Cause(void);

#endif // COP_WDT_H
void Init_COP_WDT(void) {
   // Select 1 kHz clock and 1024 cycle time-out
   SIM->COPC = SIM_COPC_COPT(3) & ~SIM_COPC_COPCLKS_MASK & ~SIM_COPC_COPW_MASK;
   }

void Service_COP_WDT(void) {
   SIM->SRVCOP = 0x55;
   SIM->SRVCOP = 0xaa;
}
#define DISABLE_WDOG 0
void SystemInit (void) {
#if (DISABLE_WDOG)
   /* SIM_COPC: COPT=0,COPCLKS=0,COPW=0 */
   SIM->COPC = (uint32_t)0x00u;
int main (void) {
   Init_COP_WDT(); //
   Init_Debug_Signals();
   Init_RGB_LEDs();
   Control_RGB_LEDs(0,0,1);
   Service_COP_WDT();  //
   Sound_Disable_Amp();
   LCD_Init();
   LCD_Text_Init(1);
   LCD_Erase();

   LCD_Erase();
   LCD_Text_PrintStr_RC(0,0, "Test Code");

   Service_COP_WDT();  //
#if 0
   // LCD_TS_Calibrate();
   LCD_TS_Test();
#endif
   Delay(70);
   Service_COP_WDT();  //
   LCD_Erase();
   Init_Buck_HBLED();
   Save_ADC_Interrupt_Setup();
   osKernelInitialize();
   Fault_Init();
   Create_OS_Objects();
   osKernelStart();

   Service_COP_WDT();  //
}
```

Evaluation of Effectiveness

As shown in **Error! Reference source not found.** the WatchDog timer resets in approximately 1.04 seconds and the system restarts in 512 ms creating a total recovery time of 1.55 seconds for full recovery. Depending on the type of Embedded system, this may not be the best time for a system to recover.

FAULT TR_DISABLE_ADC_IRQ

Fault Analysis

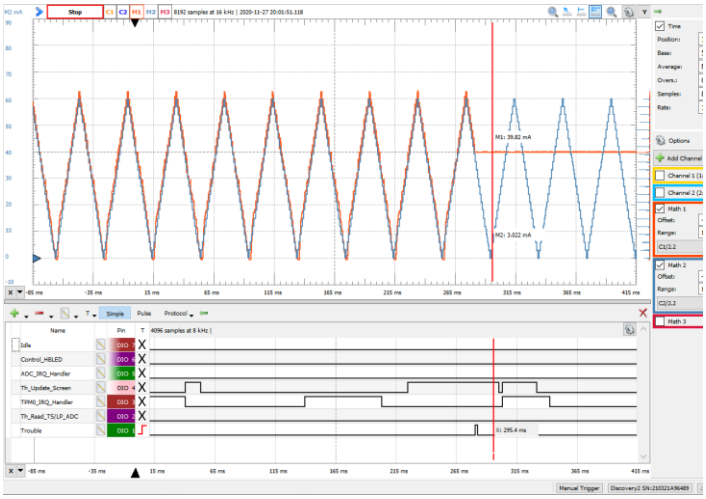


Figure 6.1. ADC interrupt is disabled preventing the new reading of current.

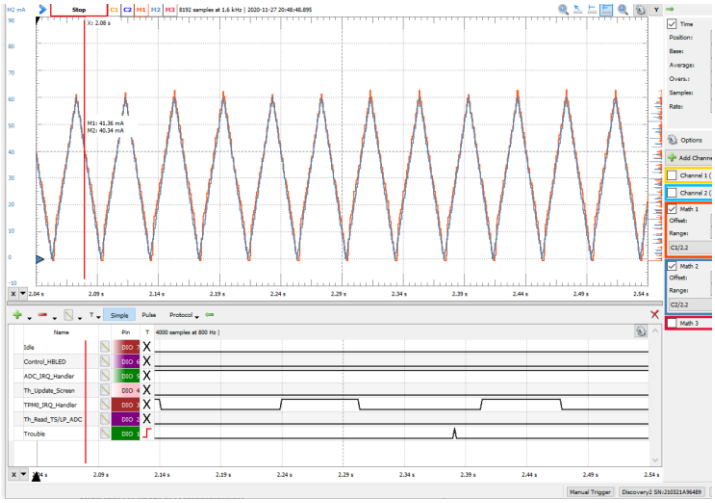


Figure 6.2 Implemented a setup and verification method for the IRQ setup.

The ADC IRQ handler gets disabled stopping the conversion of the current which prevents further readings. On Figure 6.1 we can see how the current reading stops when this event happens. This simulates a case when one of the handlers (in this case the ADC) gets disabled.

Fault Management Approach

To be able to solve this problem, I first needed to figure out which function disables an IRQ and then see if there was a pre-defined function that enabled this IRQ. I found that the function that disables IRQs is **NVIC_DisableIRQ(ADC0_IRQn);** and the **ADC0_IRQn** is the actual IRQ that needs to be disabled.

To create a setup and verification of the IRQ I created two functions. One was in charge of saving the setup of the ADC IRQ handler. This function was implemented in **main** after all the IRQ's were initialized. This function copies the IRQ setup in **ICER_store.** After this, a flag bit named **ADC_store_flag** will be raised that will help within the verification of the IRQ. This verification function runs in the **Thread_Buck_Update_Setpoint** which can be used as a setpoint for verification.

If there is a change in the IRQ, which is not specified or wanted in this design, then the IRQ for the ADC will be enabled again since that is what we are aiming for.

This modification is implemented every time **Control_HBLED** is called and control mode is set to **PID_FX**. In **PID_FX** the function **UpdatePID_FX** is called and the verification that was implemented is used.

```
uint32_t ICER_store;
int ADC_store_flag = 0;

void Save_ADC_Interrupt_Setup(void){
  ICER_store = NVIC->ICER[0U];
  ADC_store_flag =1;
}

void ADC_Setup_Verification(void){
  if (ADC_store_flag == 1){
    if(ICER_store != NVIC->ICER[0U]){
    NVIC_EnableIRQ(ADC0_IRQn);
    }
  }
}

int main (void) {
  //Init_COP_WDT(); //WDT
  Init_Debug_Signals();
  Init_RGB_LEDs();
  Control_RGB_LEDs(0,0,1);
  //Service_COP_WDT();  //
  Sound_Disable_Amp();
  LCD_Init();
  LCD_Text_Init(1);
  LCD_Erase();

  LCD_Erase();
  LCD_Text_PrintStr_RC(0,0, "Test Code");

  //Service_COP_WDT();  //
#if 0
  Delay(70);
  //Service_COP_WDT();  //
  LCD_Erase();
  Init_Buck_HBLED();
  Save_ADC_Interrupt_Setup();

void Thread_Buck_Update_Setpoint(void * arg) {
  while (1) {
    ADC_Setup_Verification();
    osDelay(THREAD_BUS_PERIOD_MS);
    Update_Set_Current();
  }
}
```

NVIC->ICER[0U]	0x00028000
ICER_store	0x00028000

Evaluation of Effectiveness

As shown in **Error! Reference source not found.** this solution eliminates any unwanted disable of the ADC IRQ. This solution only works for the ADC IRQ and will not for other handlers. The solution is instant with no delay and without the current stuck at any specific value.

RETROSPECTIVE

- This project was very engaging and fun. In this project I was able to use all the concepts learned throughout the semester. The only different thing that I would do is to complete all the faults available to hone my programmer and debugger skills.
- This time I did not have any hardware, drivers, or debugger problems. One thing that took some time is getting the proper scale and frame to view the signals in the Analog Discovery tool.
- Concerning the developing process, I would change and use more of the tools I have available. I believe I did not use the memory viewer but that may be because of the types of faults that I solved.
- The only way I may change this project is by having one problem with a guided debugging process. In other words, have steps like: What is the error behavior and what do the tools tell me? What is the error affecting? What are the best options to solve them? Implement solution, fix, and validate.