ECE 560: EMB. SYS. ARCHITECTURES

PROJECT 2: SHIELDS UP

Arpad Voros

aavoros

INTRODUCTION

The purpose of this project is to alter the current software architecture of a given blinking high-brightness LED (HBLED) system so that various faults are properly handled to keep the system running smoothly without breaking. The current of the HBLED is controlled in a linearly decreasing & increasing fashion, producing a sawtooth wave. An oscilloscope and logic analyzer (AD2) is used to observe the analog signal that is the set & measured current of the HBLED, as well as various debug signals coming from the MCU. Faults are injected into the system using a routine which changes the values of vital control variables, acquires a mutex to block a thread, causes stack overflow via an infinite recursive call, and more. The program will continue to run where, in most cases, the program never self-corrects. To prevent this, the faults must be properly handled by implemented self-correcting aspects & fail-safes within the current software architecture.

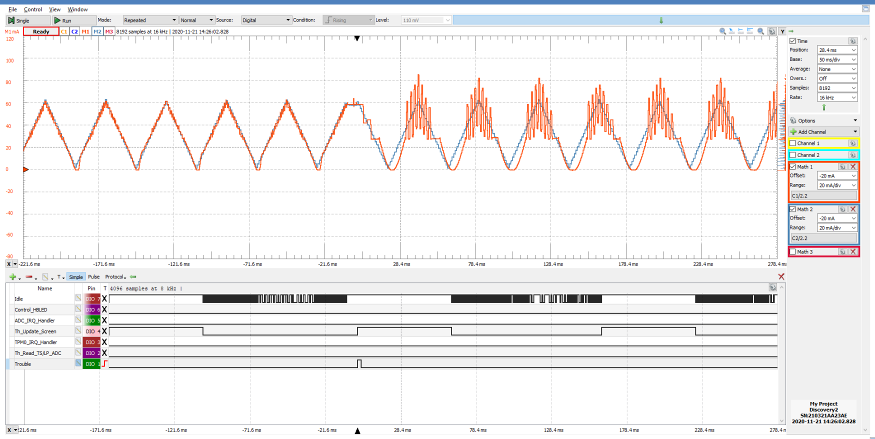
Each fault presented within this report is exactly 1 page long, accompanied by a description of the fault, before/after oscilloscope figures with descriptions, listings of added code, the approach taken to handle the fault, and an evaluation on its effectiveness.

ADDITIONAL NOTES

Some of the figures from the AD2 are difficult to see, due to the requirement of having 1 page per fault. Faults 0 through 6 have smaller & less legible scope figures than Faults 7 through 12, therefore an appendix is included to display larger versions of the former.

During the time of exporting these figures, either there was a mistake with some DIO connections or I was sampling at a low rate, but the digital channels for **Control_HBLED** and **ADC_IRQ_Handler** (and sometimes others) appear not to display at the proper high frequency they should be. Nevertheless, these two channels aren't used frequently in explanation of handling faults so it should not impact the quality of the report.

FAULT 0 – PID_FX

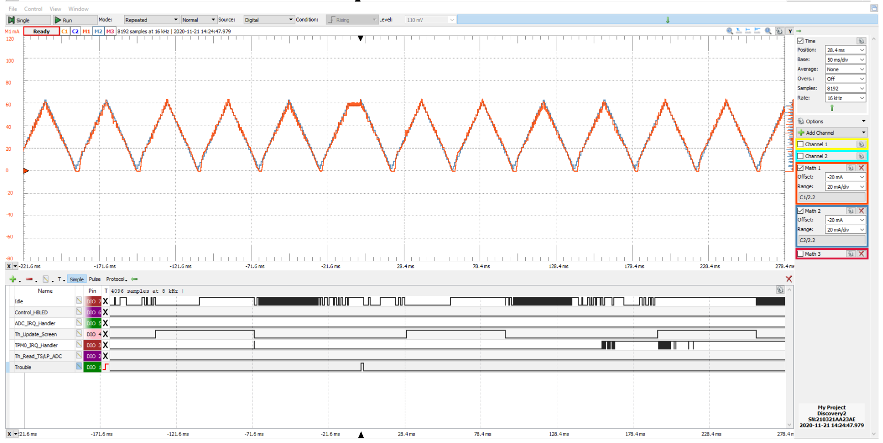


🡐 Figure 0.1. LED current does not follow setpoint after fault disables controller

```
// reset the gains to their original values
void reset_PID_FX(SPidFX * pid) {
  pid->pGain = plantPID_FX_setter.pGain; // pGain
  pid->iGain = plantPID_FX_setter.iGain; // iGain
  pid->dGain = plantPID_FX_setter.dGain;  // dGain
}

// hard set new PID values
void set_PID_FX(FX16_16 pTerm, FX16_16 iTerm, FX16_16 dTerm) {
  plantPID_FX_setter.pGain = pTerm; // pGain
  plantPID_FX_setter.iGain = iTerm; // iGain
  plantPID_FX_setter.dGain = dTerm;  // dGain
}
```

*Figure 0.2. Two new functions added to **control.c**, **control.h***



🡐 Figure 0.3. Fault detection and response code corrects variable and keeps controller enabled without much change in system behavior

```
  // make a copy of the previous one during update
  plantPID_FX_setter = *(pid);
```

*Figure 0.4. make a copy of the current PID into **plantPID_FX_setter***

Fault Description

This fault changed the proportional, integral, and derivative terms for **plantPID_FX**. This caused the current to not be properly controlled, hence the incorrect current seen in Figure 0.1.

```
FX16_16 UpdatePID_FX(SPidFX * pid, FX16_16 error_FX, FX16_16 position_FX){
  FX16_16 pTerm, dTerm, iTerm, diff, ret_val;

  if (plantPID_FX_setter.pGain != pid->pGain || plantPID_FX_setter.iGain != pid->iGain || plantPID_FX_setter.dGain != pid->dGain) reset_PID_FX(pid);
```

*Figure 0.5. Check the current PID with **plantPID_FX_setter** in **UpdatePID_FX**. If wrong, reset the PID. Figure 0.4. occurs at the end of this routine.*

Fault Management Approach

To fix the fault, an extra variable of type **SPidFX** was made called **plantPID_FX_setter**, which had the same initial values as **plantPID_FX_setter** (at the top of **control.c**). The name was selected because that's what the variable was; the 'setter', as it set the value of the PID controller and acted as a backup. In essence, **plantPID_FX** was used to control the current values, and **plantPID_FX_setter** was used to control the **plantPID_FX** values.

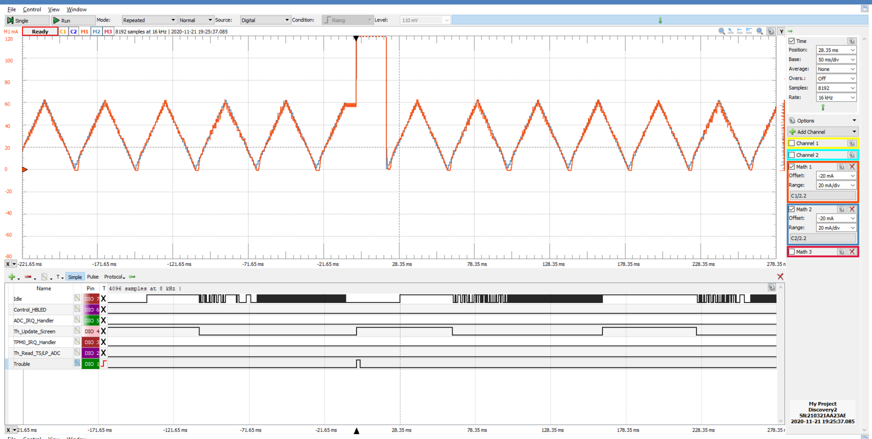
During **UpdatePID_FX**, before calculating the current value, a conditional statement (seen in Figure 0.5.) checks each value of the **plantPID_FX** and compares it to **plantPID_FX_setter**. If there are any differences, there was an improper change made to **plantPID_FX**. This calls the function **reset_PID_FX**, which takes all the PID terms from **plantPID_FX_setter** and sets them to a **SPidFX** pointer **pid**, (in this case **plantPID_FX**). The function returns, and since the PID changed the values at that address, the pointer values **pid** in the rest of the routine have changed. The function proceeds as normal, properly fixing the fault.

To give the user the option to change the PID values without having it reset, another function is made called **set_PID_FX** (see Figure 0.2.), which takes in the P, I, and D terms and sets **plantPID_FX_setter** to those values. Then next time **UpdatePID_FX** runs, **plantPID_FX** will automatically be updated to **plantPID_FX_setter** values, effectively and properly changing the PID values.

Evaluation of Effectiveness

This approach is effective in the temporal and programming sense. There is a short period of time (~8ms, seen in Figure 0.3.) where the current plateaus and fails to continue its default sawtooth pattern, since during that period is when the fault occurs and the MCU handles the fault. Programmatically, any other file which includes **control.h** has the option of calling **set_PID_FX** to properly change the PID values, if need be. If **set_PID_FX** was not an option, then the program would only allow for the initial PID values to ever be set during the course of the runtime, which is very inflexible.

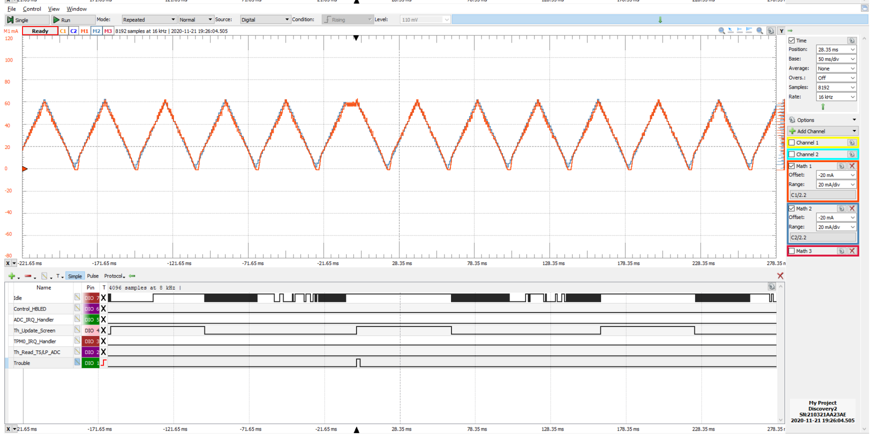
FAULT 1 – SET CURRENT HIGH



🡐 Figure 1.1. LED current does not follow setpoint after fault disables controller

```
volatile int g_set_current=0;
int g_set_current_copy=0;
```

*Figure 1.2. A copy of **g_set_current** initialized at the top of **control.c***



🡐 Figure 1.3. Fault detection and response code corrects variable and keeps controller enabled without much change in system behavior

```
// make a copy of g_set_current
g_set_current_copy = g_set_current;
```

*Figure 1.4. make a copy of **g_set_current** into **g_set_current_copy***

Fault Description

This fault changed the variable **g_set_current** to 1000, which is 1 ampere, overloading the LED, hence the incorrect current seen in Figure 1.1.

```
void Update_Set_Current(void) {
  // Ramp curent up and down
  static volatile int t=0;

  // set g_set_current to copy if updated outside of this routine
  g_set_current = (g_set_current == g_set_current_copy) ? g_set_current : g_set_current_copy;
```

*Figure 1.5. Check if **g_set_current** equals **g_set_current_copy** in **Update_Set_Current**. If not, set them equal. Figure 1.4. occurs at the end of this routine.*

Fault Management Approach

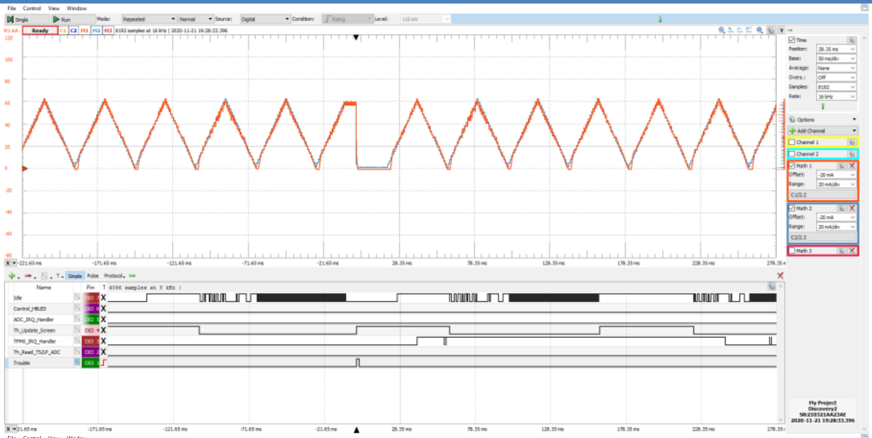
After realizing that the *only* place where **g_set_current** was being changed was in the routine **Update_Set_Current**, it was appropriate to use this function as the *only* location to evaluate whether or not **g_set_current** was changed outside of this routine. Using a similar approach seen in Fault 0, a copy is made called **g_set_current_copy** in **control.c** (Figure 1.2.) and is compared to **g_set_current** within **Update_Set_Current** (Figure 1.5.). At the end of this routine (Figure 1.3.) the copy retains the value of **g_set_current**. Therefore, next time **g_set_current** is changed outside of **Update_Set_Current**, the copy will catch this and quickly fix the value of **g_set_current**.

Evaluation of Effectiveness

This approach is effective in the temporal sense, where similar to Fault 0 there is a ~8ms period where the set current and measure current plateaus (seen in Figure 1.3.). Afterward, there is no current spike (as in Figure 1.1.), but the sawtooth pattern properly continues.

In the programming since, if the user/programmer would want the system to have a constant current then they must stop calling **Update_Set_Current** to prevent the sawtooth pattern. As a result, **g_set_current_copy** is never used, so the user is free to change **g_set_current** with no repercussions. This enables flexibility in programming while maintaining consistency with a controlling sequence such as the sawtooth in **Update_Set_Current**.

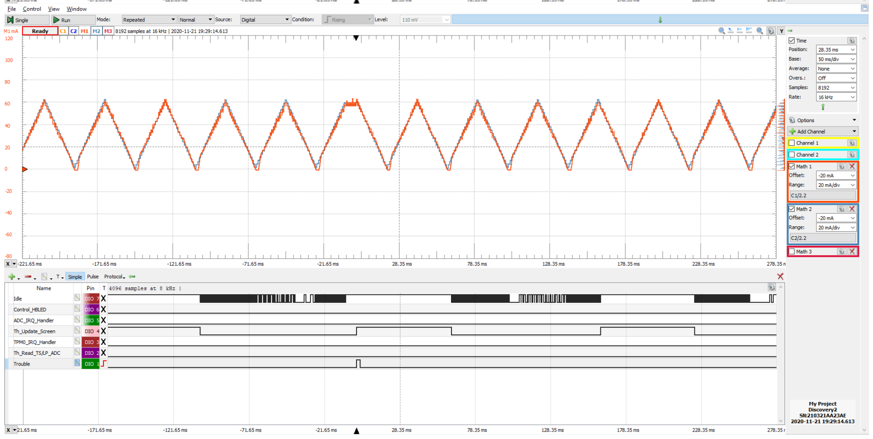
FAULT 2 – SET CURRENT LOW



🡐 *Figure 2.1. LED current does not follow setpoint after fault disables controller*

```
volatile int g_set_current=0;
int g_set_current_copy=0;
```

*Figure 2.2. A copy of **g_set_current** initialized at the top of **control.c***



🡐 *Figure 2.3. Fault detection and response code corrects variable and keeps controller enabled without much change in system behavior*

```
// make a copy of g_set_current
g_set_current_copy = g_set_current;
```

*Figure 2.4. make a copy of **g_set_current** into **g_set_current_copy***

Fault Description

This fault changed the variable **g_set_current** to 0, which turns off the LED, hence the incorrect current seen in Figure 2.1.

```
void Update_Set_Current(void) {
  // Ramp curent up and down
  static volatile int t=0;

  // set g_set_current to copy if updated outside of this routine
  g_set_current = (g_set_current == g_set_current_copy) ? g_set_current : g_set_current_copy;
```

*Figure 2.5. Check if **g_set_current** equals **g_set_current_copy** in **Update_Set_Current**. If not, set them equal. Figure 2.4. occurs at the end of this routine.*

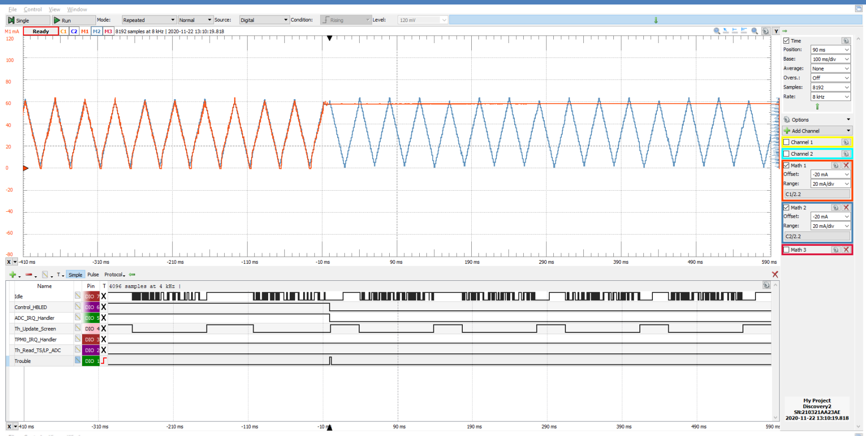
Fault Management Approach

Since Fault 1 did not consider only "high" currents, but rather *any change made to **g_set_current** outside of **Update_Set_Current***, the same approach fixes this low-current fault. See Fault 1 "Fault Management Approach" for more details regarding the fault management of this approach.

Evaluation of Effectiveness

Since both Fault 1 and 2 operate the same way, their effectiveness is the same. See Fault 1 "Evaluation of Effectiveness" for more details regarding the effectiveness of this approach.

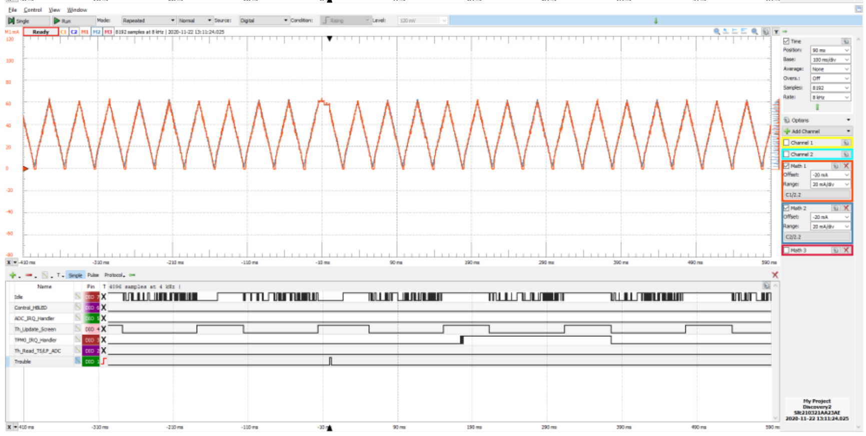
FAULT 3 – DISABLE ADC IRQ



🡐 *Figure 3.1. LED current does not follow setpoint after fault disables controller. Can observe in digital that ADC_IRQ goes low at the same time when the current stays constant*

```
volatile int g_measured_current;
int g_measured_current_prev;
int g_measured_current_temp;
```

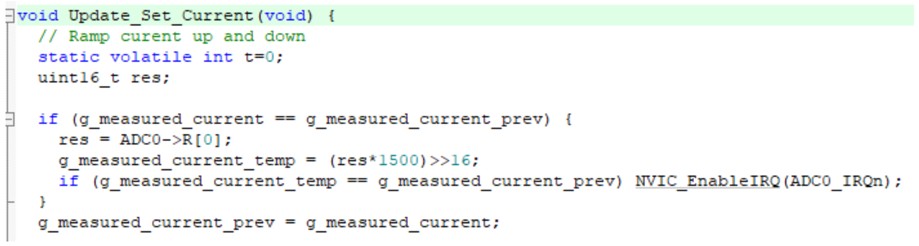
Figure 3.2. A copy and temporary variable of ***g_measured_current*** *initialized at the top of* ***control.c***



🡐 *Figure 3.3. Fault detection and response code corrects variable and keeps controller enabled without much change in system behavior*

Fault Description

This fault disabled the **ADC0_IRQ** handler, meaning no new ADC values can be generated. This keeps measured current constant, hence the incorrect updated current seen in Figure 3.1.



```
void Update_Set_Current(void) {
  // Ramp curent up and down
  static volatile int t=0;
  uint16_t res;

  if (g_measured_current == g_measured_current_prev) {
    res = ADC0->R[0];
    g_measured_current_temp = (res*1500)>>16;
    if (g_measured_current_temp == g_measured_current_prev) NVIC_EnableIRQ(ADC0_IRQn);
  }
  g_measured_current_prev = g_measured_current;
```

Figure 3.4. A couple lines which determine whether or not to enable the ***ADC0_IRQ****, explained in Fault Management Approach below*

Fault Management Approach

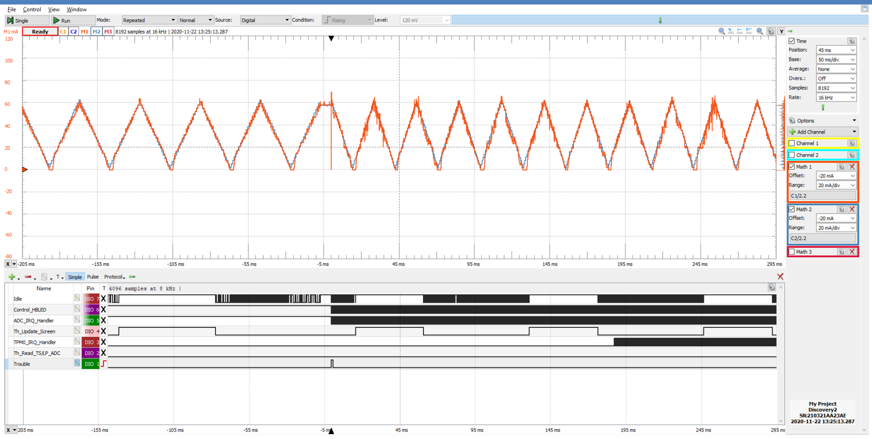
To fix the fault, it first have to be understood what happens when the ADC IRQ is disabled. **Control_HBLED** is the only routine which pulls from the ADC, and the only place where **Control_HBLED** is called is within the **ADC0_IRQ** handler. Therefore, **Control_HBLED** stops being called, resulting in the current pattern in Figure 3.1. One can see that the blue trace still continues in the expected sawtooth pattern, meaning **Update_Set_Current** is still being called, updating the set current in the proper sawtooth pattern. Therefore, we can use **Update_Set_Current** to check the ADC and see whether or not **Control_HBLED** was called or not.

In **Control_HBLED**, **g_measured_current** is updated by reading from the ADC. So if we make a variable called **g_measured_current_prev**, and copy **g_measured_current** in **Update_Set_Current**, then every time while the ADC is turned on **g_measured_current** will have the current current value while **g_measured_current_prev** will hold the previous current value. These two values are compared in **Update_Set_Current** (since this routine is called when ADC is turned off). If they are equal, it means that the ADC has not updated its value. Just to be sure that the ADC didn't just generate the same value, the ADC is pulled from again and stored into **g_measured_current_temp** (to prevent messing with calculations which use **g_measured_current** and **g_measured_current_prev**). If this temporary variable and the previous current measurement are equal yet again, **NVIC_EnableIRQ** is called to enable **ADC0_IRQ**. If they aren't equal, it continues and stores the now-previous current value into **g_measured_current_prev**.

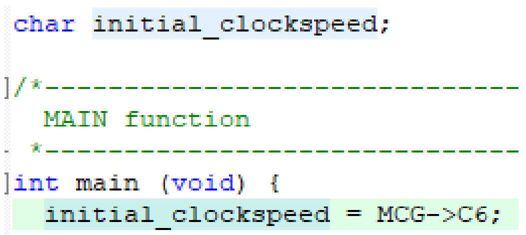
Evaluation of Effectiveness

Temporally, this approach immediately handles the fault, with a similar 8ms window of current plateauing as seen in prior faults. Programmatically, this approach will continue to enable the ADC IRQ as long as **Update_Set_Current** is called to run. Otherwise, the ADC will not be enabled. If the user wishes to disable the ADC and run **Update_Set_Current**, this will not be possible unless they are aware of this added code. A simple Boolean can be implemented in the if statement in Figure 3.4. which defines auto-enabling of the ADC to solve the problem in the previous sentence.

FAULT 4 – CHANGE MCU CLOCK



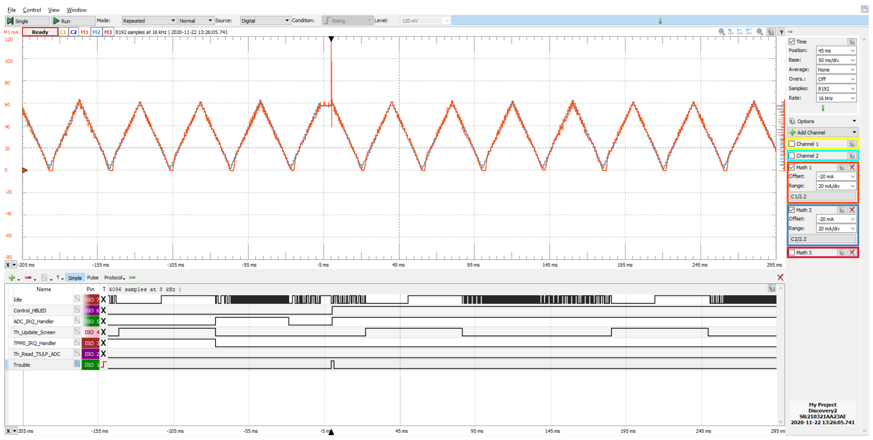
🡐 Figure 4.1. LED current follows setpoint but increases frequency



```
char initial_clockspeed;

/*-------------------------------
  MAIN function
 *-------------------------------
int main (void) {
  initial_clockspeed = MCG->C6;
```

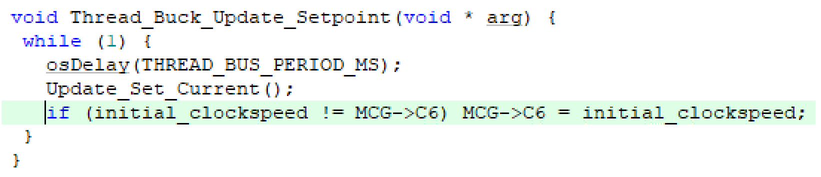
*Figure 4.2. A global copy of the initialized system clock is made in **main.c***



🡐 Figure 4.3. Fault detection and response code corrects variable and keeps controller enabled without much change in system behavior

Fault Description

This fault changed the clock speed of the system by setting a new value to **MCG->C6**, hence the increased rate of current pattern seen in Figure 4.1.

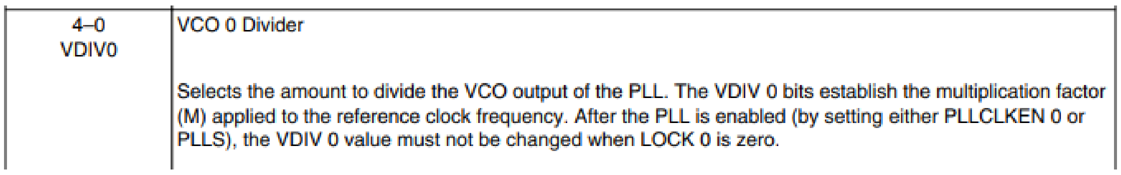


```
void Thread_Buck_Update_Setpoint(void * arg) {
 while (1) {
   osDelay(THREAD_BUS_PERIOD_MS);
   Update_Set_Current();
   if (initial_clockspeed != MCG->C6) MCG->C6 = initial_clockspeed;
 }
}
```

*Figure 4.4. Check during the thread **Thread_Buck_Update_Setpoint** if the value of **MCG->C6** equals the initial clock speed value set in **main** (seen in Figure 4.2.). If not, set them equal.*

Fault Management Approach

The clock speed for the main clock of the KL25z uses Control Register 6 of the MCG (multipurpose clock generator). A **char** is initialized in **main.c** at the first line of **main** to copy the initialized clock speed (initialized in **SystemInit** in **system_MKL25Z4.c**) titled **initial_clockspeed**. The 4-most LSB's are important to consider since they control the multiplication factor to the reference clock frequency. **MCG->C6** is initialized 0x40 (0 = multiplication factor of 24), but then the fault changes it to 0x4A (A = multiplication factor of 34), hence increasing the clock speed as seen in Figure 4.1. To counter this, every time the thread **Thread_Buck_Update_Setpoint** is called, a check is made on the register 6 of MCG to ensure the clock speed stays at the initialized value.



4–0 VDIV0	VCO 0 Divider Selects the amount to divide the VCO output of the PLL. The VDIV 0 bits establish the multiplication factor (M) applied to the reference clock frequency. After the PLL is enabled (by setting either PLLCLKEN 0 or PLLS), the VDIV 0 value must not be changed when LOCK 0 is zero.

*Figure 4.5. Description of reference to **MCG->C6** LSB's*

Evaluation of Effectiveness

This approach is effective immediately, since the thread **Thread_Buck_Update_Setpoint** has the highest priority (being **osPriorityAboveNormal**) and is called at a high frequency to make small updates to the **g_set_current** via **Update_Set_Current** (as shown in previous faults). Meaning, since this thread has the highest priority, it is very effective in catching the fault quickly. From a programming standpoint, the clock speed is unable to be changed unless **initial_clockspeed** is changed with it. This is not ideal, but it prevents register 6 from being tampered with throughout the runtime.

FAULT 5 – SLOW TPM

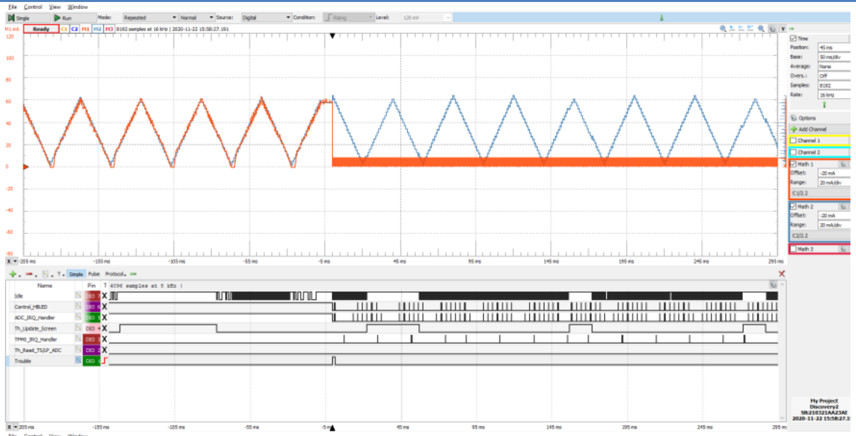


Figure 5.1. LED current does not follow setpoint after fault disables controller

```
PWM_Init(TPM0, PWM_HBLED_CHANNEL, PWM_PERIOD,
```

Figure 5.2. A line (not added) which is used to initialize the TPM0 counter in **control.c**, **Init_Buck_HBLED**

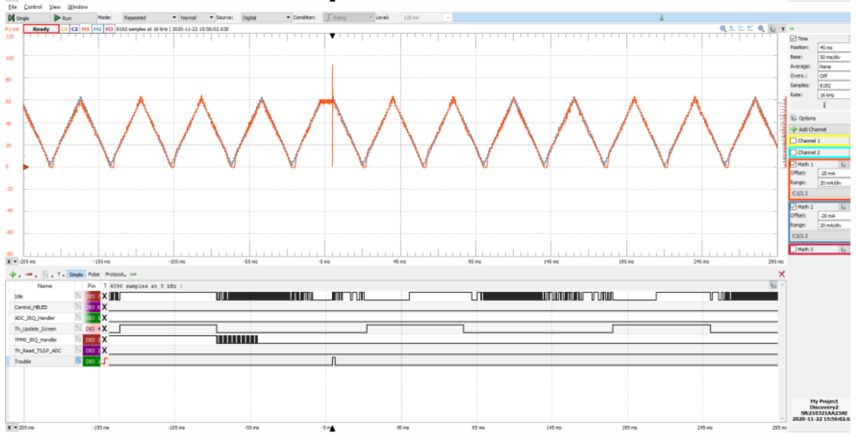


Figure 5.3. Fault detection and response code corrects variable and keeps controller enabled without much change in system behavior

Fault Description

This fault changed the value for **TPM0->MOD**, the modulo register of the timer/PWM module, which changes the duty cycle of the PWM, hence the incorrect current seen in Figure 5.1.

```
void Thread_Buck_Update_Setpoint(void * arg) {
 while (1) {
   osDelay(THREAD_BUS_PERIOD_MS);
   Update_Set_Current();
   if (initial_clockspeed != MCG->C6) MCG->C6 = initial_clockspeed
   if (TPM0->MOD != tpm_periods[0]) TPM0->MOD = tpm_periods[0];
```

Figure 5.4. Check during the thread **Thread_Buck_Update_Setpoint** if the value of **TPM0->MOD** equals the initialized period for TPM0 (seen in Figure 5.2. and 5.5.). If not, set them equal.

```
volatile uint16_t tpm_periods[3];

void PWM_Init(TPM_Type * TPM, uint8_t chan
  uint8_t pos_polarity, uint8_t prescaler_
{
  //turn on clock to TPM
  switch ((int) TPM) {
    case (int) TPM0:
      SIM->SCGC6 |= SIM_SCGC6_TPM0_MASK;
      tpm_periods[0] = period;
      break;
    case (int) TPM1:
      SIM->SCGC6 |= SIM_SCGC6_TPM1_MASK;
      tpm_periods[1] = period;
      break;
    case (int) TPM2:
      SIM->SCGC6 |= SIM_SCGC6_TPM2_MASK;
      tpm_periods[2] = period;
      break;
```

Figure 5.5. →
Store values for each TPM in **tpm_periods** array in the switch statement..

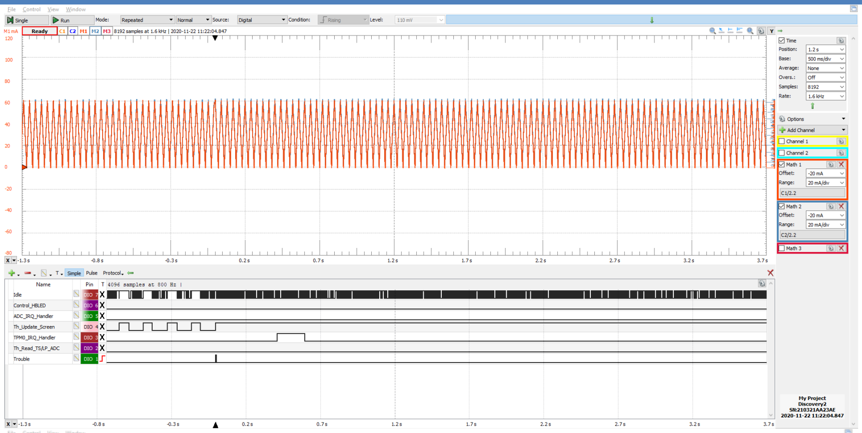
Fault Management Approach

The timer/PWM module has three modules. The first of which is initialized from **control.c** (as seen in Figure 5.2.) and is the timer used to control the speed of the HBLED updating process. TPM0 is initialized in **Init_Buck_HBLED → PWM_Init** by passing in the location of TPM0 and its period. Within **PWM_Init**, the initialized periods are stored in a uint16_t array **tpm_periods** (seen in Figure 5.5.) which is a volatile and external array accessed in **thread.c** to check whether or not the current value in **TPM0->MOD** equals the initialized TPM period (Figure 5.4.). The same thread as the previous fault is used (thread **Thread_Buck_Update_Setpoint**) as this is the highest priority thread and will catch the fault during a very short period of time.

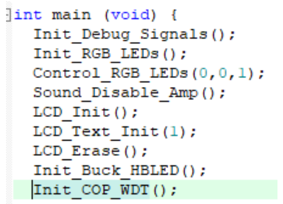
Evaluation of Effectiveness

Similar to the previous fault, the highest priority thread **Thread_Buck_Update_Setpoint** is used. Though a new high priority thread could be made to check both these faults, its effective in the way it works and how frequently this thread runs, allowing faults to be caught early. Therefore, the placement of this statement is temporally efficient but programmatically inefficient. However, each TPM module has its period stored and each TPM module can be initialized again to change the values in **tpm_periods**. So, unlike the previous fault, **TPM0->MOD** can be updated properly by calling **PWM_Init** again. These two aspects make the addition of **tpm_periods** an overall more efficient and flexible design, which could further be improved by making running a separate high priority thread solely for fault management.

FAULT 6 – LCD MUTEX

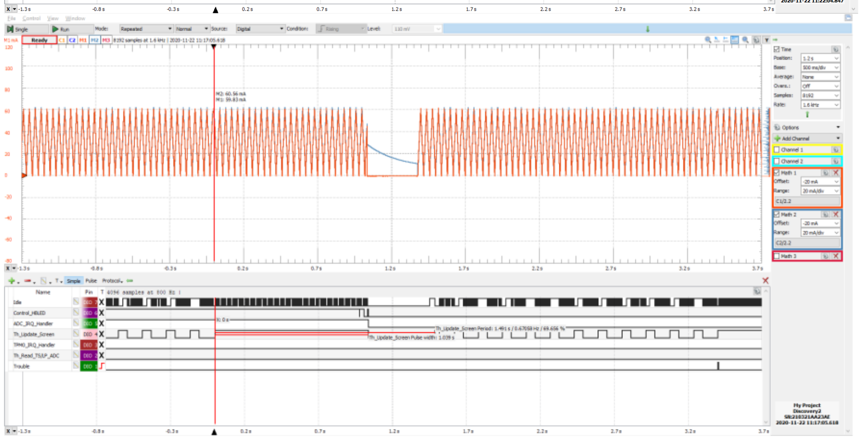


← *Figure 6.1. LED current follows, but in digital **Th_Update_Screen** stopped being services*



```
int main (void) {
    Init_Debug_Signals();
    Init_RGB_LEDs();
    Control_RGB_LEDs(0,0,1);
    Sound_Disable_Amp();
    LCD_Init();
    LCD_Text_Init(1);
    LCD_Erase();
    Init_Buck_HBLED();
    Init_COP_WDT();
```

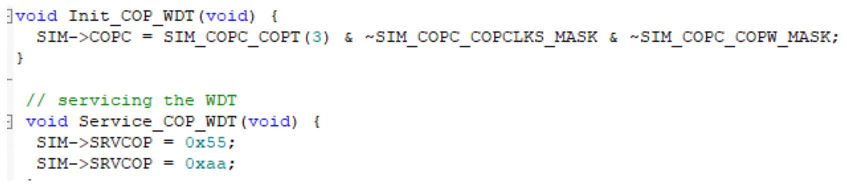
*Figure 6.2. Added function **Init_COP_WDT** to initialize the watchdog timer in **main.c***



← *Figure 6.3. Fault detection and response code resets system to properly start again*

FAULT DESCRIPTION

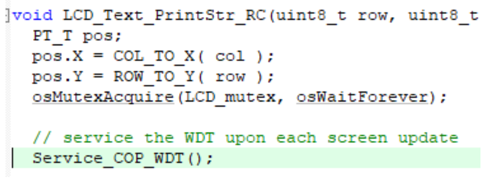
This fault acquires the LCD mutex which blocks **LCD_Text_PrintStr_RC** from running, halting **Thread_Update_Screen**. Once the mutex is returned, only then will the OS continue to run the thread to properly update the screen



```
void Init_COP_WDT(void) {
    SIM->COPC = SIM_COPC_COPT(3) & ~SIM_COPC_COPCLKS_MASK & ~SIM_COPC_COPW_MASK;
}

// servicing the WDT
void Service_COP_WDT(void) {
    SIM->SRVCOP = 0x55;
    SIM->SRVCOP = 0xaa;
```

*Figure 6.4. Setup for **Init_COP_WDT** and **Service_COP_WDT** functions in **timers.c***



```
void LCD_Text_PrintStr_RC(uint8_t row, uint8_t
    PT_T pos;
    pos.X = COL_TO_X( col );
    pos.Y = ROW_TO_Y( row );
    osMutexAcquire(LCD_mutex, osWaitForever);

    // service the WDT upon each screen update
    Service_COP_WDT();
```

*Figure 6.5. Service the WDT within **LCD_Test_PrintStr_RC***

Fault Management Approach

After realizing that **LCD_mutex** is only used within functions **LCD_Fill_Rectangle** and **LCD_Text_PrintStr_RC**, this meant that a thread will be blocked upon the acquired mutex. After deciding to backtrack **LCD_Text_PrintStr_RC** (there were more instances of these function calls compared to **LCD_Fill_Rectangle**, meaning greater priority), the hierarchy was found to stem from thread **Thread_Update_Screen** and follow: **UI_Draw_Screen → UI_Draw_Fields → LCD_Text_PrintStr_RC**. If **Thread_Update_Screen** was being blocked, one of the most effective ways to unblock it without polling to check whether or not the **LCD_mutex** has been acquired or not was to service a watchdog timer (WDT) within **LCD_Text_PrintStr_RC** (Figure 6.5.) so once the thread it blocked for a sufficient amount of time, the WDT resets the system.

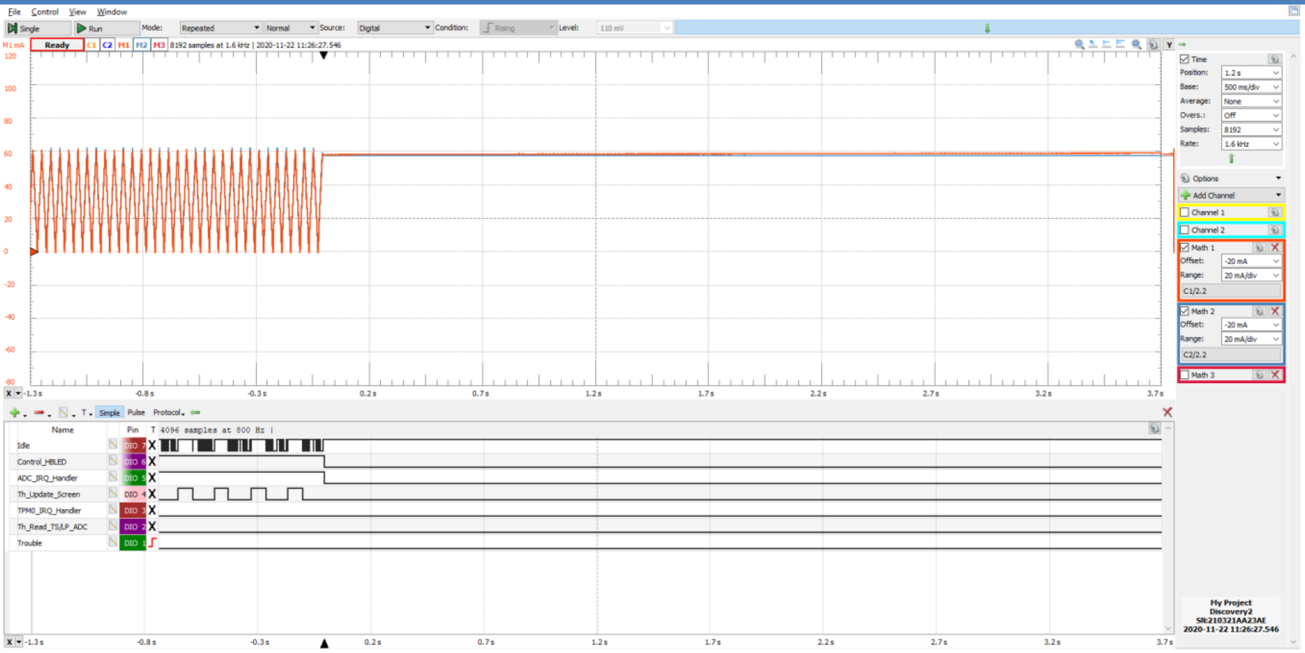
The WDT is initialized by a call from **main** (Figure 6.2.) to **timers.c** to initialize the KL25z COP WDT with the specifications show in Figure 6.4. To service the WDT, a function in **timers.c** called **Service_COP_WDT** is created to change the value of **SIM->SRVCOP** to any value other than 0x55. After some time of not being serviced, the WDT will then reset the system.

Evaluation of Effectiveness

Since the system is reset using the WDT, there is a loss in temporal efficiency. **Th_Update_Screen** in the digital scope oscillates between 62.5ms high and 102.5ms low (duty cycle of ~38%) servicing the thread to update the LCD screen. Then in Figure 6.1. it can be seen that it stops being serviced and stays high. After the WDT is implemented and serviced (Figure 6.5.), it can be seen in Figure 6.3. that there is a 1.04s period where the system continues to run without the WDT resetting the system, roughly 6.3 cycles of **Th_Update_Screen**. Then it takes roughly 350ms for the system to reset (Figure 6.3.) before the sawtooth pattern emerges again to properly run.

In most instances, the **LCD_mutex** would not be acquired outside of the **Thread_Update_Screen** thread, because it would be poor programming to update the LCD outside of a thread dedicated to updating the screen. In addition, it would be absurd to poll and continuously check whether this **LCD_mutex** value had been acquired or not, so a WDT is a good solution for handling this fault and any other non-responsive screen.

FAULT 7 – DISABLE ALL IRQS



Mode: Repeated Normal Source: Digital Condition: Rising Level: 110 mV

8192 samples at 1.6 kHz | 2020-11-22 11:26:27.546

4096 samples at 800 Hz

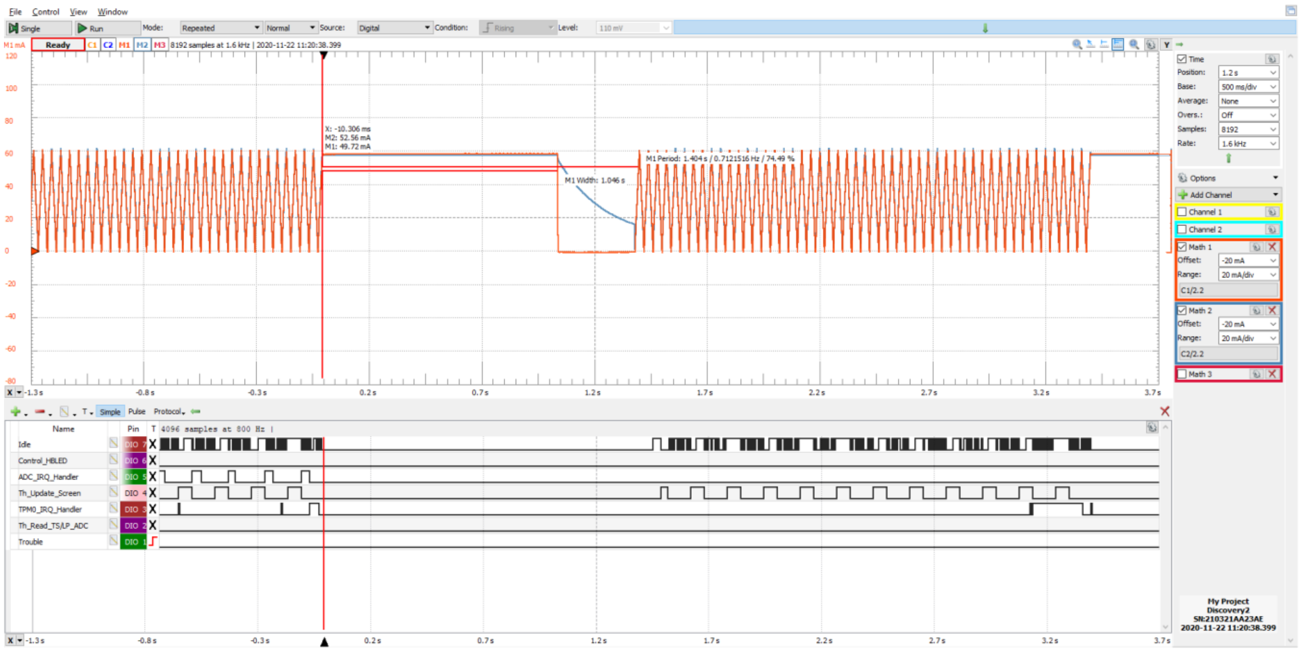
Position: 1.2 s
Base: 500 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 1.6 kHz

Math 1 Offset: -20 mA Range: 20 mA/div C1/2.2

Math 2 Offset: -20 mA Range: 20 mA/div C2/2.2

My Project
Discovery2
SN:210321AA23AE
2020-11-22 11:26:27.546

Figure 7.1. All processes stop due to disabling of all IRQs



Mode: Repeated Normal Source: Digital Condition: Rising Level: 110 mV

8192 samples at 1.6 kHz | 2020-11-22 11:20:38.399

4096 samples at 800 Hz

X: -10.306 ms
M2: 52.56 mA
M1: 49.72 mA

M1 Width: 1.046 s

M1 Period: 1.404 s / 0.7121516 Hz / 74.49 %

Position: 1.2 s
Base: 500 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 1.6 kHz

Math 1 Offset: -20 mA Range: 20 mA/div C1/2.2

Math 2 Offset: -20 mA Range: 20 mA/div C2/2.2

My Project
Discovery2
SN:210321AA23AE
2020-11-22 11:20:38.399

Figure 7.2. Fault detection and response code resets system to properly start again

Fault Description

This fault disables all IRQs, which prevents most aspects of the system from running

Fault Management Approach

Since all IRQs are disabled, the thread fault injector happens to stay within a continuous while loop with **osDelay(FAULT_PERIOD)**. The delay occurs and the idle thread should run, but as seen in Figure 7.1. there is no idle thread signal. No other threads are called during this osDelay period, thus returning to the same **osDelay(FAULT_PERIOD)** line within the **Thread_Fault_Injector**. Because of this, **Thread_Update_Screen** never runs, thus never servicing the WDT within **LCD_Test_PrintStr_RC**. The WDT then decides to reset the system after the allotted time in the same way done in Fault 6.

Refer to Figures 6.2., 6.4., and 6.5. for listings related to the WDT.

Evaluation of Effectiveness

The same temporal resolution for handling the fault results as in Fault 6: ~1.04s for the WDT to reset the system and ~350ms for the system to reset.

FAULT 8 – STACK OVERFLOW

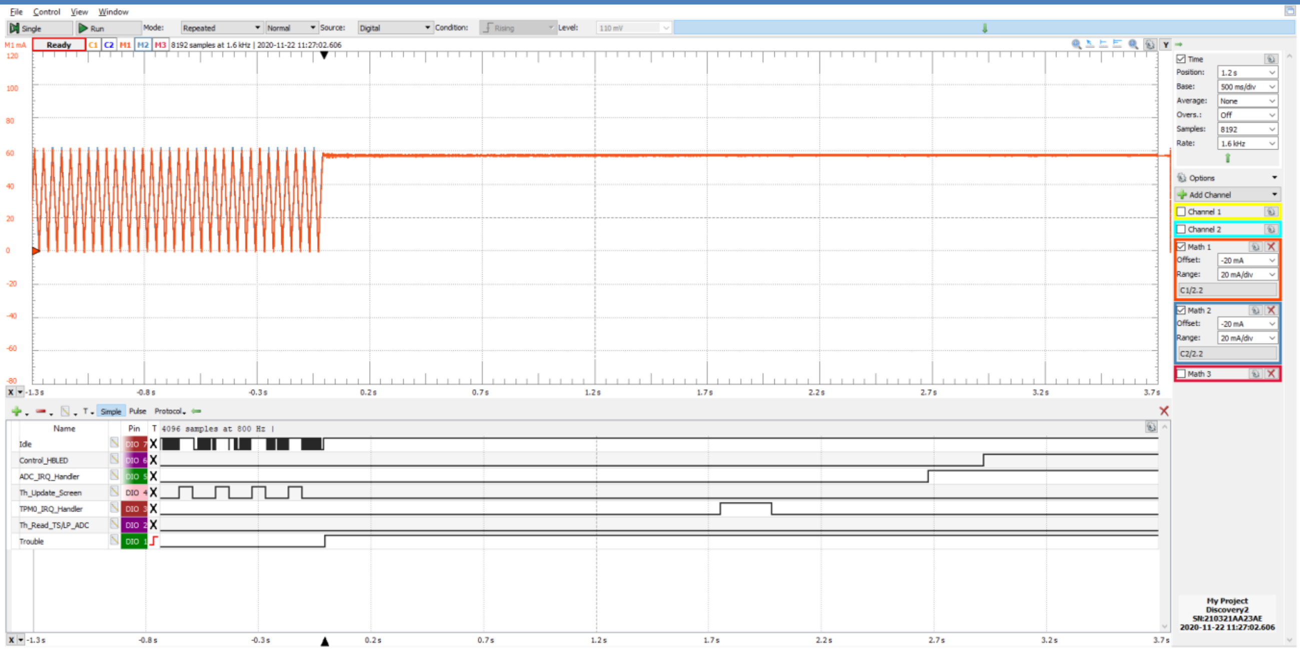


Figure 8.1. All processes (threads) other than IRQ handlers stop due to infinite recursive loop

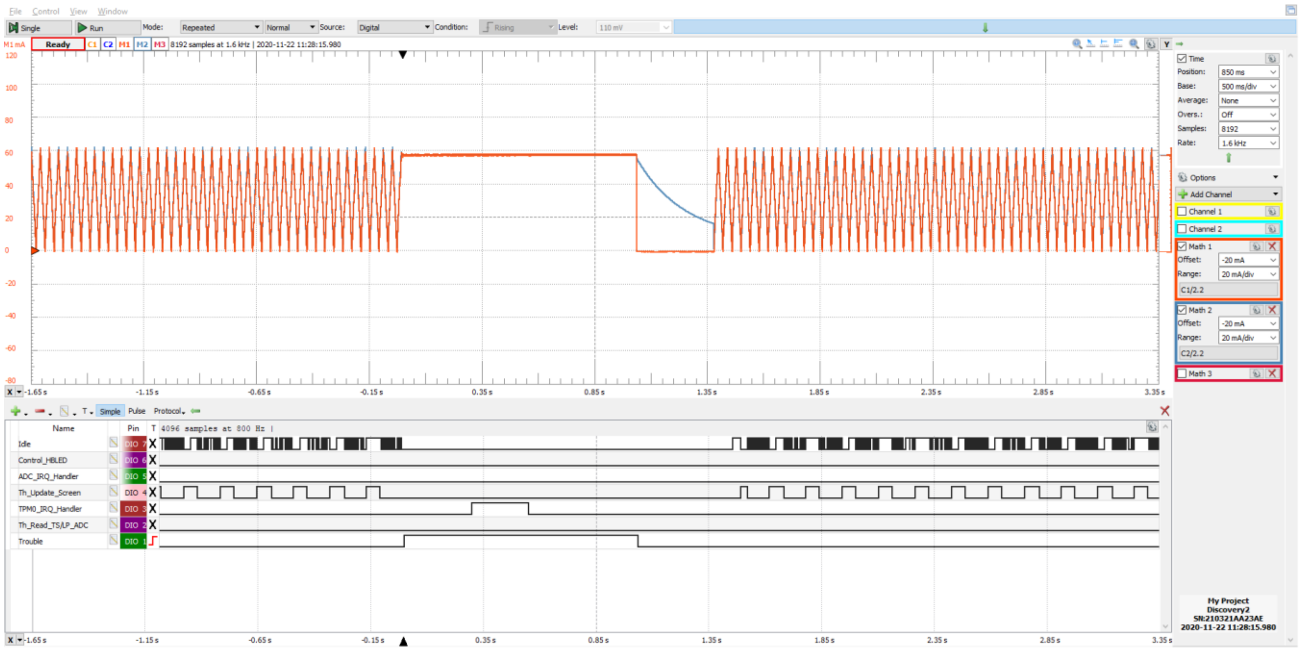


Figure 8.2. Fault detection and response code resets system to properly start again

Fault Description

This fault decides to run an infinite recursive loop, setting the argument of the function to 1 greater than the previous to cause a stack overflow

Fault Management Approach

The routine **Fault_Recursion_Test(n)** within **fault.c** returns the value **Fault_Recursion_Test(n + 1)**, thus causing an infinite recursive loop of parameters within the stack. The stack will thus overflow. The only routines within the system capable of temporarily escaping this boundless hell are the interrupt service routines, which are all serviced properly. After being serviced, the CPU is directed back toward the **Fault_Recursion_Test**, never having time to run any of the threads within the RTOS. Because of this, **Thread_Update_Screen** never runs, thus never servicing the WDT within **LCD_Test_PrintStr_RC**. The WDT then decides to reset the system after the allotted time in the same way done in Fault 6 & 7.

Refer to Figures 6.2., 6.4., and 6.5. for listings related to the WDT.

Evaluation of Effectiveness

The same temporal resolution for handling the fault results as in Faults 6 & 7: ~1.04s for the WDT to reset the system and ~350ms for the system to reset.

FAULT 9 – FILL QUEUE

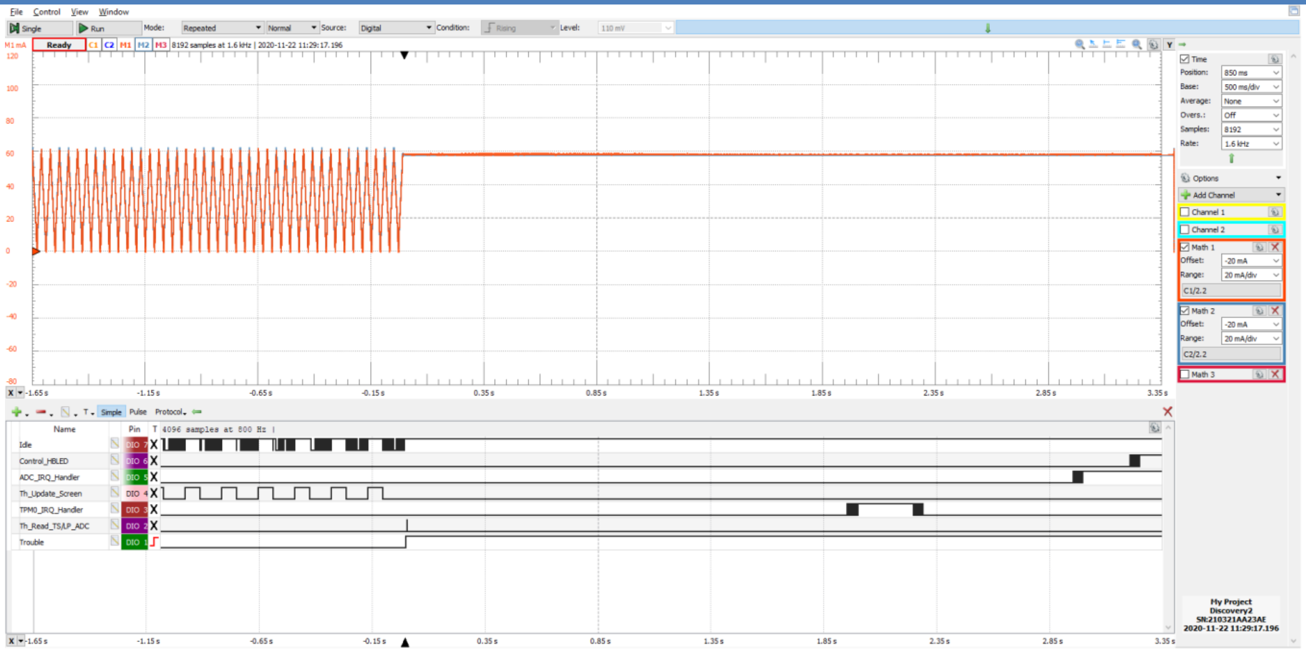


Figure 9.1. All processes (threads) other than IRQ handlers stop due to infinite loop

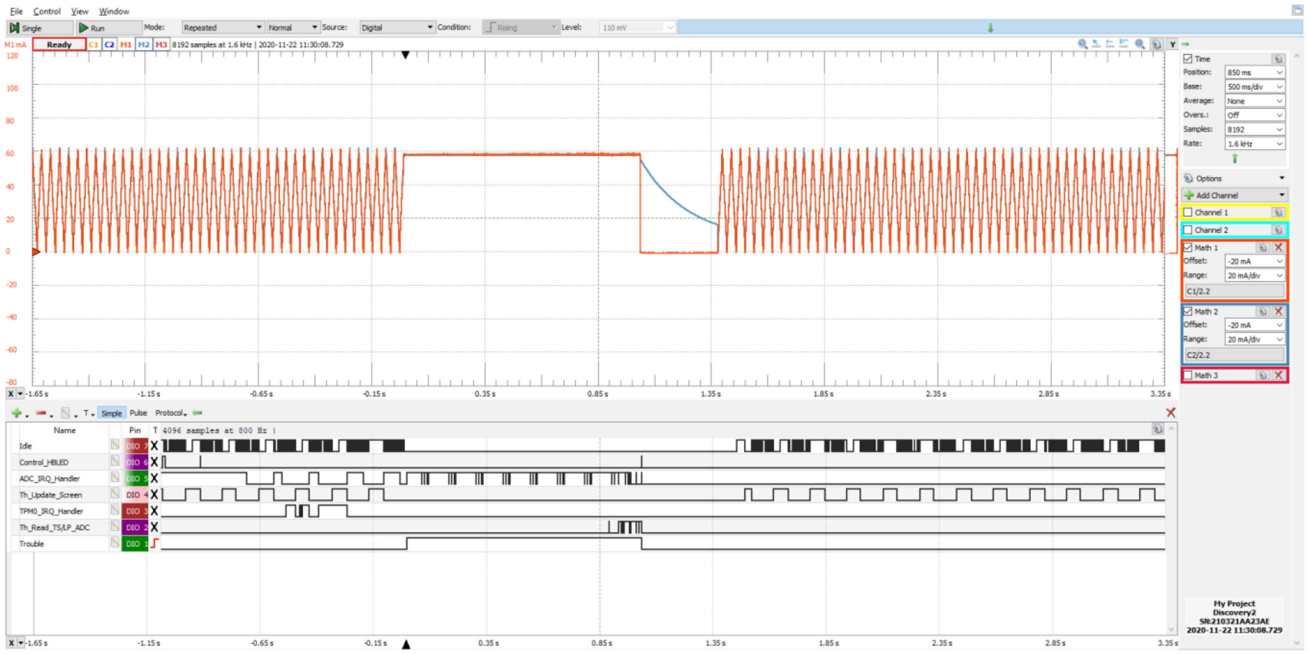


Figure 9.2. Fault detection and response code resets system to properly start again

Fault Description

This fault decides to run an infinite loop in **Fault_Fill_Queue** where OS messages are continuously added to a queue

Fault Management Approach

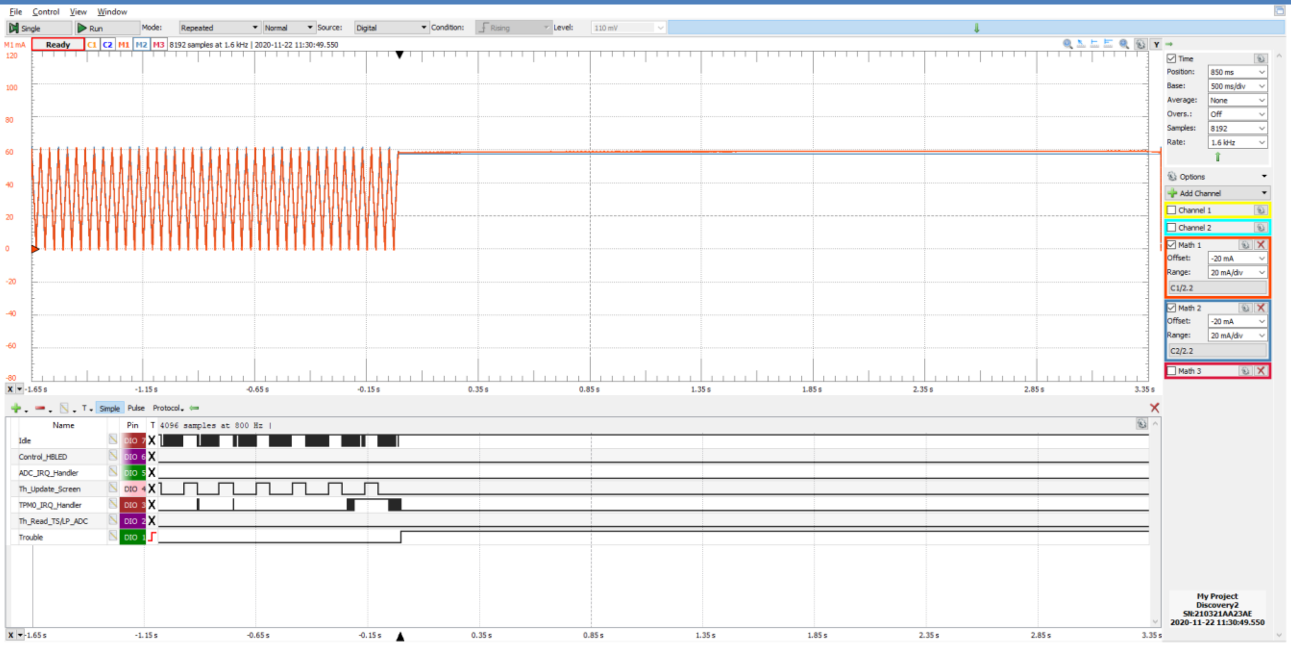
Very similar to the previous fault (Fault 8), there is an infinite loop which uses up all the time of the RTOS preventing other threads from running. It is evident how similar this fault is from the last by observing the digital signal between Figure 8.1. and Figure 9.1., where IRQs are still serviced but threads are not. Because of this, **Thread_Update_Screen** never runs, thus never servicing the WDT within **LCD_Test_PrintStr_RC**. The WDT then decides to reset the system after the allotted time in the same way done in Fault 6, 7, & 8.

Refer to Figures 6.2., 6.4., and 6.5. for listings related to the WDT.

Evaluation of Effectiveness

The same temporal resolution for handling the fault results as in Faults 6, 7, & 8: ~1.04s for the WDT to reset the system and ~350ms for the system to reset.

FAULT 10 – DISABLE PERIPHERAL CLOCKS



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

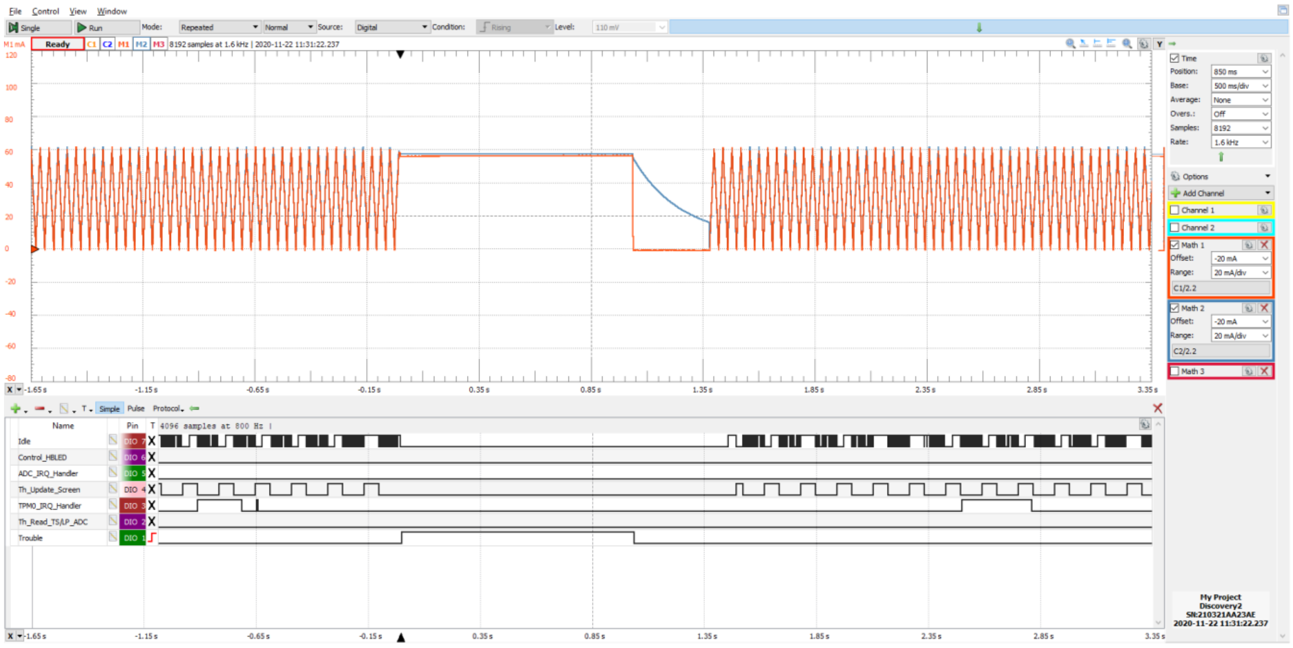
8192 samples at 1.6 kHz | 2020-11-22 11:30:49.550

Position: 850 ms
Base: 500 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 1.6 kHz

Math 1 — Offset: -20 mA, Range: 20 mA/div, C1/2.2
Math 2 — Offset: -20 mA, Range: 20 mA/div, C2/2.2

T 4096 samples at 800 Hz

Figure 10.1. Processes halt due to disabling of peripheral clock



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

8192 samples at 1.6 kHz | 2020-11-22 11:31:22.237

Position: 850 ms
Base: 500 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 1.6 kHz

Math 1 — Offset: -20 mA, Range: 20 mA/div, C1/2.2
Math 2 — Offset: -20 mA, Range: 20 mA/div, C2/2.2

T 4096 samples at 800 Hz

Figure 10.2. Fault detection and response code resets system to properly start again

Fault Description

This fault sets the register **SIM->SCGC6**, which is the clock gating control register used, to 0, disabling all clocks set by **SCGC6**

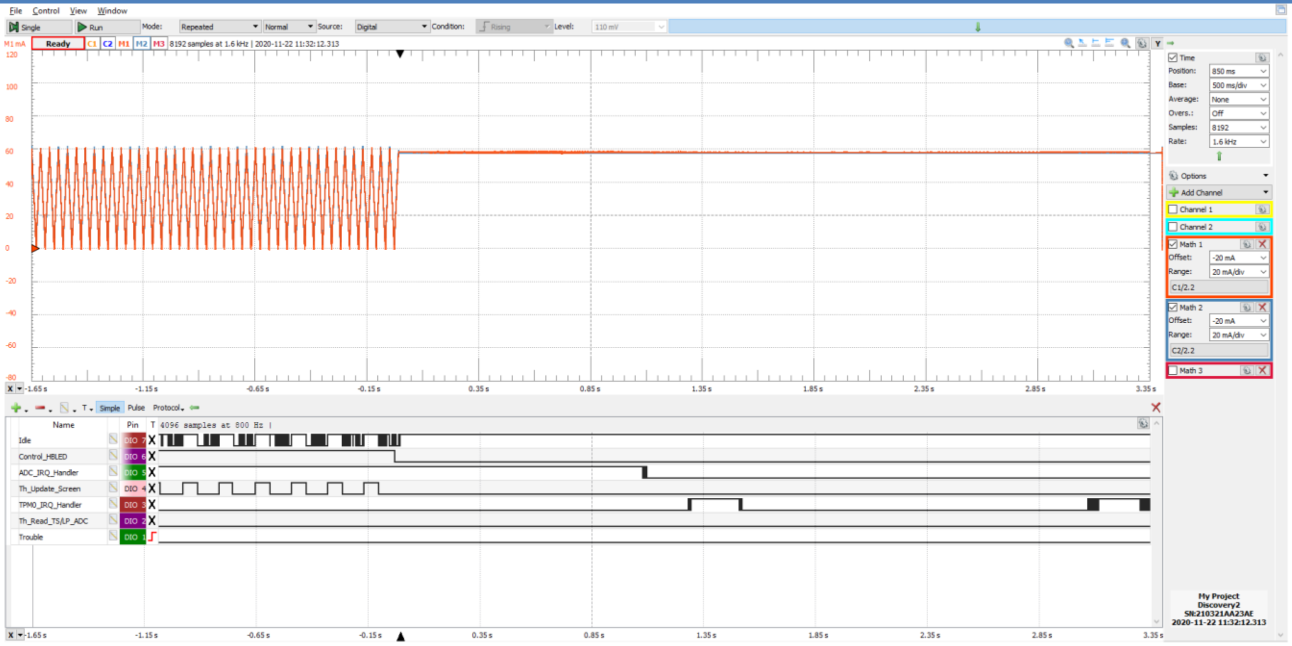
Fault Management Approach

SIM->SCGC6 is the 6th register used by **SIM** which connects the clock to the following peripherals: **TMP0-2**, **ADC0**, **DAC0**, **PIT**, **DMAMUX**, and **FTF**. The 29th bit defines **RTC** access, which enables or disables access and interrupts. Because the entire register was set to 0, all the peripherals listed above cut their connections to the clock and **RTC** access is disabled. This means that **TMP0**, **ADC0**, and other interrupts used in this project are unable to be serviced. This can be seen in Figure 10.1. (contrast to Figures 8.1. and 9.1.) where interrupts are all low. And since the clock prevents some essential peripherals from running, the program appears to get stuck on an **osDelay** call right after the switch condition in **Test Fault**. Because of this, **Thread_Update_Screen** never runs, thus never servicing the WDT within **LCD_Test_PrintStr_RC**. The WDT then decides to reset the system after the allotted time in the same way done in Fault 6, 7, 8, & 9. Refer to Figures 6.2., 6.4., and 6.5. for listings related to the WDT.

Evaluation of Effectiveness

The same temporal resolution for handling the fault results as in Faults 6, 7, 8, & 9: ~1.04s for the WDT to reset the system and ~350ms for the system to reset.

FAULT 11 – OS KERNEL LOCK



Mode: Repeated Normal Source: Digital Condition: Rising Level: 110 mV

8192 samples at 1.6 kHz | 2020-11-22 11:32:12.313

Time — Position: 850 ms; Base: 500 ms/div; Average: None; Overs.: Off; Samples: 8192; Rate: 1.6 kHz

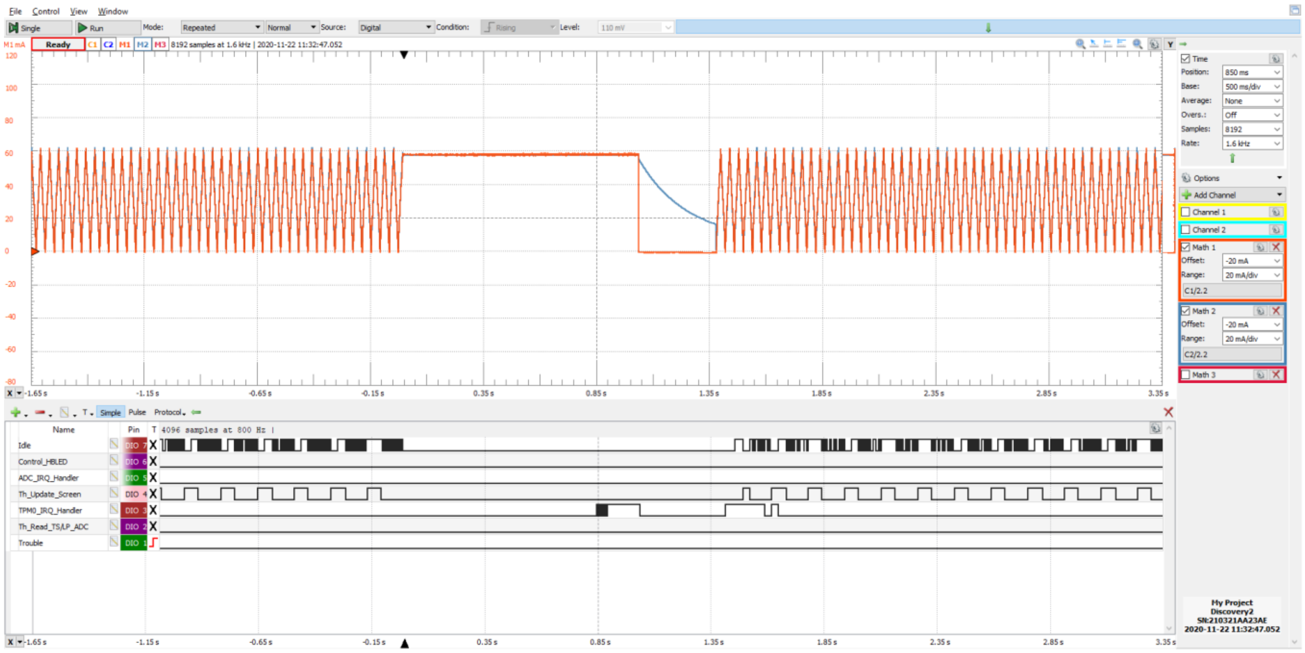
Math 1 — Offset: -20 mA; Range: 20 mA/div; C1/2.2

Math 2 — Offset: -20 mA; Range: 20 mA/div; C2/2.2

T 4096 samples at 800 Hz

SN:210321AA23AE 2020-11-22 11:32:12.313

Figure 11.1. All threads locked while rest of the system continues



Mode: Repeated Normal Source: Digital Condition: Rising Level: 110 mV

8192 samples at 1.6 kHz | 2020-11-22 11:32:47.052

Time — Position: 850 ms; Base: 500 ms/div; Average: None; Overs.: Off; Samples: 8192; Rate: 1.6 kHz

Math 1 — Offset: -20 mA; Range: 20 mA/div; C1/2.2

Math 2 — Offset: -20 mA; Range: 20 mA/div; C2/2.2

T 4096 samples at 800 Hz

SN:210321AA23AE 2020-11-22 11:32:47.052

Figure 11.2. Fault detection and response code resets system to properly start again

Fault Description

This fault calls **osKernelLock** at a forbidden location with no **osKernelUnlock** to restore the RTOS

Fault Management Approach

The function **osKernelLock** essentially locks the RTOS kernel, preventing all tasks and threads from executing. When improperly called with no **osKernelUnlock**, the system has nothing to do, except service IRQs, since this system has already executed **main**. Normally, the OS would continue running forever. But due to the locking of the kernel, no threads are accessed. Because of this, **Thread_Update_Screen** never runs, thus never servicing the WDT within **LCD_Test_PrintStr_RC**. The WDT then decides to reset the system after the allotted time in the same way done in Fault 6, 7, 8, 9, & 10.

Refer to Figures 6.2., 6.4., and 6.5. for listings related to the WDT.

Evaluation of Effectiveness

The same temporal resolution for handling the fault results as in Faults 6, 7, 8, 9, & 10: ~1.04s for the WDT to reset the system and ~350ms for the system to reset.

FAULT 12 – HIGH PRIORITY THREAD

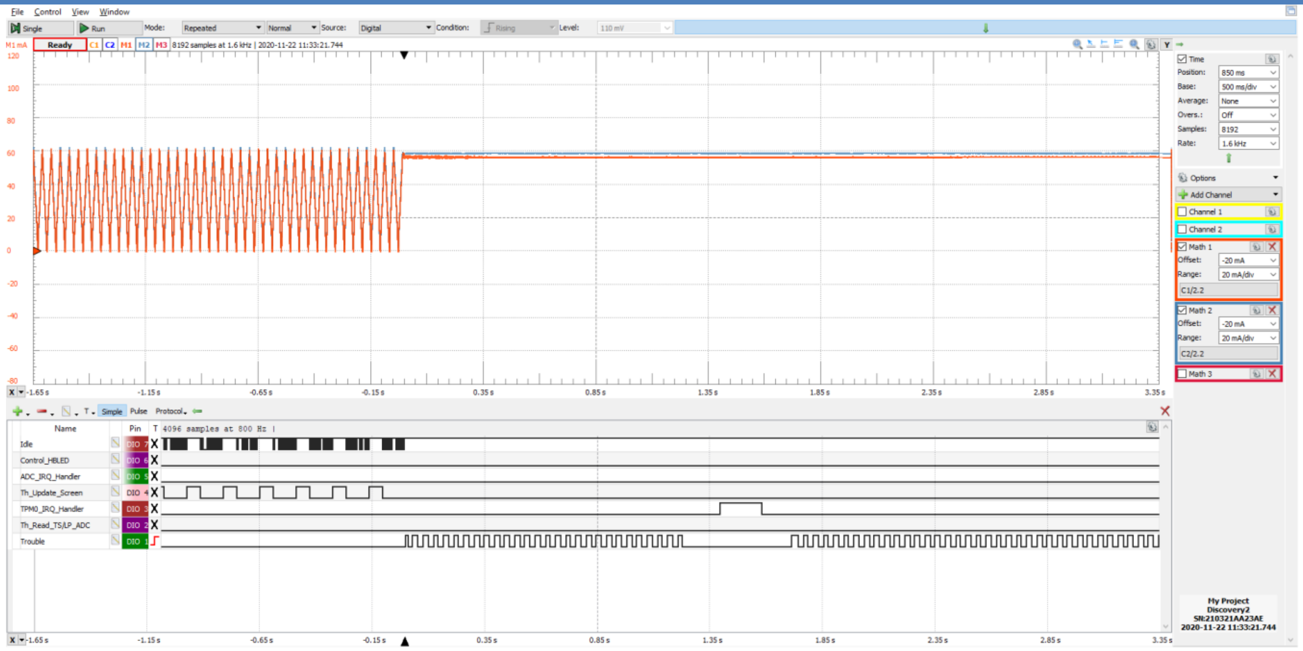


Figure 12.1. Most processes stop due to introduction of new higher priority thread

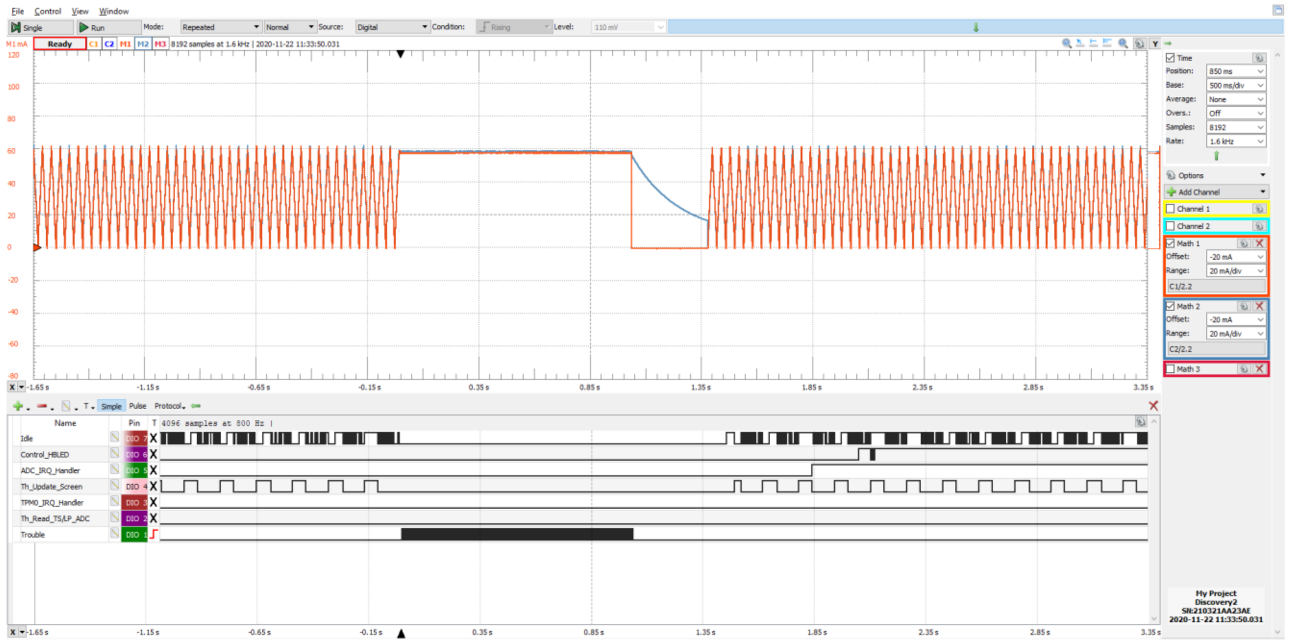


Figure 12.2. Fault detection and response code resets system to properly start again

Fault Description

This fault manifests a new higher priority thread whose only task is to toggle the **DBG_FAULT** signal

Fault Management Approach

A new thread of priority **osPriorityRealtime** (higher than our previous highest thread being priority **osPriorityAboveNormal**) is established and runs an infinite while loop toggling the **DGB_FAULT** signal, seen in the digital outputs of Figures 12.1. and 12.2. (though in Figure 12.1. it's much lower resolution). The thread consumes the RTOS, never allowing any other threads to run (but allowing for normal IRQs, as seen in Figure 12.1.). Because of this, **Thread_Update_Screen** never runs, thus never servicing the WDT within **LCD_Test_PrintStr_RC**. The WDT then decides to reset the system after the allotted time in the same way done in Fault 6, 7, 8, 9, 10 & 11.

Refer to Figures 6.2., 6.4., and 6.5. for listings related to the WDT.

Evaluation of Effectiveness

The same temporal resolution for handling the fault results as in Faults 6, 7, 8, 9, 10, & 11: ~1.04s for the WDT to reset the system and ~350ms for the system to reset.

EXTRA CREDIT

- Properly and efficiently handled all of the 13 faults (0 through 12) out of the 10 required for ECE 560
- Same solution used to handle Faults 1 & 2
- Same solution used to handle Faults 6, 7, 8, 9, 10, 11, & 12
- Additional features of some solutions to add flexibility in further software development

RETROSPECTIVE

One of the key lessons I learned is how incredibly useful the watchdog timer can be in the dire most situations. It is best to service the watchdog timer on a thread or routine that should consistently be called throughout the runtime of the system, so that when the system breaks/crashes/etc., the watchdog timer is able to catch this and reset the system. It was not intended for the watchdog timer to solve 6 of the 13 faults, but it ended up being a very valuable resource. One thing I would do different next time is inhibit myself from using the watchdog timer for faults that can be solved otherwise. This would lead to a low-level analysis of the software as well as hardware architecture to understand precisely what is going wrong and how it can be caught, before getting out of hand. And this would result in a greater understanding of the KL25z as well as embedded systems as a whole.

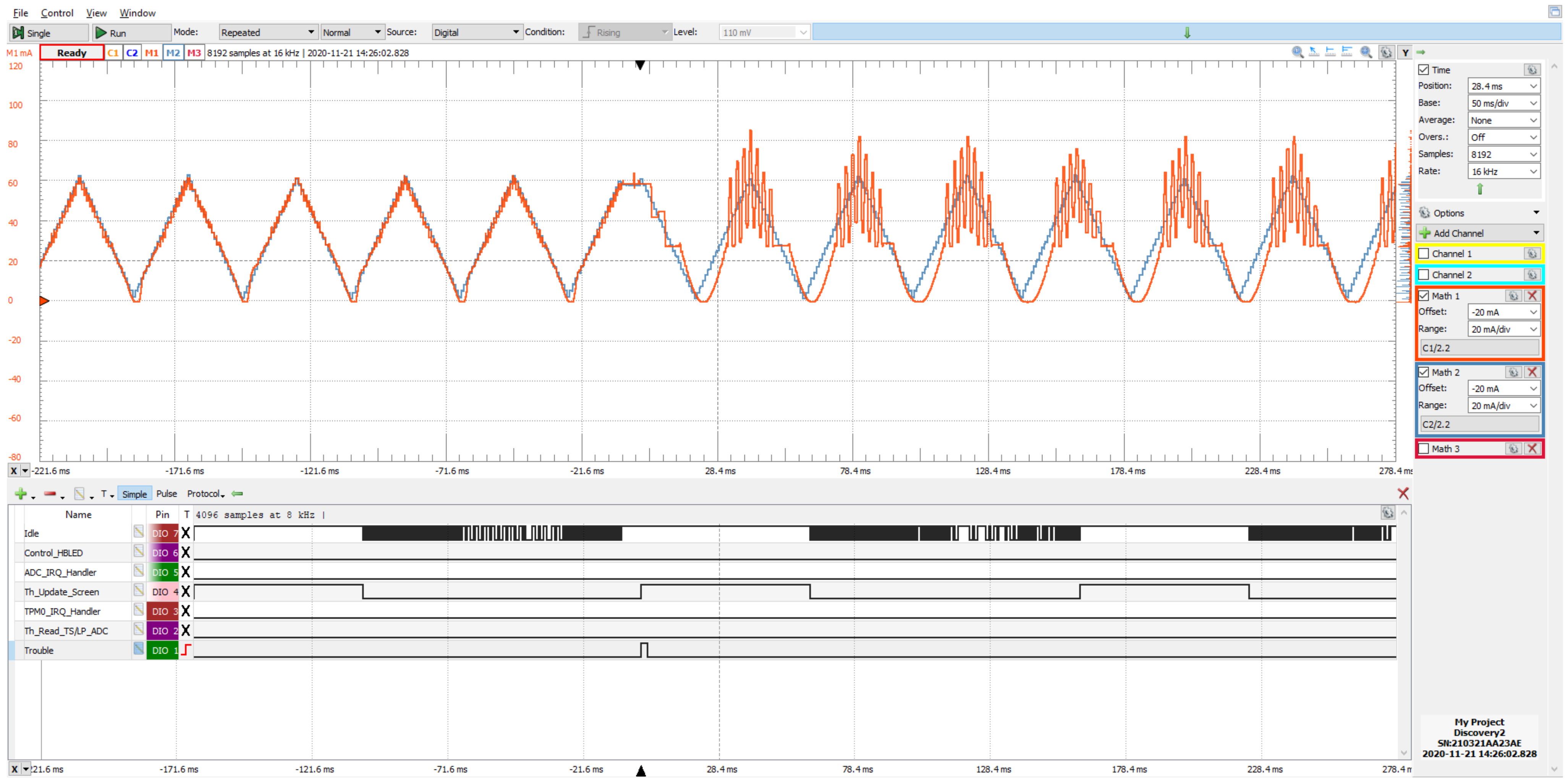
I would not change the project/course material to make it more effective and feasible, because I already think the way the information was present it already was effective and feasible. However, I would alter the project/course material to give students more hands-on development with the KL25z as well as more time.

Rather than the premise of this project being “there are faults injected, handle them”, students should work on developing some end-of-semester project (similar to ECE 306) *with prior specifications of making the software fault proof*. What I mean, is giving the assignment “program X in a way where unsolicited changes of control variables A, B, and C are managed as well as managing the following common errors: stack overflow, filled queue, disable all IRQs, etc.”

This would not only allow students to explore the KL25z more in developing this (simple) program “X”, but also they would have to develop their own fault testbench. If their program is clean, then they will have no problem implemented these fault handlers. If their program consists of spaghetti code, they will have a difficult time. And this project was quite straight forward because the code provided is extremely clean.

APPENDIX

Oscilloscope figures for Faults 0 – 6 are shown below, as an additional reference (if needed) due to their small scales on the pages. Faults 7 – 12 are not shown since these are already legible.



8192 samples at 16 kHz | 2020-11-21 14:26:02.828

Position: 28.4 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

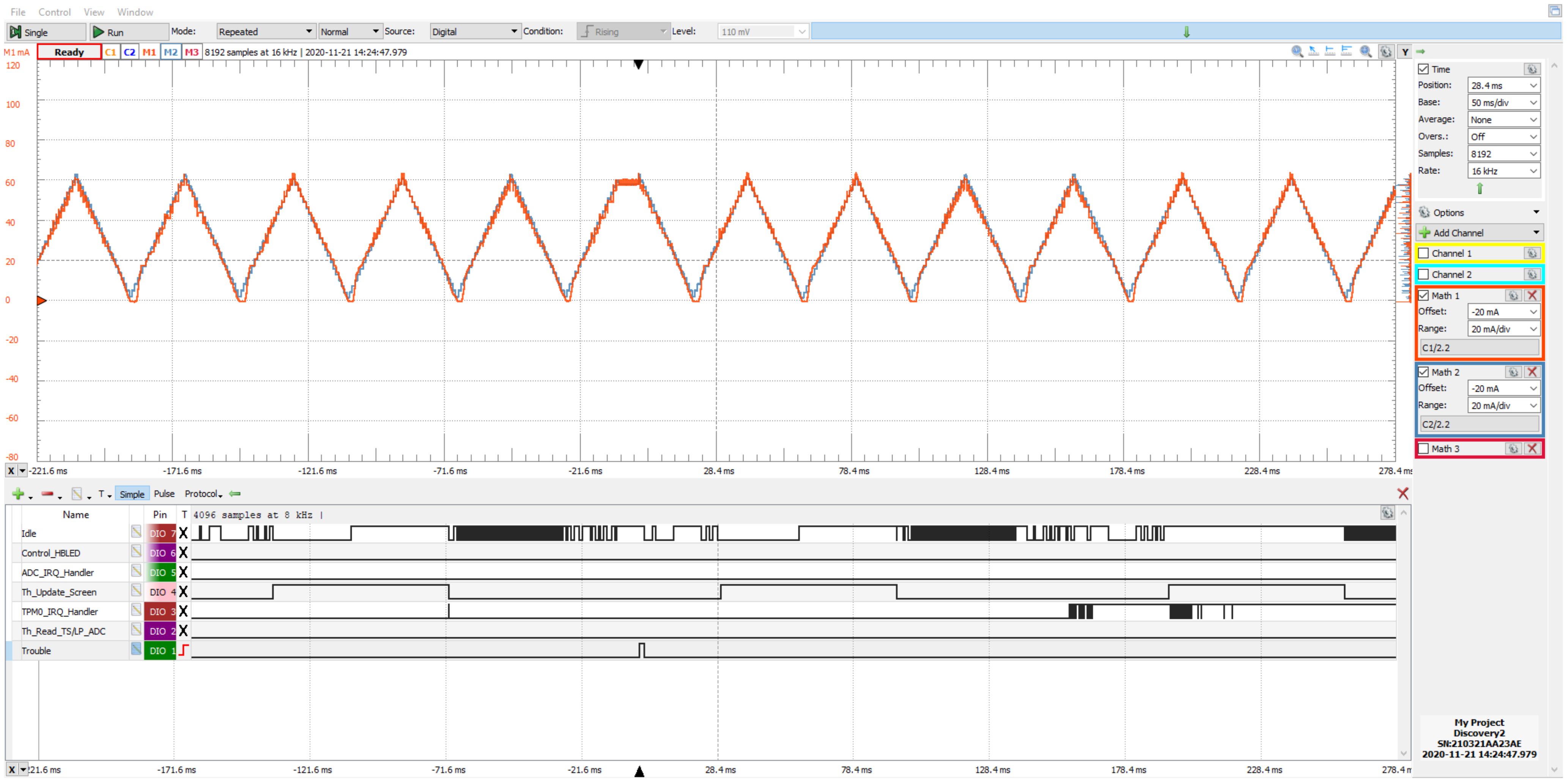
Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

4096 samples at 8 kHz |

My Project
Discovery2
SN:210321AA23AE
2020-11-21 14:26:02.828

Figure 0.1.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

Ready | 8192 samples at 16 kHz | 2020-11-21 14:24:47.979

Time
Position: 28.4 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

Channel 1
Channel 2

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

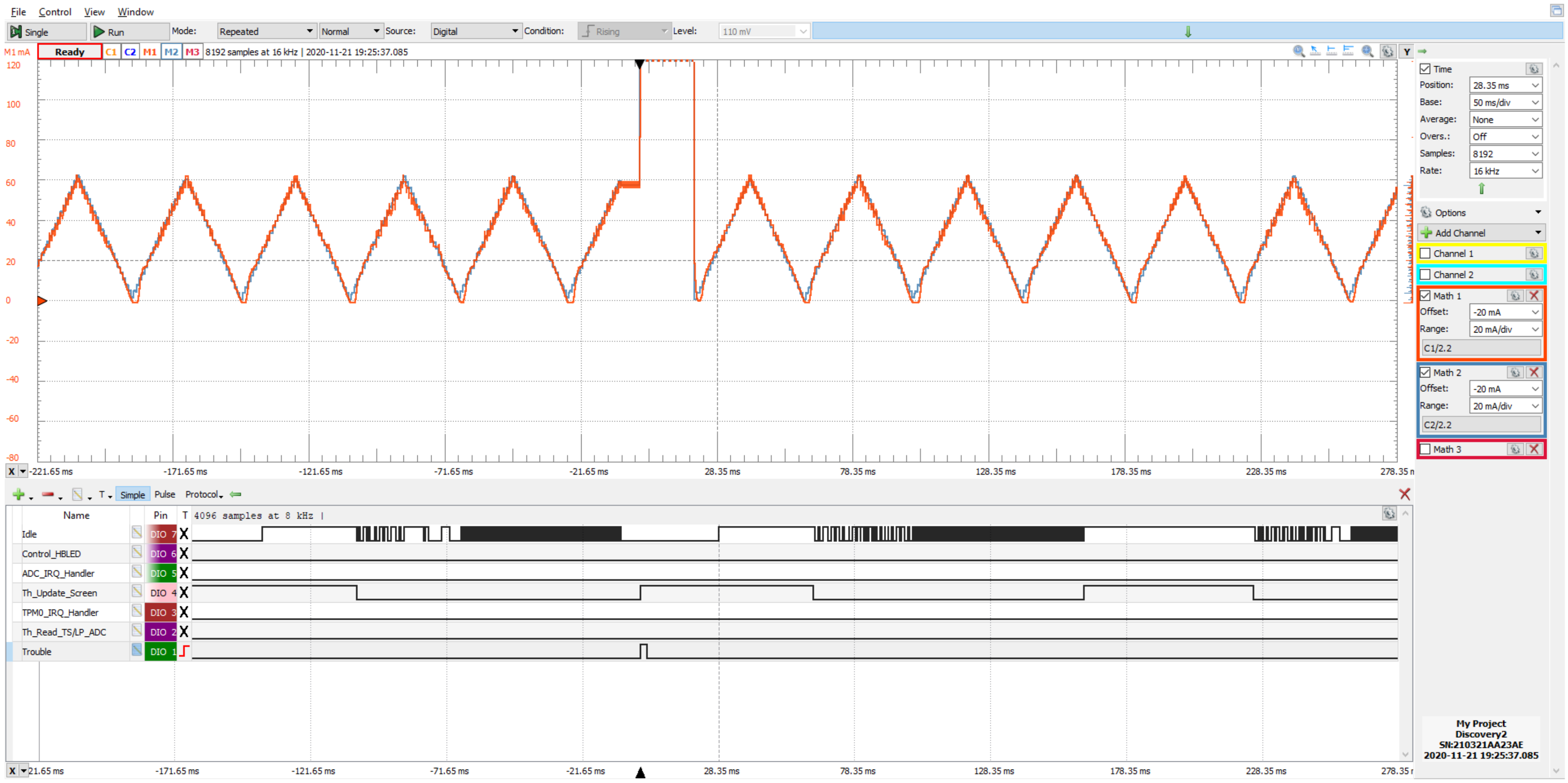
Math 3

```
4096 samples at 8 kHz |
```

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-21 14:24:47.979

Figure 0.3.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

8192 samples at 16 kHz | 2020-11-21 19:25:37.085

Time
Position: 28.35 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

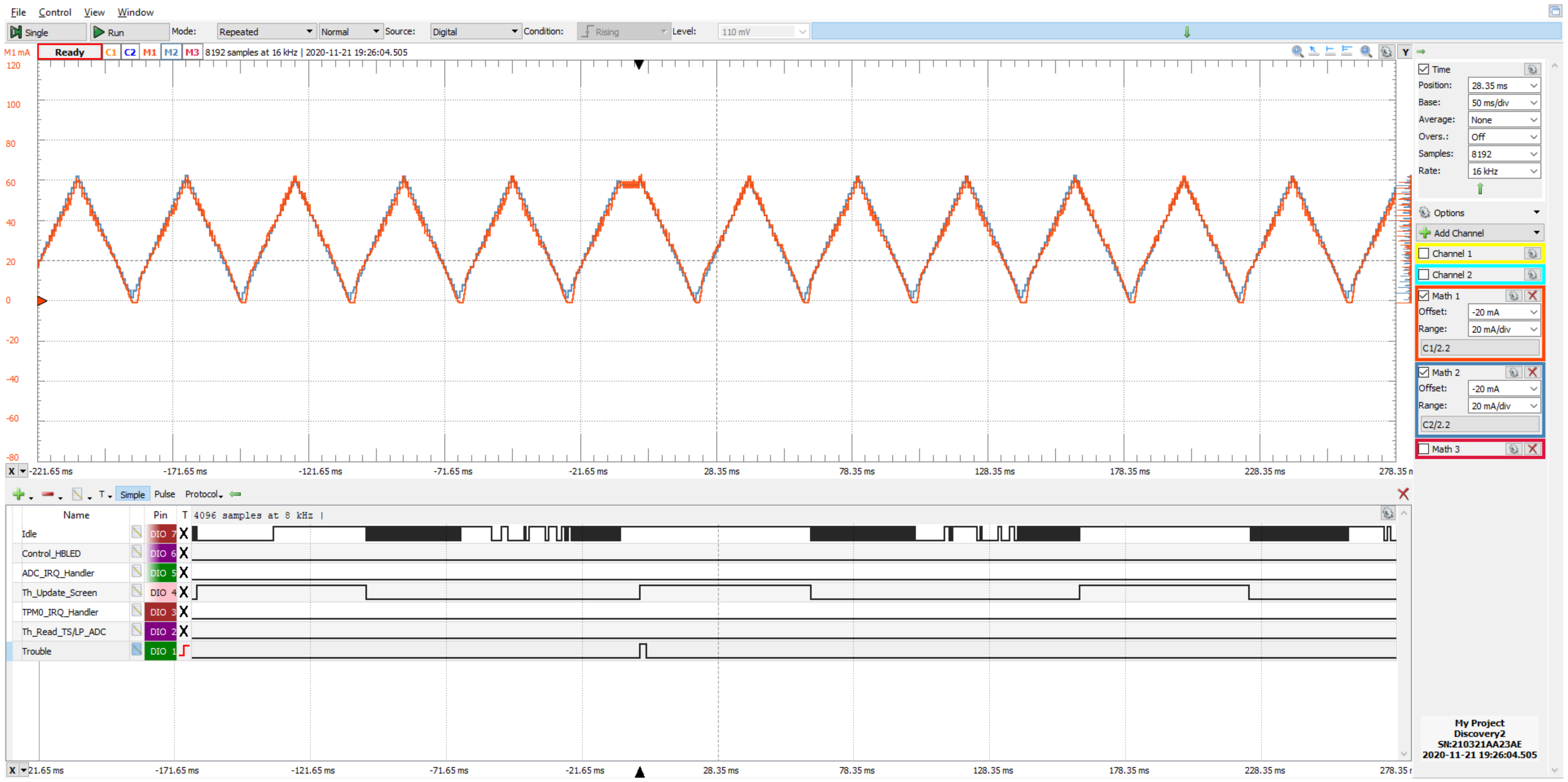
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

4096 samples at 8 kHz |

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-21 19:25:37.085

Figure 1.1.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

Ready | 8192 samples at 16 kHz | 2020-11-21 19:26:04.505

Time
Position: 28.35 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

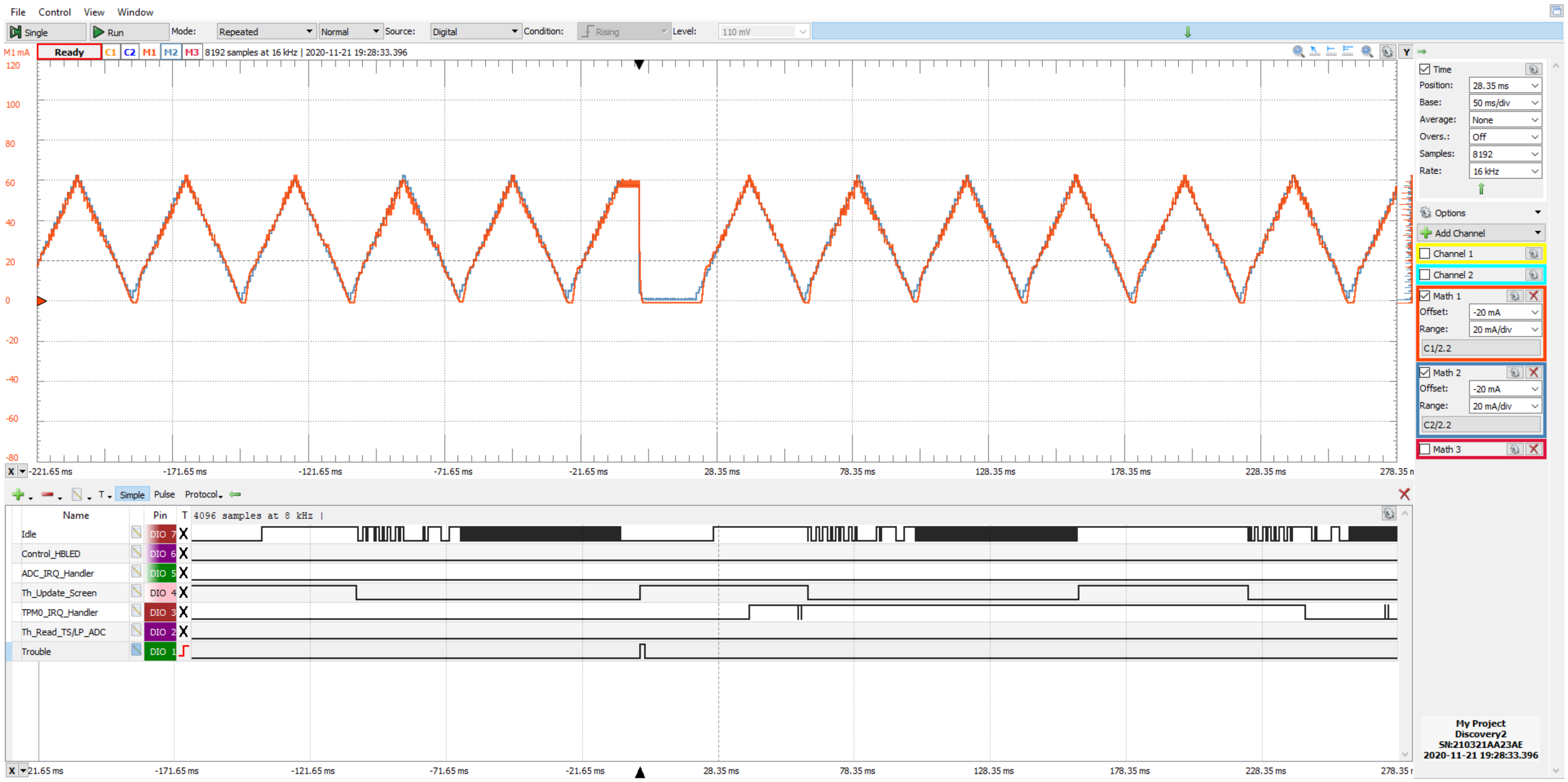
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

4096 samples at 8 kHz |

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-21 19:26:04.505

Figure 1.3.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

Ready | 8192 samples at 16 kHz | 2020-11-21 19:28:33.396

Time
Position: 28.35 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

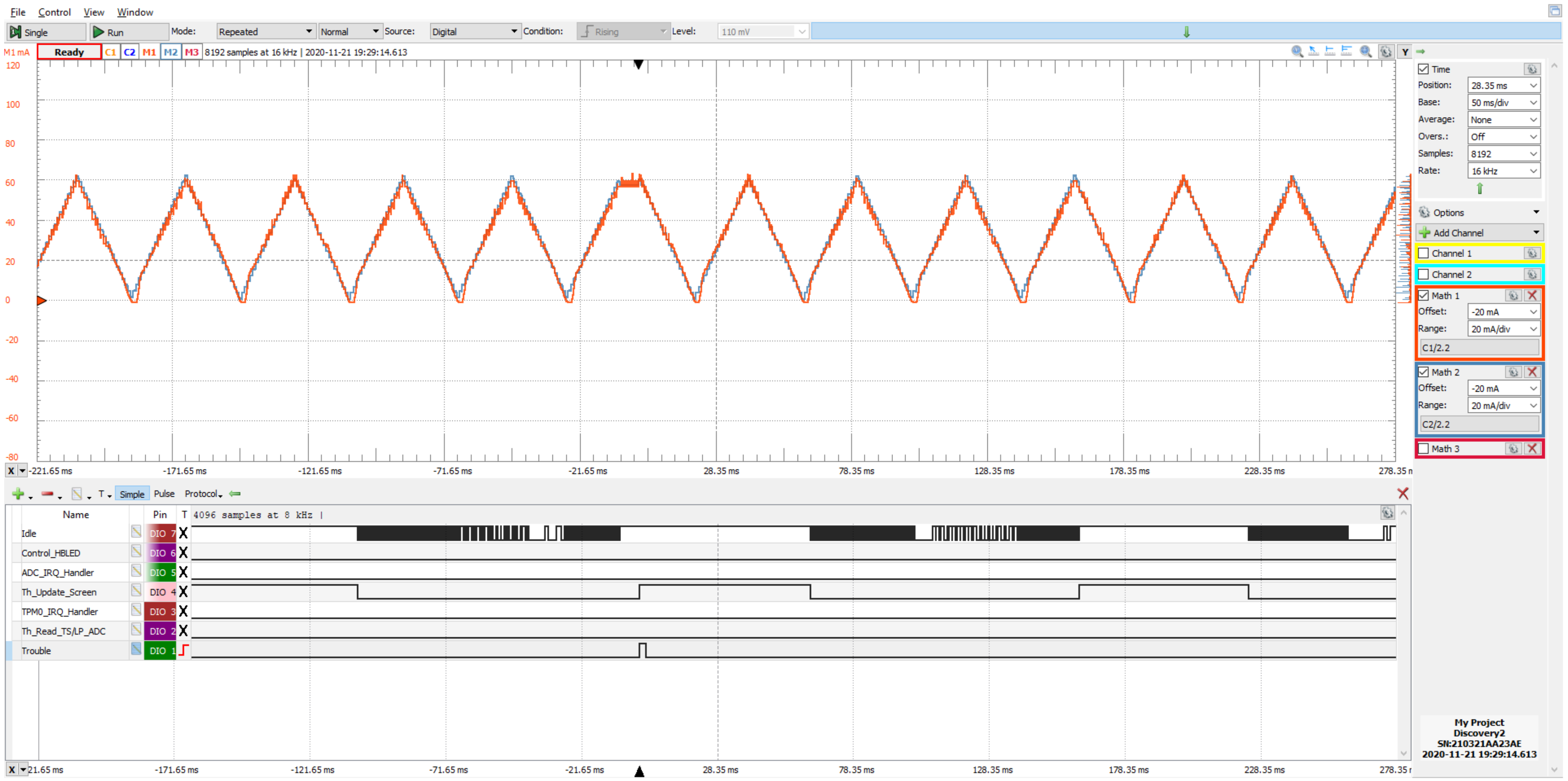
Channel 1
Channel 2
Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2
Math 3

4096 samples at 8 kHz |

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-21 19:28:33.396

Figure 2.1.



Mode: Repeated | Normal
Source: Digital
Condition: Rising
Level: 110 mV

Ready | 8192 samples at 16 kHz | 2020-11-21 19:29:14.613

Time
Position: 28.35 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

Channel 1
Channel 2

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

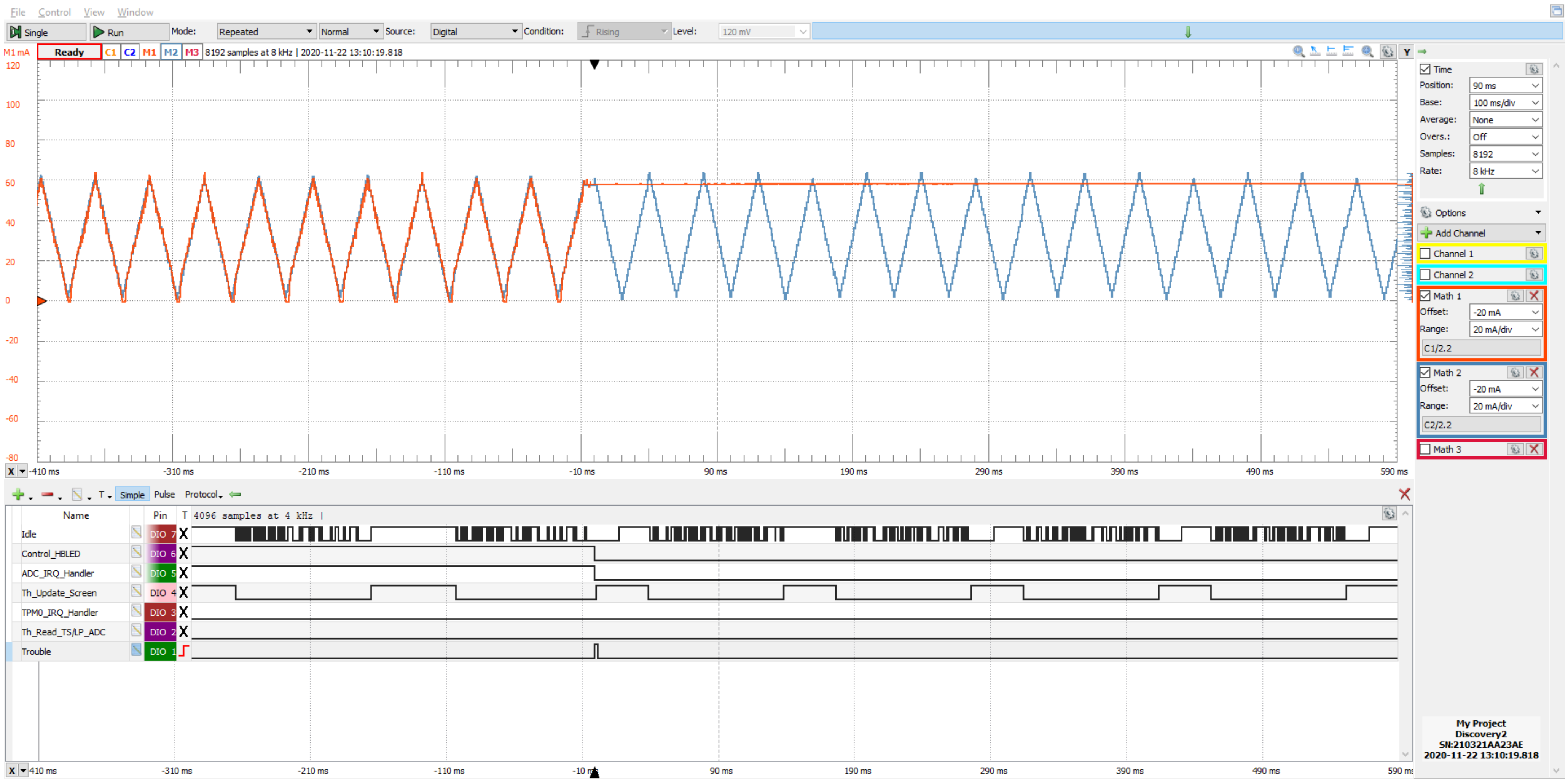
Math 3

4096 samples at 8 kHz |

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-21 19:29:14.613

Figure 2.3.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 120 mV

8192 samples at 8 kHz | 2020-11-22 13:10:19.818

Time
Position: 90 ms
Base: 100 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 8 kHz

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

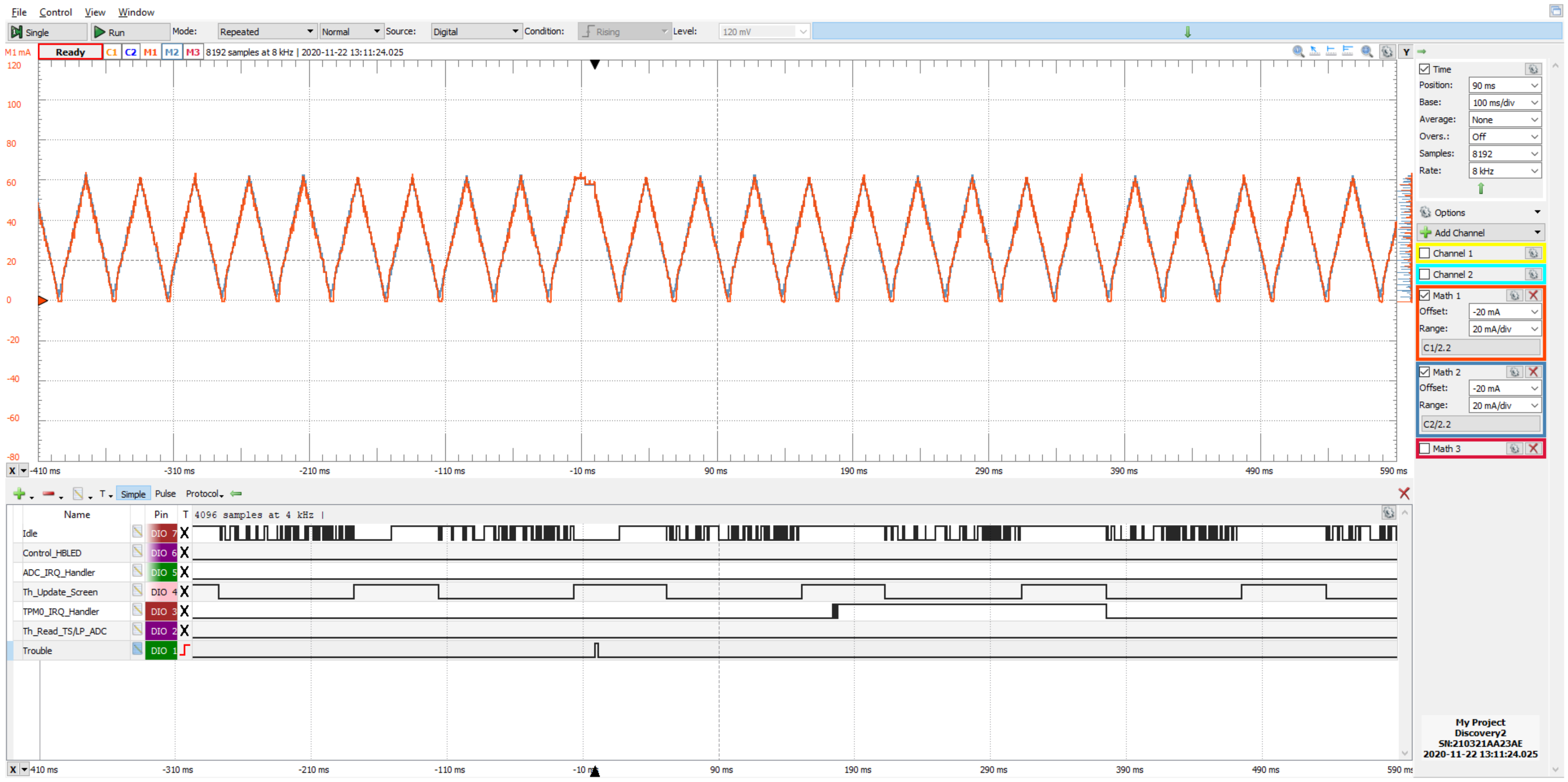
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

4096 samples at 4 kHz |

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-22 13:10:19.818

Figure 3.1.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 120 mV

Ready | 8192 samples at 8 kHz | 2020-11-22 13:11:24.025

Time
Position: 90 ms
Base: 100 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 8 kHz

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

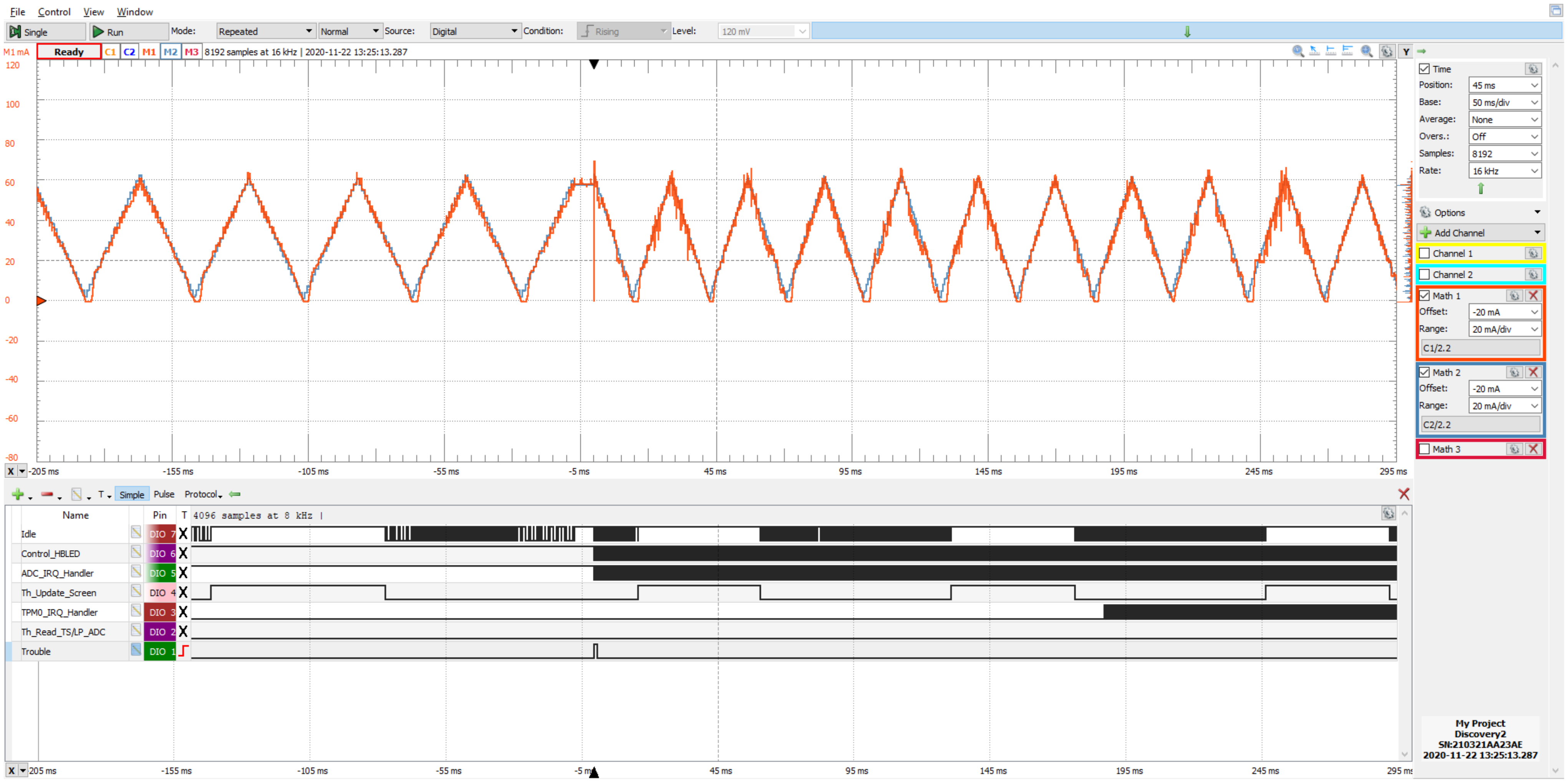
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

```
4096 samples at 4 kHz |
```

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-22 13:11:24.025

Figure 3.3.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 120 mV

Ready | 8192 samples at 16 kHz | 2020-11-22 13:25:13.287

Time
Position: 45 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

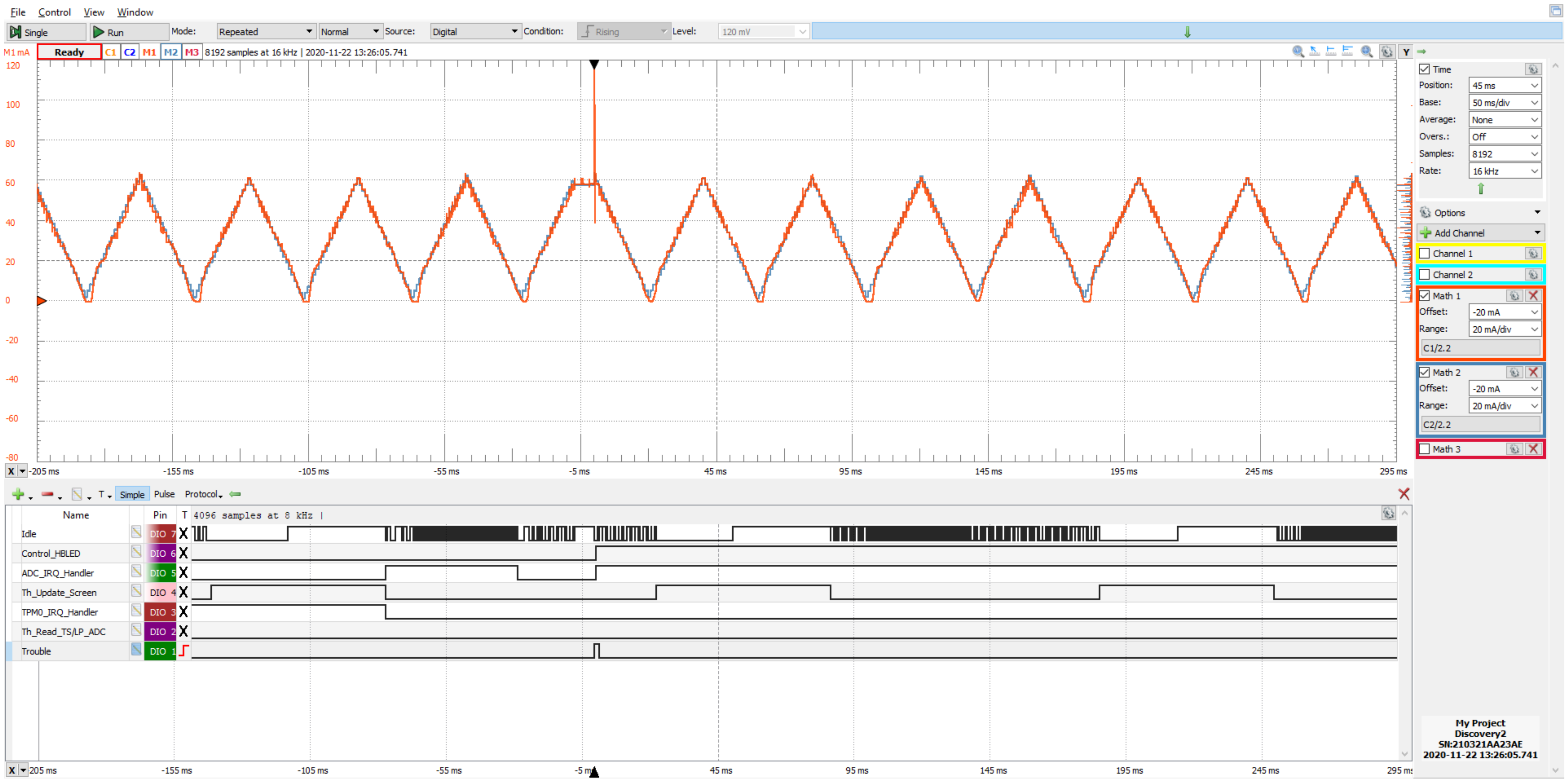
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

```
4096 samples at 8 kHz |
```

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-22 13:25:13.287

Figure 4.1.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 120 mV

Ready | 8192 samples at 16 kHz | 2020-11-22 13:26:05.741

Time
Position: 45 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

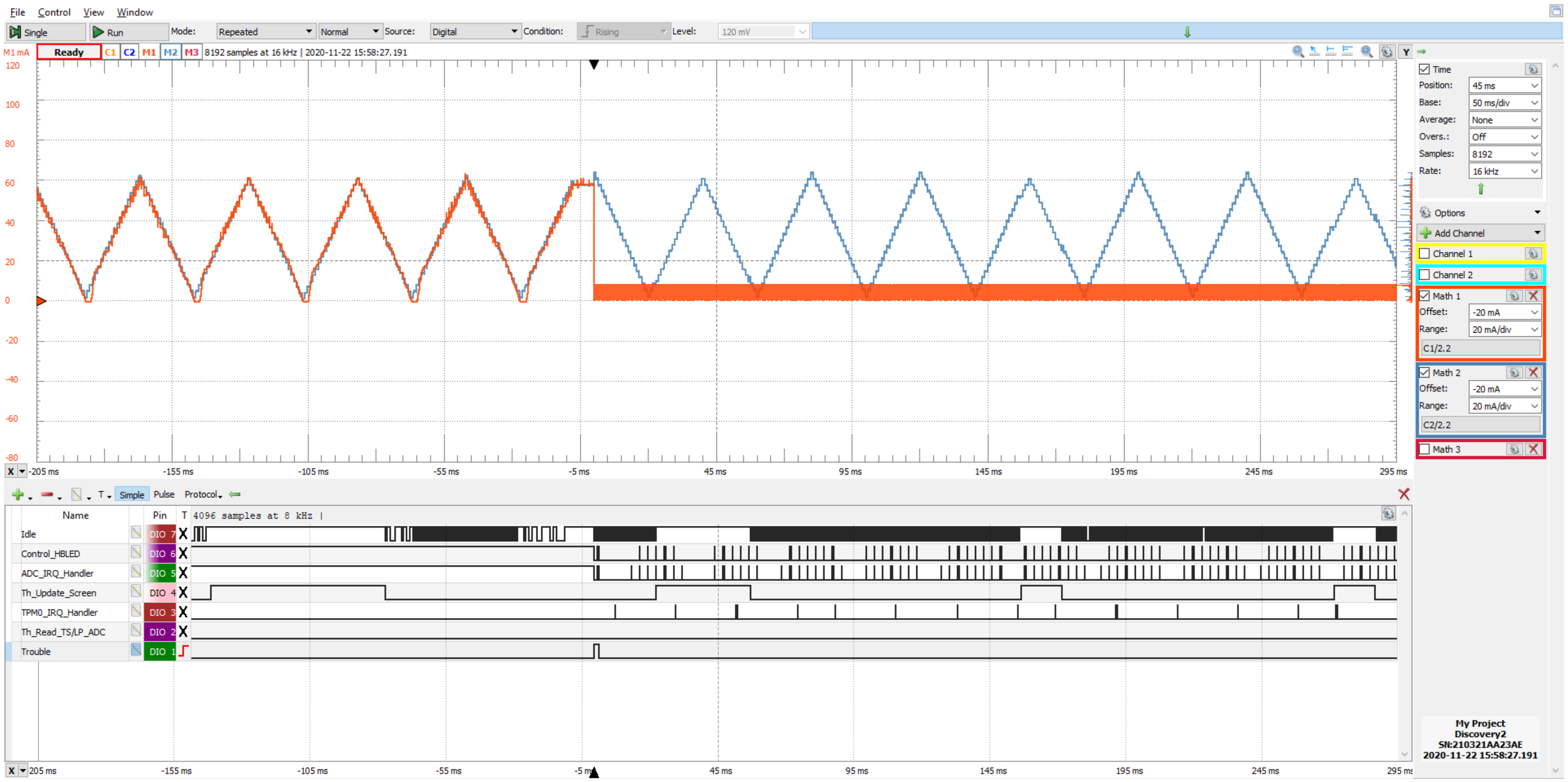
Channel 1
Channel 2
Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2
Math 3

4096 samples at 8 kHz |

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-22 13:26:05.741

Figure 4.3.



Mode: Repeated | Normal

Source: Digital

Condition: Rising

Level: 120 mV

Ready

8192 samples at 16 kHz | 2020-11-22 15:58:27.191

Time	
Position:	45 ms
Base:	50 ms/div
Average:	None
Overs.:	Off
Samples:	8192
Rate:	16 kHz

Math 1	
Offset:	-20 mA
Range:	20 mA/div
	C1/2.2

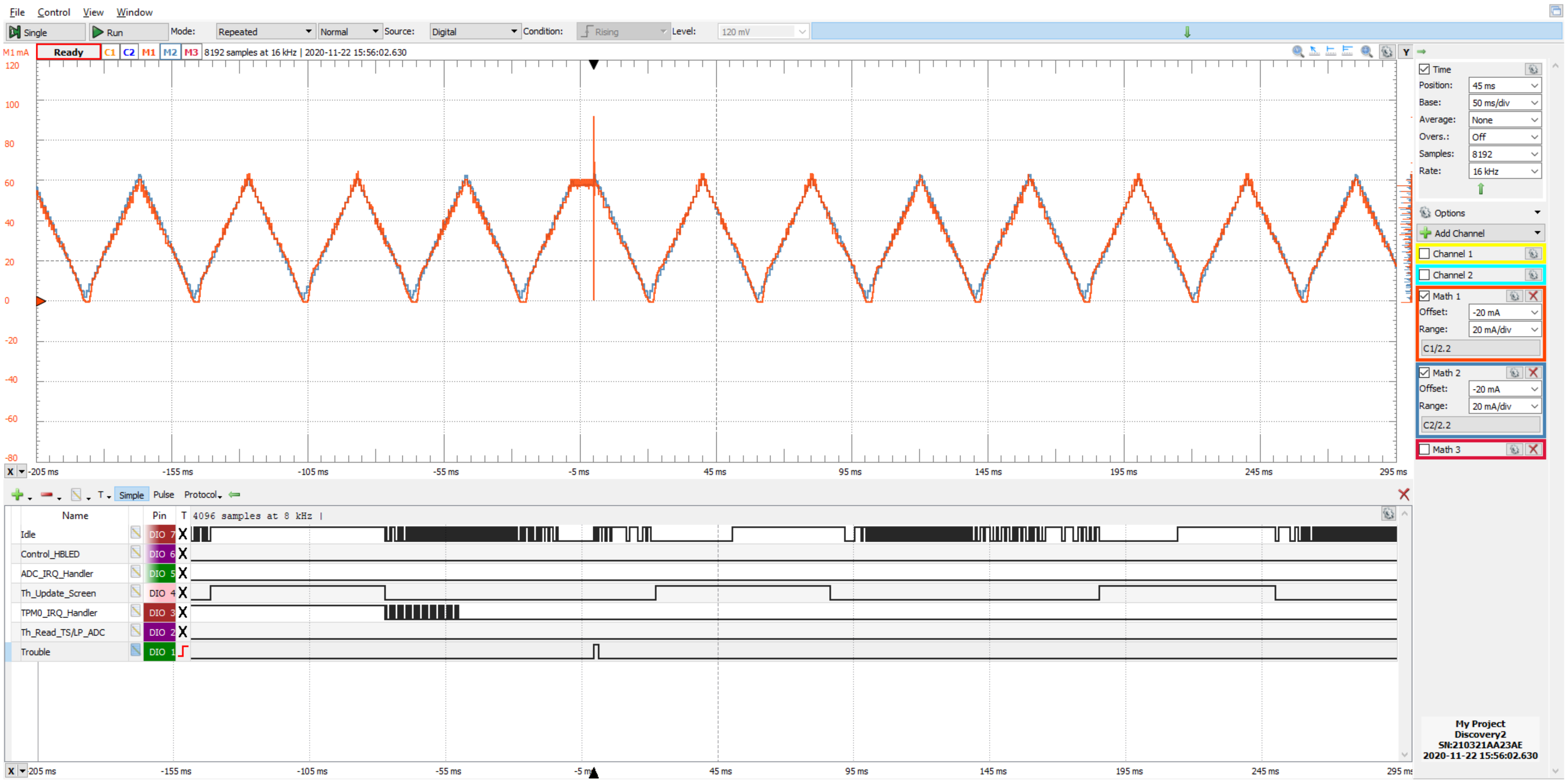
Math 2	
Offset:	-20 mA
Range:	20 mA/div
	C2/2.2

```
4096 samples at 8 kHz |
```

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-22 15:58:27.191

Figure 5.1.



Mode: Repeated Normal Source: Digital Condition: Rising Level: 120 mV

8192 samples at 16 kHz | 2020-11-22 15:56:02.630

Time
Position: 45 ms
Base: 50 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 16 kHz

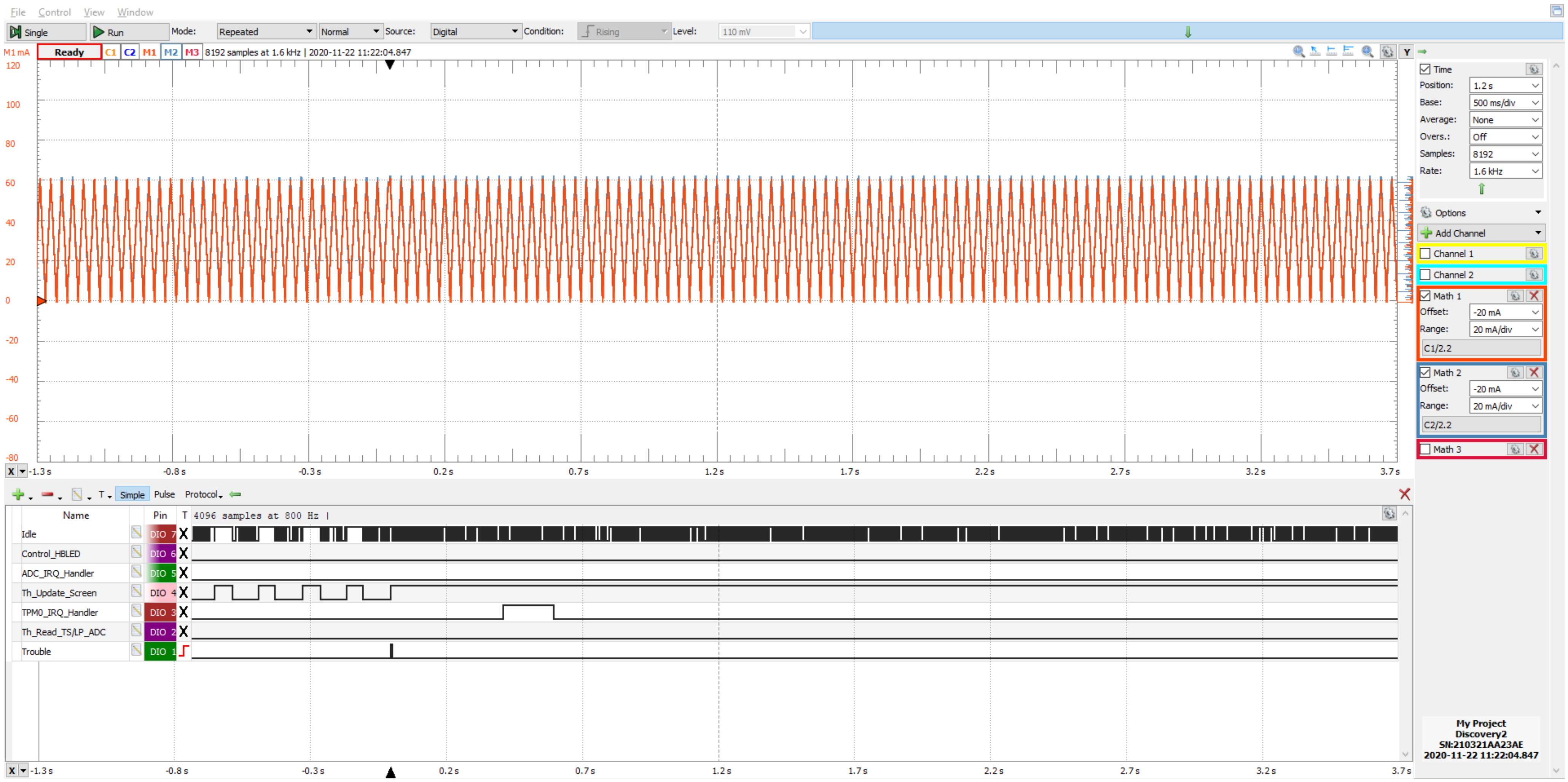
Channel 1
Channel 2
Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2
Math 3

4096 samples at 8 kHz |

Idle DIO 7
Control_HBLED DIO 6
ADC_IRQ_Handler DIO 5
Th_Update_Screen DIO 4
TPM0_IRQ_Handler DIO 3
Th_Read_TS/LP_ADC DIO 2
Trouble DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-22 15:56:02.630

Figure 5.3.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

8192 samples at 1.6 kHz | 2020-11-22 11:22:04.847

Time
Position: 1.2 s
Base: 500 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 1.6 kHz

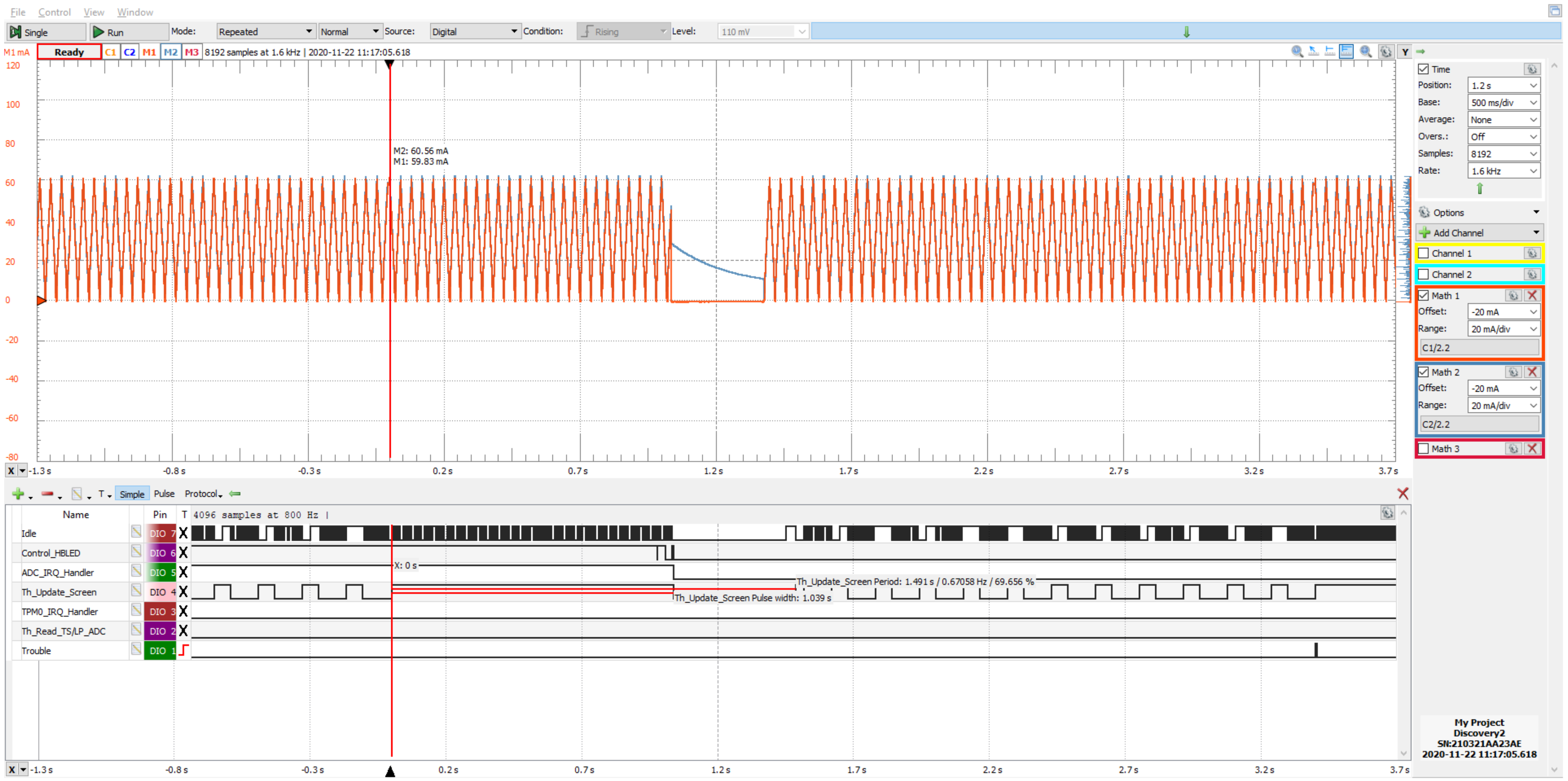
Channel 1
Channel 2
Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2
Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2
Math 3

4096 samples at 800 Hz |

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

My Project
Discovery2
SN:210321AA23AE
2020-11-22 11:22:04.847

Figure 6.1.



Mode: Repeated | Normal | Source: Digital | Condition: Rising | Level: 110 mV

Ready | 8192 samples at 1.6 kHz | 2020-11-22 11:17:05.618

M2: 60.56 mA
M1: 59.83 mA

Time
Position: 1.2 s
Base: 500 ms/div
Average: None
Overs.: Off
Samples: 8192
Rate: 1.6 kHz

Math 1
Offset: -20 mA
Range: 20 mA/div
C1/2.2

Math 2
Offset: -20 mA
Range: 20 mA/div
C2/2.2

4096 samples at 800 Hz

Name	Pin
Idle	DIO 7
Control_HBLED	DIO 6
ADC_IRQ_Handler	DIO 5
Th_Update_Screen	DIO 4
TPM0_IRQ_Handler	DIO 3
Th_Read_TS/LP_ADC	DIO 2
Trouble	DIO 1

X: 0 s

Th_Update_Screen Period: 1.491 s / 0.67058 Hz / 69.656 %

Th_Update_Screen Pulse width: 1.039 s

My Project
Discovery2
SN:210321AA23AE
2020-11-22 11:17:05.618

Figure 6.3.