ESA 2025 FINAL PROJECT: SHIELDS UP!

CONTENTS

OVERVIEW

The starter program uses the buck converter to flash the shield's white LED as a strobe light with adjustable period. The LCD and touchscreen provide a user interface for changing operating parameters. The LCD also plots the measured LED current and requested current setpoint like an oscilloscope. Tilting the board changes the flash period. The program shares the ADC between the buck converter controller and the touchscreen code while still maintaining correct timing for the buck converter controller**.** The program uses the RTXv5 real-time operating system.

There are two parts to this project:

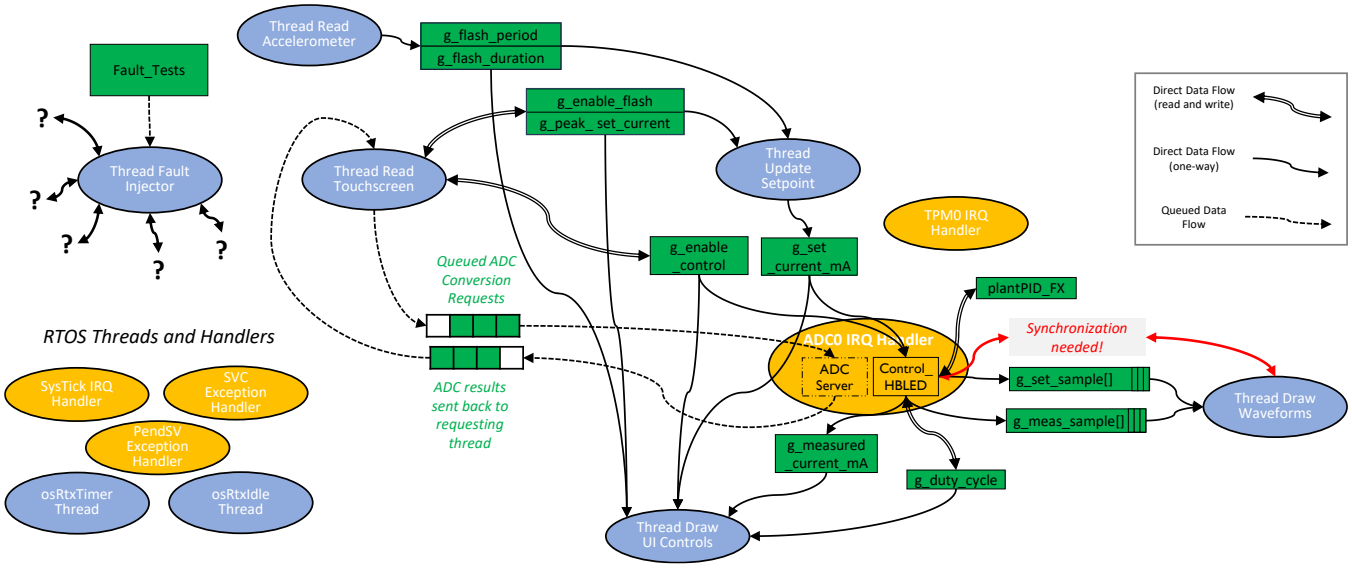
- **Part 1: Modify the program to synchronize different processes in the system** using shared variables, interrupts and RTOS mechanisms.
- **Part 2: Modify the program to help protect it against internal and external faults** such as unexpected inputs, coding and design errors, explicit attacks, device failures, electromagnetic interference, etc. Note that this is not perfect protection against every possible fault, but it improves the system's robustness and reliability. ECE 460 students must address one fault, while ECE 560 students must address two. These faults are identified in a table later in this assignment.

Many types of faults may be triggered; refer to the code in fault.c for details. You must modify the program to detect and handle them. If feasible, handle them so the program keeps running without restarting. Otherwise, handle the fault by restarting the program (e.g. with the watchdog timer (COP)). Document in your report how you tackled the faults.

Please refer to the Submission Information section for submission details.

Start with the code on the GitHub repository in Final_Project\Project_Base. This section provides a quick overview; there is a more detailed explanation later in the *Details of Starter Code* section.

OVERALL SYSTEM ARCHITECTURE

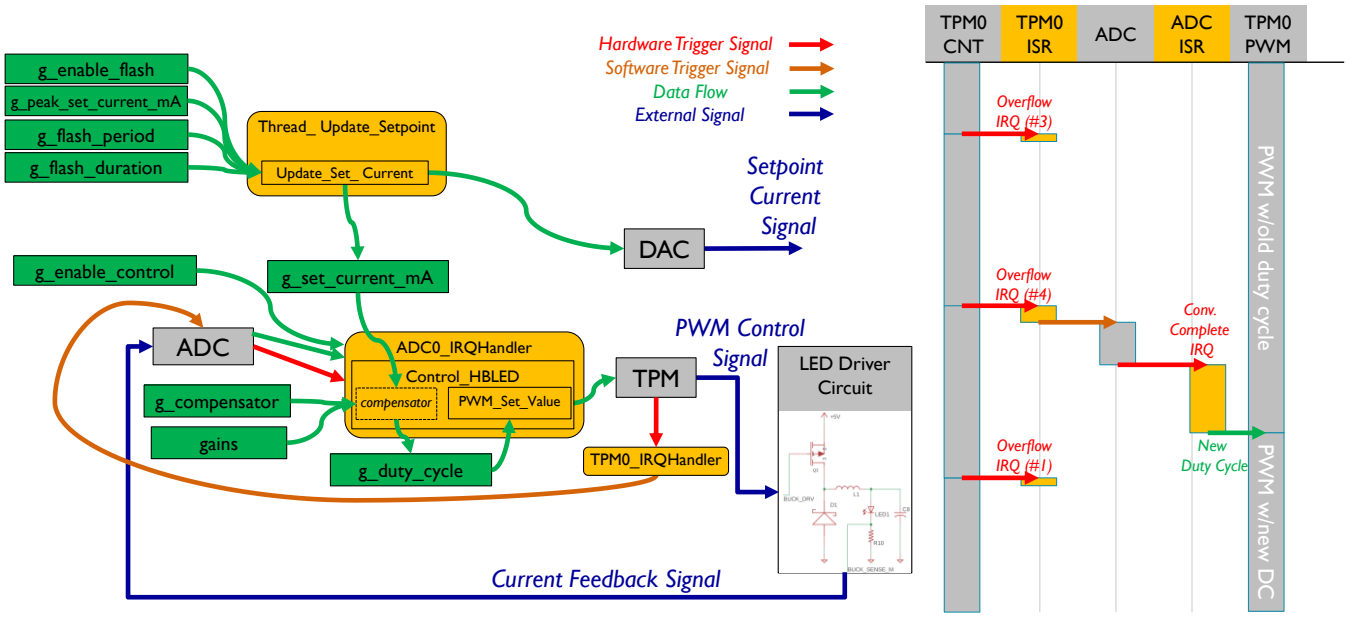


There are several threads and IRQ Handlers implementing key parts of the system.

- **Thread_Update_Setpoint** adjusts the HBLED's current setpoint and causes flashing.

- **Thread_Read_Touchscreen** (Thread_Read_TS) calls the function LCD_TS_Read to see if the screen is touched. If the screen is touched, then LCD_TS_Read will use the ADC to determine where the screen is pressed. Thread_Read_TS then calls UI_Process_Touch to update various control parameters appropriately.
- **Thread_Draw_UI_Controls** renders control parameters and other operating data on the LCD.
- **Thread_Draw_Waveforms** draws an oscilloscope-style plot of the setpoint (requested) current and actual measured current.
- **Thread_Read_Accelerometer** measures the roll of the FRDM board to set the flashing period for the HBLED.
- **ADC0_IRQHandler** implements a server to share the ADC between the HBLED control system (implemented by calling Control_HBLED) and any other ADC use requests (Thread_Read_TS in this program). Note that code related to controlling the HBLED has been moved into different files compared with PHW3: the definitions previously in HBLED.h have been moved to control.h, and the control functions in main.c have been moved to control.c.
- **TPM0_IRQHandler** triggers an ADC conversion of the current sense channel for the buck converter once for every SW_CTL_FREQ_DIV_FACTOR times this handler runs. This allows a higher switching frequency than the control loop frequency, saving CPU time.
- **Thread_Fault_Injector** periodically injects a fault into the system after updating the LCD with the fault test number and raising a debug signal (DBG_FAULT). This debug signal is helpful for triggering your scope.

LED CURRENT CONTROLLER



The controller is similar to the one used in Practical Homework 3. The system uses a high switching frequency to reduce ripple, but a lower control loop frequency reduces the CPU load. This frees up time for other processing, such as the user interface.

The code uses TPM0 to generate the PWM signal BUCK_DRV. TPM0 also generates an interrupt request on overflow. The TPM0_IRQ handler will sometimes trigger the ADC with software to start a conversion of the current sense voltage. This conversion will be triggered every SW_CTL_FREQ_DIV_FACTOR times the handler runs, leading to a control loop frequency of $f_{switching}$/ SW_CTL_FREQ_DIV_FACTOR. The ADC's conversion complete interrupt causes the ADC0_IRQ handler to run.

USER INTERFACE

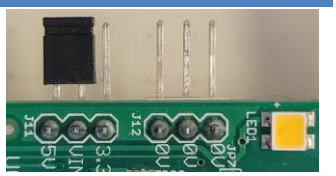
A user interface graphically displays the setpoint current (blue) and the measured current (yellow). The numerical value of key parameters is also displayed.

Normally a parameter in green can be changed by pressing the parameter name and then using the grey slider control at the bottom of the display to increase the value (press on the right side of the slider) or decrease it (left side). The amount of change increases as you press closer to the left or right edge of the screen. However, this code is not working correctly yet. So don't expect the touchscreen to operate correctly. You do not need to use it for this project.

FAULT INJECTION

A thread (Thread_Fault_Injector) will inject faults into the program periodically from a list in the array Fault_Tests. This thread will indicate the test number on the screen and pulse a digital debug output (called Fault) just before injecting the fault. Trigger the scope on this fault signal to see exactly how the system responds to the fault.

PREPARATION



- ❑ Connect a shorting jumper on J11 across the pins marked VIN and 5V as shown in the photo. This provides power at 5 V to the switching converter. If your code malfunctions and overdrives the LED, you can pull off the jumper quickly to remove power.
- ❑ If possible, use the CMSIS-DAP.S19 debugger. You will need to observe and change variables, and this debugger lets you do it as the program runs.

SHIELD AND ANALOG DISCOVERY CONNECTIONS

DIRECT CONNECTIONS

The Analog Discovery 2/3 is directly connected via J6 to key signals on the expansion shield, shown in the table below.

AD2/3 Signal	Group	MCU Port Bit	Shield Signal	Convenience Name in debug.h	Description
DIO0	Software Debug Signals	PTD0	DBG_0	DBG_TPM_ISR_POS	TPM0 ISR (IRQ Handler) active
DIO1		PTD2	DBG_1	DBG_ADC_ISR_POS	ADC0 ISR active
DIO2		PTD3	DBG_2	DBG_LOPRI_ADC_POS	Low-priority ADC conversion pending (requested but not yet completed)
DIO3		PTD4	DBG_3	DBG_TUSP_POS	Thread_Update_Setpoint active
DIO4		PTB8	DBG_4	DBG_CONTROLLER_POS	Function Control_HBLED active
DIO5		PTB9	DBG_5	DBG_PENDING_WVFM_POS	Scope update pending (sample buffers hold waveforms which haven't been drawn yet)
DIO6		PTB10	DBG_6	DBG_T_DRAW_WVFMS_POS	Thread_Draw_Waveforms active
DIO7		PTB11	DBG_7	DBG_T_DRAW_UI_CTLS_POS	Thread_Draw_UI_Controls active
DIO8		PTE2	DBG_8	DBG_BLOCKING_LCD_POS	Thread blocking on LCD_mutex
DIO9		PTE3	DBG_9	DBG_LCD_COMM_POS	LCD communication active. **DEBUG_START/STOP macros to be added by student.**
DIO10		PTE1	DBG_10	DBG_FAULT_POS	Fault injected (1-3 ms long pulse)
DIO11		PTE4	DBG_11	DBG_IDLE_LOOP	Optional, can be toggled by idle thread's loop to show CPU free time but will create noise in DAC output. Initially **commented out**, check RTX_Config.c to see.
DIO12	SMPS	n/a	SMPS_DRV	n/a	SMPS PWM drive signal

ANALOG LED CURRENT MONITORS

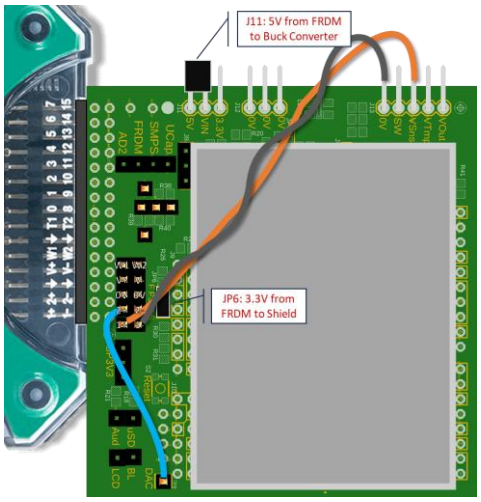


Figure 1. Overview of analog input connections (channels 1+, 1-, 2+) for viewing with Scope tool.

The Waveforms Scope tool is helpful for evaluating the LED current controller behavior when channels 1+ and 2+ are connected as shown in Figure 1. You can determine the LED current by measuring the voltage across R10. This signal is brought out to the VSns signal on J13.

Figure 2 and Figure 3 show how to connect to VSns on J13 to the scope's channel 1 positive differential input (1+).

To simplify development, the code generates an analog voltage to indicate the desired output current (set point). This is done by calling the function **Set_DAC_mA**. This voltage will be scaled so that it matches the desired voltage across R10. This will allow you to compare the voltage across R10 directly with the DAC output. The code uses the digital to analog converter (DAC0) to indicate the set point (desired current output) of the system. Each 2.2 mV represents 1 mA of current. For example, the DAC should indicate a set point of 100 mA with a voltage of 220 mV.

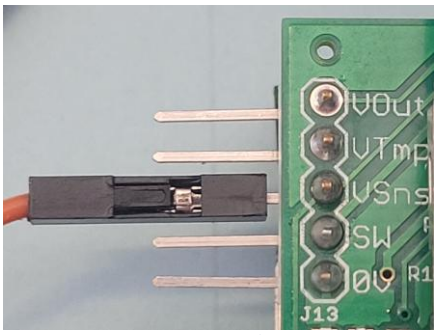


Figure 2. Connect VSns (voltage across current sense resistor) to channel 1+.

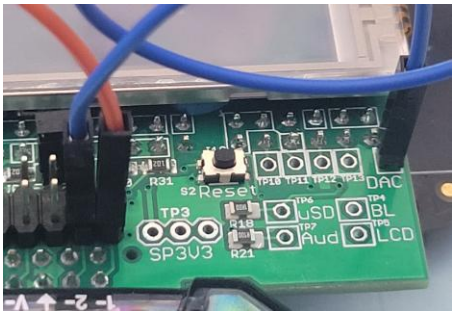


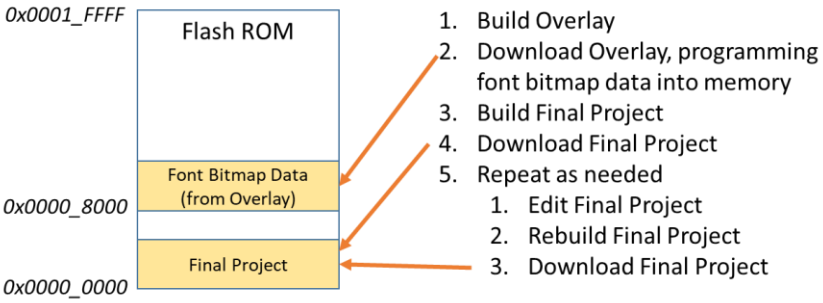
Figure 3. Connect DAC output (setpoint current reference) to AD2/3 channel 2+.

To monitor the voltage with the AD2/3, connect the scope's channel 2 positive differential input (2+) to the DAC output at TP8, as shown in **Error! Reference source not found.**. The WaveForms program can use the voltage across R10 to calculate and display the actual current. The Project V14.dwf3work workspace has been configured with math channels M1 and M2 to display the current and the current setpoint.

MEMORY OVERLAY SUPPORT

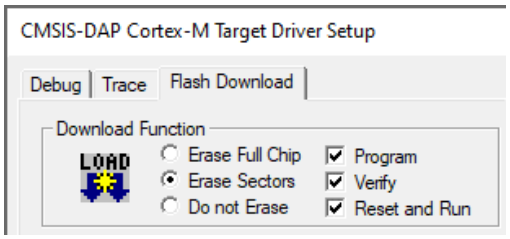
The free version of MDK-ARM limits object image size to 32 kB, even though the MCU has 128 kB of flash ROM. To keep from reaching 32 kB, some large read-only data (font bitmaps) has been moved out of the application project into a separate project (ESA-25\Tools\TestCode\Overlay). The program's main function uses LCD_Text_Init for initialization and to confirm the font data is present. If missing, the program will signal the error by repeatedly flashing the red LED with two quick blinks and then a pause.

As seen in this diagram, in steps 1 and 2 you'll build and download the Overlay project to your KL25Z first to install the Overlay ROM data. In steps 3 and 4, you'll build your application project (e.g. the final



project) and download it *without erasing the overlay ROM data.* As you edit the project, you'll build it and download it again, as shown in steps 5.1, 5.2 and 5.3. You do not need to repeat steps 1 and 2 (unless you erase the full chip).

How does this work? The MCU's Flash ROM has 128 sectors (1 kB each) which can be individually erased and programmed. The final project's download settings are configured so that only the sectors which are to be used are erased and programmed, rather than all sectors (the full chip). This means that the Overlay data will remain in ROM as long as you use a Flash Download setting of **Erase Sectors** instead of **Erase Full Chip** as shown in the figure and don't reprogram the sectors holding the data.



Using the overlay places the data for three fonts in overlay ROM (starting at address 0x00008000). It also puts a trivial, expendable program into ROM (starting at 0x00000000) which will be overwritten safely by your application program.

REQUIREMENTS

SYNCHRONIZATION

You will need to synchronize different processes in the system.

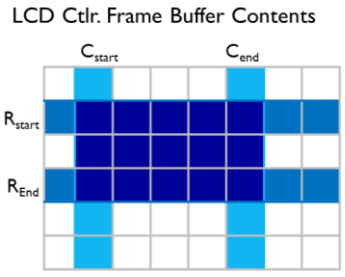
MUTUAL EXCLUSION FOR LCD UPDATE OPERATIONS

Two threads update the LCD: Thread_Draw_Waveforms and Thread_Draw_UI_Controls. If instructions for certain LCD operations from these two threads are interleaved in certain ways, the information sent to LCD controller will be corrupted, and the wrong pixels will be displayed. These LCD operations form **critical sections** of code.

To prevent corruption, the threads must synchronize their LCD operations to ensure **mutually exclusive** execution of the critical sections for the LCD update. That is, don't start executing an LCD critical section if another thread has **started but not finished** an LCD critical section. The base code uses an OS mutex called **LCD_mutex** to do this. The threads follow two rules: acquire LCD_mutex before accessing the LCD, and release it after accessing the LCD. Note that the fault injection thread does not use LCD_mutex and corrupts what is displayed.

CRITICAL SECTIONS OF CODE FOR LCD OPERATIONS

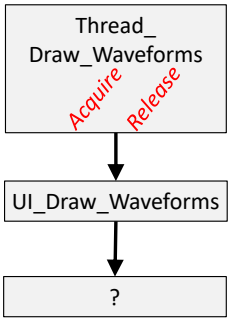
LCD image updates are done by drawing rectangles. The MCU requests the LCD controller to fill in a rectangle (defined by a range of columns and a range of rows) pixel by pixel with a list of colors. This takes three LCD commands as shown. The command to write pixel colors can have a very long list of colors, and is usually much longer than the range-setting instructions. We will call these three transactions for drawing one rectangle a **rectangle transaction sequence.**



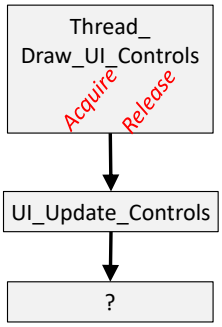
Set Column Range	Set Row Range	Write Pixel Colors

Letting one thread start an LCD command before another thread has finished its ongoing **rectangle transaction sequence** will corrupt the image. Therefore, the code for each rectangle transaction sequence is a **critical section**, starting with the command setting that rectangle's column range and finishing after that rectangle's last pixel color has been sent.

FIRST STEP: USE MUTEX AT HIGH LEVEL




```
void Thread_Draw_Waveforms(
        void * arg) {
  …
  // Regular operation
  while (1) {
    osDelayUntil(tick);
    osMutexAcquire(LCD_mutex,
      osWaitForever);
    UI_Draw_Waveforms();
    osMutexRelease(LCD_mutex);
    …
  }
}
```




```
void Thread_Draw_UI_Controls(
        void * arg) {
  …
  // Regular operation
  while (1) {
    osDelayUntil(tick);
    osMutexAcquire(LCD_mutex,
      osWaitForever);
    UI_Update_Controls(0);
    osMutexRelease(LCD_mutex);
    …
  }
}
```

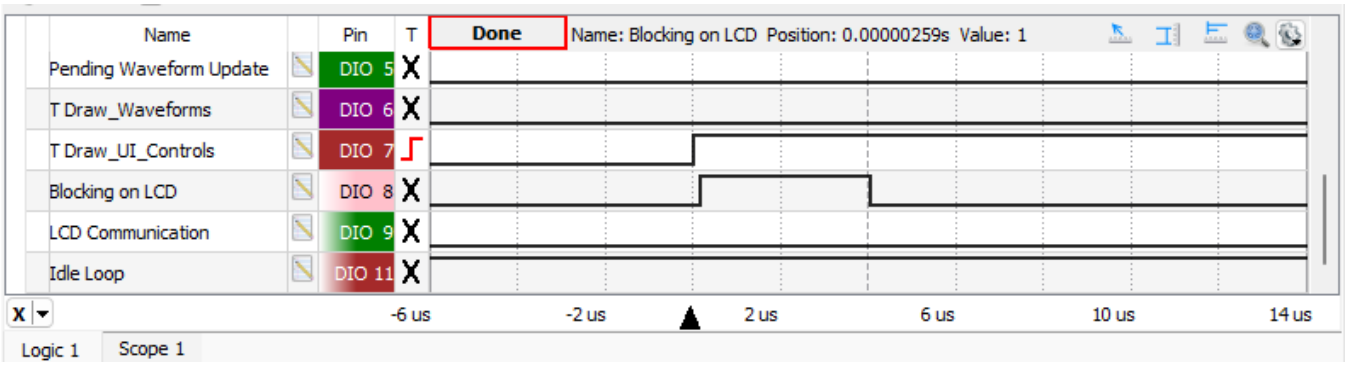
The code for Thread_Draw_Waveforms and Thread_Draw_UI_Controls performs initialization and then repeats a loop periodically. Without more analysis, we don't know how many critical sections are in the LCD access code, or where they start and finish. For simplicity, the base code acquires the mutex at the beginning of each loop iteration and only releases it at the end. This definitely protects all the critical sections, but probably holds the mutex for longer than necessary.

MEASURING BLOCKING TIME

To help visualize system behavior, the base code sets a debug output signal (specified by DBG_BLOCKING_LCD_POS in debug.h) to 1 whenever a thread is trying to access and possibly blocking on (waiting for) the LCD_Mutex, and clears it to 0 otherwise. We will call this the LCD Blocking signal.

The Waveforms Logic analyzer tool can display this signal to help understand system behavior, letting you measure the pulse width to find the blocking time. One of the smart cursors available (marked in blue) will measure pulse width, period, and duty cycle.

The diagram below shows Thread_Draw_UI_Controls starts running at 0 us, tries to acquire the LCD mutex, and spends about 4 us before successfully continuing. This time consists of time overhead for the OS call, and may also include blocking time if the mutex is not immediately available.



This blocking time will vary as the program runs, so you'll need to gather statistics. In the Logic window, do the following:

- Open or select the Logic tool window.
- Make sure that the DIO input with the LCD Blocking signal is listed on the left side of the window. If it is not there, add it as a signal:

  - Select the green + and select Signal.
  - In the Add signal dialog box:
    - Select the DIO signal connected to the LCD Blocking signal.
    - Type a descriptive name (e.g. LCD Blocking) in the Name box.
    - Click Add.
- Confgure the analyzer to trigger on the rising edge of the LCD Blocking signal.
- Select View and ensure Measurements is checked.
- In the Measurements window pane, select Add.
  - In the Add measurement dialog box:
    - Select the LCD Blocking signal in the left column and PosWidth in the right column.
    - Press Add.
    - Press Close.
  - Select Show and ensure Mean, Minimum and Maximum are checked.
  - Select the gear icon (for settings) and ensure Multiple Acquisitions is checked.
- You can now run the logic analyzer by pressing the Run button. Timing measurements will be displayed in the Measurements window.
- There are limitations to this measurement approach because the tool captures a burst of samples. The sample count and sampling rate are displayed above the trace window:

  16384 samples at 160 kHz | 2024-11-13 10:59:43.326.827.670 (16/16bit)

  These are controlled indirectly by the analyzer's time base control (the Base field in the toolbar).
  - Sampling introduces some errors. In this example, the sampling rate of 160 kHz indicates a sampling period of 6.25 µs. Pulses shorter than this period may not be captured. Pulses which are captured may have width measurement errors of up to two sampling periods (e.g. 12.5 µs in this example).
  - Only pulses which start and end within a single data acquisition (data capture) will be measured. A 100 ms pulse will be ignored if only 10 ms are captured at a time.
  - Experiment with changing the time base to ensure you are not missing long pulses, but still using a sampling rate providing good accuracy . The Maximum PosWidth measurement for the LCD blocking pulse should be over 20 ms.

TIMING ANALYSIS

1. Provide two screenshots of the logic analyzer with trace and measurement windows showing each thread accessing the mutex (i.e. there must be a pulse on the LCD blocking signal).
2. Measure and provide these statistics for the given threads.

	Thread_Draw_Waveforms			**Thread_Draw_UI_Controls**		
	Minimum	**Average**	**Maximum**	**Minimum**	**Average**	**Maximum**
Time accessing or blocking on LCD mutex						
Thread execution time (from rise to fall for thread's debug signal)						

3. How are the timing statistics related to each other? Are any statistics unexpected? Note that the minimum blocking time is the time overhead required for each possible mutex access, assuming the LCD_mutex it is available.

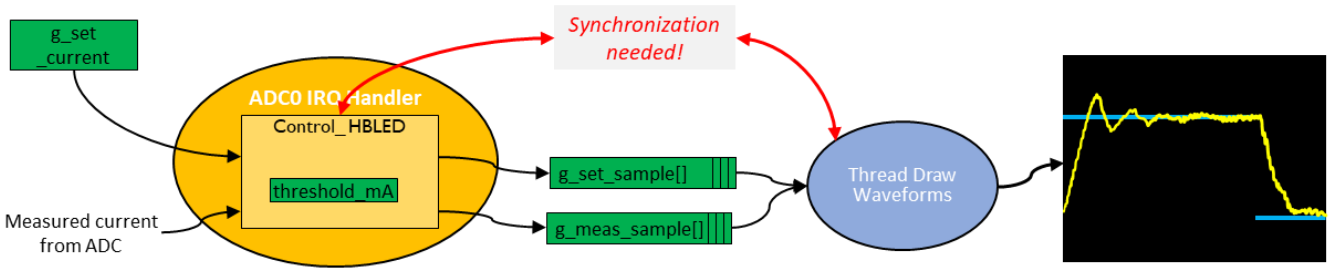
NEXT STEPS POSSIBLE TO IMPROVE RESPONSIVENESS

The base code is very conservative when using the LCD mutex, acquiring it as early as possible and releasing it as late as possible. This raises the blocking time for other threads waiting to use the LCD. The LCD controller will work correctly if each critical section (i.e. rectangle transaction sequence) is performed sequentially (not concurrently). Switching between threads will not corrupt the LCD communication as long as the switch happens when no critical section (i.e. transaction sequence) is active (has started but not finished).

You can improve the responsiveness of the LCD updates by shortening the maximum time that any thread holds the LCD mutex. You could change the project code so that LCD_mutex is acquired before and released after each critical section, rather than for all of the thread's critical sections (as in the base code).

However, it might not be worth doing all of this work. For example, if many critical sections are shorter than the mutex time overhead, the LCD updates will run much more slowly. It would be good to know how long the critical sections really are before adding the OS mutex calls. You would start by finding the critical sections in the source code and instrumenting them with code to control output debug bits (to be able to view them on the logic analyzer). You could then analyze the timing behavior and decide how to proceed. *This analysis and optimization is outside the scope of this assignment.*

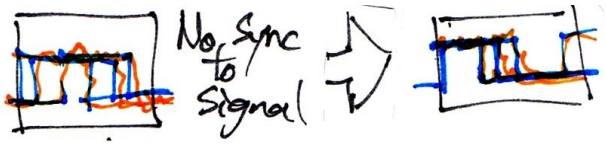
SYNCHRONIZATION FOR SCOPE WAVEFORM DISPLAY



The program displays the current setpoints and measurements over time in an oscilloscope-style plot, allowing evaluation of control system performance. The ADC IRQ handler does two things: it executes the control loop and it also gathers current data to display on the LCD. There are two software processes involved:

- The ADC0 IRQ Handler ("ADC ISR" here) calls the function Control_HBLED as a subroutine each time the control loop needs to run. Besides updating the PWM duty cycle, Control_HBLED can also acquire two waveforms by saving latest values of the setpoint current and the measured current to data buffers (g_set_sample and g_meas_sample). It does not synchronize this data acquisition; you will add it.
- Thread_Draw_Waveforms ("TDW" here) plots two waveforms (setpoint current and measured current) on the LCD using the data from the buffers.

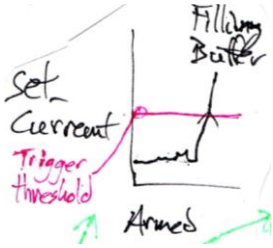
We need synchronization here for two reasons:



- Stable Display: We want the waveforms to be displayed so that the start of the setpoint pulse is always at the left edge of the display. This stabilizes the displayed waveforms and simplifies analysis and is called triggering for oscilloscopes and logic analyzers. It is an example of synchronization.
- Buffer Management: Drawing the waveforms (even one pair of pixels at a time) may be slow enough to get in the way of the urgent control loop processing. In certain cases, the IRQ handler may preempt a thread which is already

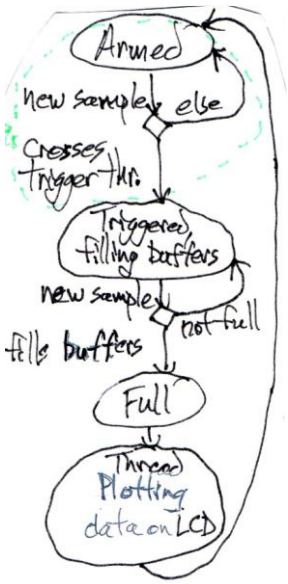
using the LCD. When the handler plots on the LCD it will corrupt the ongoing thread's LCD operation. To prevent this, the data is buffered, allowing the system to defer drawing until later with a lower-priority thread.

This part of the system requires synchronization between the input data, software processes, and data buffers. In this particular application, the triggering and the buffer management are related and can both fit together into a single set of synchronization rules.

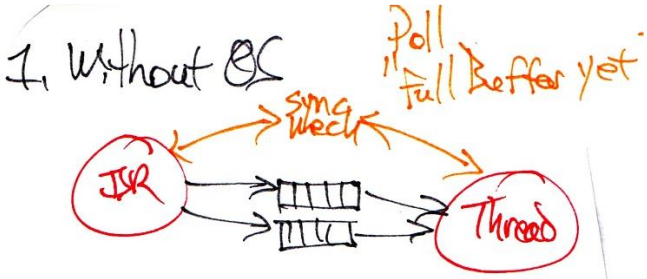


- To provide triggering like an oscilloscope, Control_HBLED must not start filling the buffers until the current setpoint exceeds the trigger threshold (threshold_mA).
- Control_HBLED must stop filling the buffers when they are full.
- TDW must not start plotting the waveforms until the two data buffers are full.
- Control_HBLED must not start filling the buffers if TDW has not finished plotting the waveforms from the data buffers.

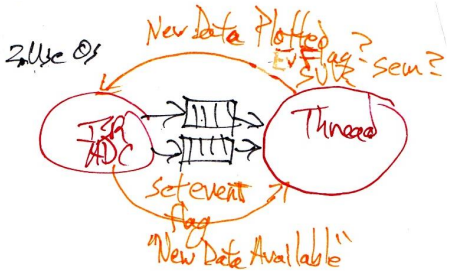
ECE 560 students will implement this behavior in two ways: without and with RTOS mechanisms. ECE 460 students will choose one of these ways (or both for extra credit).



- The first approach does not use the RTOS for these synchronization activities (but the rest of the system does use the RTOS). We can use a state machine to describe the desired behavior. Design code using this state machine (or your improved version) where the thread and ISR explictly examine the state machine's state variable to decide what to do and whether to change the state. Your code must clearly show each state's code (e.g. with variable names and comments), as shown in the ESF textbook (e.g. Listing 3-14).



- The second approach uses the RTOS. The ISR and thread use the RTOS to signal when the other may proceed. Event flags are a simple solution, though other OS mechanisms may be feasible as well. Note that your code won't necessarily be structured as a state machine, as some calls to the OS by the thread will provide the blocking needed.



Your code must select the mechanism using C pre-processor's conditional compilation:

- In config.h, add a directive to define a symbol SCOPE_SYNC_WITH_RTOS to specify which approach to enable. Define it to 1 to use scope synchronization with RTOS mechanisms, or 0 to use a state machine approach (without RTOS mechanisms).
- In your application code (e.g. in threads.c, control.c, UI.c), select the code by testing that symbol:

```
#if SCOPE_SYNC_WITH_RTOS
    // sync using RTOS
#else
    // sync without using RTOS
#endif
```

4. In your report, explain briefly your approach(es) and reasons for your choice(s).

IMPROVING ROBUSTNESS

You need to modify the program to manage the faults injected by Thread_Fault_Injector. Before protecting the program against the fault, you need to understand the fault and the impact on the system. You'll use the scope/logic analyzer and debugger to see what happens.

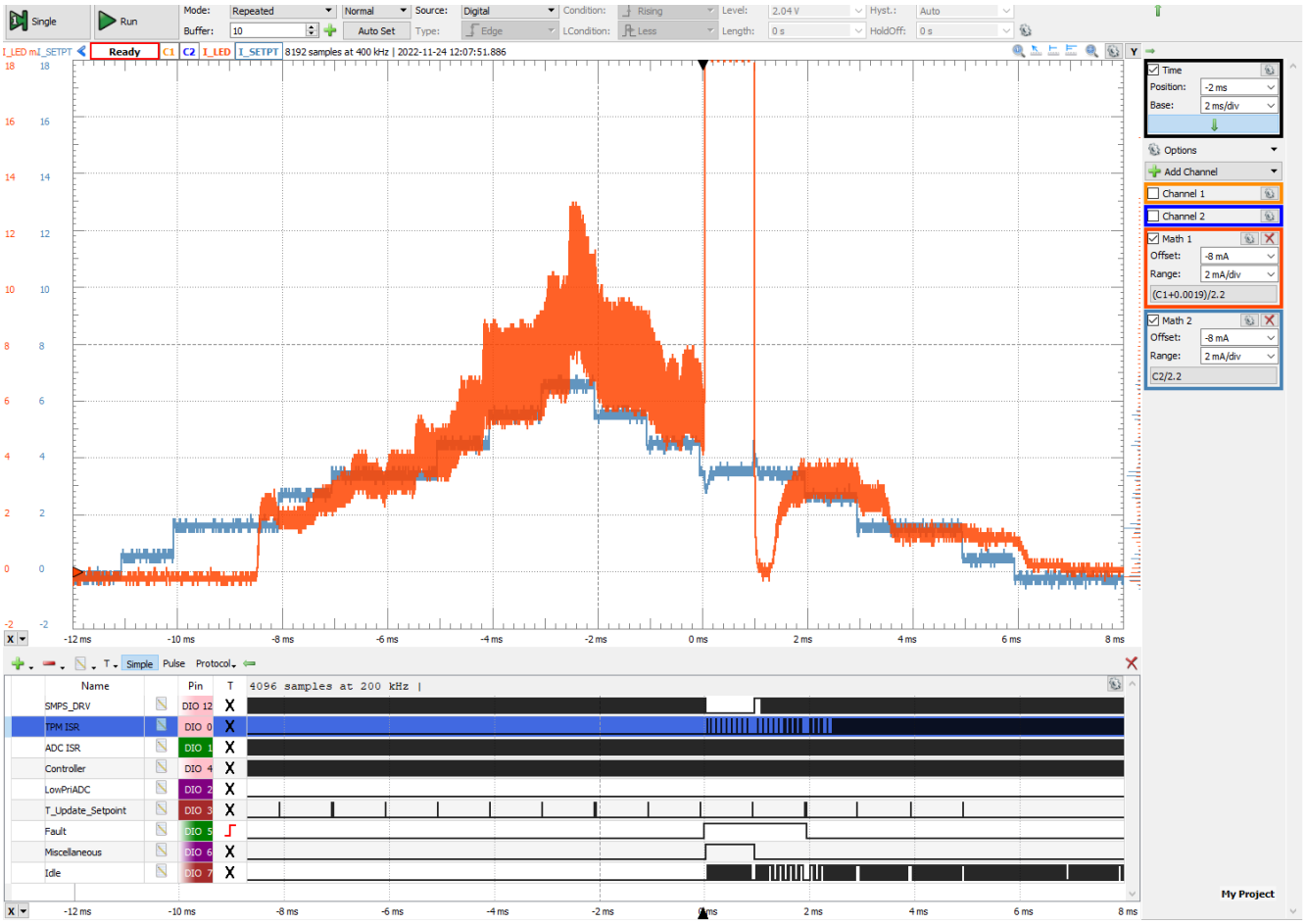
The following table describes the faults and suggests solutions. You must address the faults marked **Required** based on your class. You may address additional faults for extra credit.

Category	Fault Name	Description	Suggested Solutions	Required for ECE 460	Required for ECE 560
Shared Data	TR_Setpoint_High	Overwrites g_set_current_mA, causing short burst of excess current	Data validation		
	TR_Flash_Period	Overwrites g_flash_period, causing fast flashing	Data validation		
	TR_PID_FX_Gains	Overwrites PID FX compensator gain(s), causing poor controller performance	Data validation	Yes	Yes
Language Run-Time Support	TR_Stack_Overflow	Fills and overflows thread's stack space	Goes to HardFault Handler. Use WDT, or have HFH trigger reset.		
Interrupts	TR_Disable_ADC_IRQ	Disables ADC IRQ	Scrub, WDT server		
	TR_Disable_All_IRQs	Sets PRIMASK bit to disables all configurable-priority interrupts	WDT server		Yes
Peripherals and Clocks	TR_Disable_PeriphClocks	Disable clock signals for peripherals.	Scrub		
	TR_Change_MCU_Clock	Changes MCU's clock frequency	Scrub, or timing test		
	TR_Slow_TPM	Changes TPM overflow frequency, reducing switching and control loop frequencies	Check & scrub peripheral settings. Measure timing periodically.		
RTOS IPC	TR_LCD_mutex_Hold	Acquires but doesn't release LCD_mutex	Add time-out to application thread? Other solutions possible too: delete and re-create mutex?		
	TR_LCD_mutex_Delete	Deletes LCD_mutex	Re-create mutex?		
	TR_Fill_Queue	Fill ADC request queue (used to read touchscreen)	Add time-out to application thread? Other solutions possible		
RTOS Scheduling	TR_High_Priority_Thread	Fault injector thread boosts own priority, runs infinite loop.	WDT server		
	TR_osKernelLock	Lock kernel to current thread, prevent switching	WDT server		

1

EVALUATE FAULT AND IMPACT ON SYSTEM

- Examine the source code of Test_Fault (in fault.c) to see the code which injects the fault. Review relevant documentation to understand its immediate effects.
- Comment out all other faults in the array Fault_Tests (in fault.c). Be sure to keep TR_End un-commented to mark the end of the array.
- Rebuild the code and download it.
- Configure the scope for normal triggering on the rising edge of the Fault debug signal and start the scope running.
- Run the code. If not using the debugger, press the reset switch. If using the debugger, stop, reset and run the code.
- Examine the data captured by the scope.



The diagram above shows how a similar system which was **differently-configured** (different LED current profile) responded to a fault (TR_Setpoint_High) at time = 0 ms which changes the variable g_set_current_mA to 1000 mA. The scope is set for normal triggering on the rising edge of the Fault signal to see what happens when the fault is injected.

The compensator increases the duty cycle to 100% causing the LED current (I_LED, orange trace) to shoot up and go off the screen. However, the system starts to recover at 1 ms when Thread_Update_Setpoint runs again. That thread writes a correct value to g_set_current_mA.

[1] a

Note that the DAC-generated analog debug singal (I_SETPT, blue) does not change. The fault only modified the variable g_set_current_mA but not the DAC data register.

DESIGN AND IMPLEMENT A SOLUTION

Which fault management method to use depends on the type of fault and its impact on the system. We can try to detect the fault and respond to it, or we can redesign the system to mask the fault regardless of whether it happens. Some example methods are presented below. You will use the scope to evaluate how the system responds with your solution. You may need to iterate several times to get the behavior you want.

MODIFICATIONS TO FAULT.C

You will modify fault.c to select the faults you wish to test by changing the definition of the array Fault_Tests. You may temporarily disable tests to simplify development. Do not make other changes to fault.c or fault.h without first confirming with the instructor. Be sure that your submitted code's definition of the array Fault_Tests only has the faults which you manage (and TR_End to indicate the end of the test list). The other faults must be commented out.

SUMMARY OF APPROACHES

One type of solution **detects faults** and then **responds** by running fault handling code.

- First, detect the fault. There are various approaches possible:
  - Use system support:
    - MCU Interrupt System: use the Hard Fault exception handler. Please refer to Practical Homework 2 to refresh yourself on causes of hard faults. There is also a Low Voltage exception handler.
    - RTOS: use OS Error Callback function (osRtxErrorNotify), which runs if the OS detects stack overflows, illegal OS service calls, queue overflows and other faults.
  - Software testing of data values: variables, parameters, message contents:
    - Do range or sanity checks on data. Is the data out of range? Did the data change too much since the last reading? Does the data match an internal model?
    - Use a hash or checksum on the data.
    - Use two copies of the data and compare them to detect the fault.
  - Analyze timing behavior
    - Use watchdog timer to detect lack of progress by program.
    - Use timeouts on blocking RTOS operations.
    - Use counters to make sure threads are running often enough
  - Other
    - If any peripherals are configured improperly, reconfigure them.
- Second, handle the detected fault.
  - Correct the data (or at least reduce amount of error)
    - Clip data to the closest legal value in range.
    - Use a back-up copy of the data.
    - Use previous data value (hopefully valid).
    - Use estimate (model) of correct data value.
  - Delete and recreate RTOS component.
  - Reset entire system. Done automatically by watchdog timer.

Another type of solution is to modify the system design so faults are **automatically masked and corrected**. This will always incur additional processing, regardless of whether the fault occurred.

- Redesign to mask the fault without detection.
    - For a critical data variable, use three copies of a variable and do majority voting to select the best value.
    - Scrub the system: Periodically re-initialize key peripherals and data.

REFERENCES FOR APPROACHES

For more information, please refer to:

- Example Fault Test in ESA Project Report Template.
- The Dependable System lecture notes and the associated videos,
- **Embedded Systems Fundamentals**, Chapter 7 on Watchdog Timers (p. 185, pp. 190-194).
- Watchdog timer example code for **ESF** Chapter 7 on Github at https://github.com/alexander-g-dean/ESF/tree/master/NXP/Code/Chapter_7/WDT%20Demo
- NASA Software Fault Tolerance Tutorial section 4.1.2. on Error Detection.

SUBMISSION INFORMATION

GENERAL

1. You may work individually or with one partner on this project.
2. Your code may be examined for possible plagiarism by using MOSS (http://theory.stanford.edu/~aiken/moss/).
3. There are various opportunities for extra credit described in this project specification.

GRADING

Report quality, code functionality, and code quality will be used to determine your grade. The instructors may deduct points for remarkably bad coding practices (for example functions longer than a page, magic numbers, non-descriptive names). Various coding recommendations are described online at http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html. If you have a question about what is acceptable, please post it on the class discussion forum so others can learn as well.

DELIVERABLES

Submit the following items online:

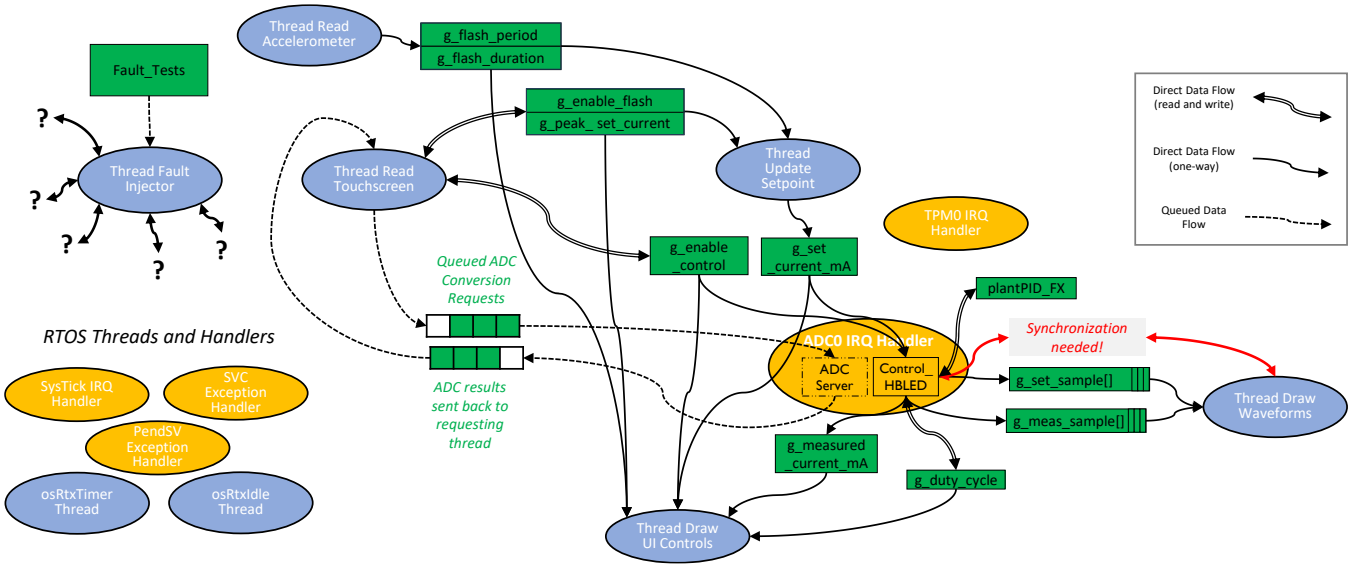
- Project report (PDF) including all answers, based on the ESA Project Report Template.docx.
- Spreadsheet with answers to specified questions for faster grading, based on this spreadsheet: https://docs.google.com/spreadsheets/d/1tmoeZfrCedaUlmg_DirbqBHKZXydbEcZyum8kqedGtU/edit?usp=sharing. The spreadsheet includes instructions.
- Archive (zip file) of your project directory and subdirectories (except Listings and Objects subdirectories).

REPORT

Submit a PDF report based on the provided template (ESA Project Report Template.docx).

All screenshots must be legible and support the explanatory text. For example, the image below does not support the assertion that Control_HBLED runs at 96 kHz. Since the logic analyzer data is gathered with a 40 kHz sampling rate, the Control_HBLED signal is undersampled and only provides limited frequency information.

DETAILS OF STARTER CODE



FAULT INJECTION

A thread (Thread_Fault_Injector) will inject faults into the program periodically. Every two seconds (FAULT_PERIOD), the thread will do the following:

- Read the type of the current fault test from the array Fault_Tests
- Light the RGB LED red
- Print a message on the LCD (yellow on red) indicating the test number
- Call Test_Fault, which will
  - Set the fault debug output (see debug.h for DBG_FAULT_POS) to 1 for scope triggering
  - Inject the fault
  - Block briefly to make the fault debug output signal longer and more visible
  - Clear the fault debug output to 0
- Increment the fault test number

- Turn off the RGB LED

When the last test has been completed, the thread will light the RGB LED green as an indication.

GRAPHICAL CURRENT DISPLAY

Besides performing closed loop control, the function Control_HBLED also gathers current data to be plotted later on the LCD. Each time the function runs it updates two simple buffers (data starts at element 0) with the latest setpoint current $I_{LED\ setpoint}$ ((g_set_sample[]) and the latest measured current I_{LED} (g_meas_sample[]).

Thread_Draw_Waveforms calls UI_Draw_Waveforms, which calls UI_Draw_Scope to plot the data from the buffers. The plot shows SAM_BUF_SIZE (e.g. 960) samples of data and and is not synchronized to start of the flash.

SHARING THE ANALOG TO DIGITAL CONVERTER

The ADC needs to be used by both the HBLED controller and the touchscreen controller thread. The system uses a software-implemented ADC server which prioritizes conversions for the HBLED controller over other conversion requests. There are two types of ADC conversion.

- Conversions for the buck converter are time-critical, so they are triggered with software in the TPM0 ISR.
- Conversions for the queued requests are triggered by software in the ADC ISR.

A **static** state variable identifies the type of conversion. The thread and ADC handler in the ISR communicate as shown in Figure 4.

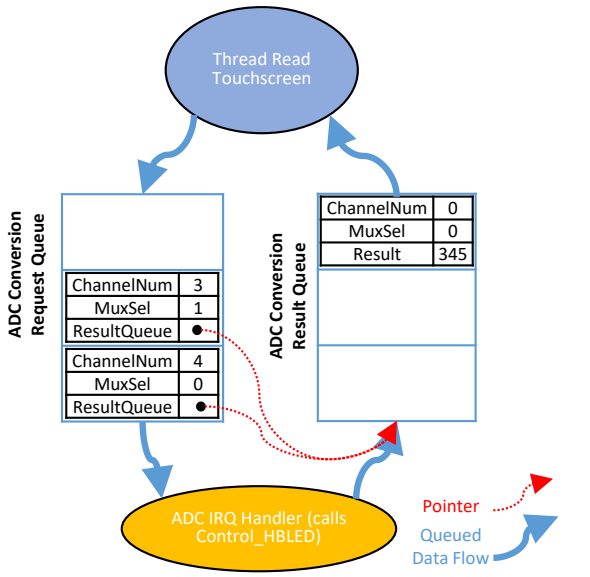


Figure 4. Diagram of queues, data structures and interconnection. The queue sizes shown are examples.

ADC CONVERSION REQUESTS

A client thread (e.g. Thread_Read_TS) sends a request message to the ADC ISR through a queue (the request queue). Each message uses a data structure that holds the following information:

- ChannelNum: Number of channel to convert, used for field ADCH in ADC register SC1n.
- MuxSel: Multiplexer setting for ChannelNum, used for field MUXSEL in ADC register CFG2.
- ResultQueue: Pointer to queue to hold conversion result. This field allows the system to be extended so that other threads can request A/D conversions and get their results correctly.

ADC CONVERSION RESULTS

The ADC ISR sends result messages back to the client thread through another queue (the result queue). Each message uses a data structure that holds the following information:

- Result: Value of ADC conversion result
- ChannelNum: Number of channel which was converted

ADC ISR STRUCTURE

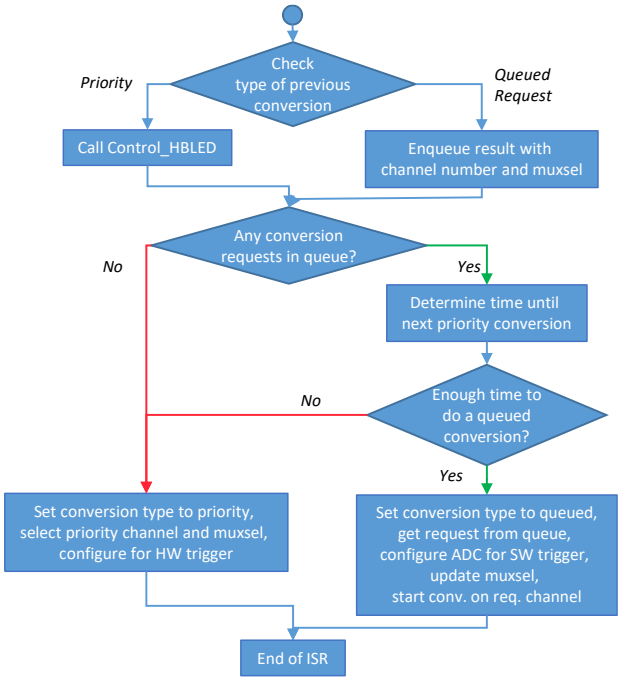


Figure 5. Control flow of ADC IRQ Handler

The control flow for the ADC IRQ handler is shown in Figure 5.