PES Project 3 – Total 75 Points (with 10 points possible Extra Credit)

This third PES project will extend your use of the KL25Z with the MCUXpresso IDE to memory and pointer operations as well as let you try some of the UML diagramming that we discussed in class. As if creating a test for memory on a new board, you'll create a command processor to run tests against memory by parsing a script, and you'll create the first phase of a logger utility to trace program execution. Extra credit options are available for developing a set of unit tests via µcUnit.

Part 1 (65 points) – Create a memory test utility to run against allocated heap memory on the KL25Z

There are several key elements to creation and demonstration of this utility, including: a command processor and script parser, a suite of memory tests and support functions, a random pattern generator, a logger, an LED control, a main program to run an example test script, and (optionally) µcUnit test cases.

Command Processor and Script Parser

- Create a framework for storing and executing command functions (similar to the one described in the Hardware Lecture slides 33-38)
- Your command processor will contain command elements defined by a typedef similar to Hardware slide 34, which will include the command name, a string to print when executed, and a function pointer to the associated memory test function
    - You **MUST** demonstrate function pointer use in handling commands
- Your processor will take in a set of string commands as a test script shown below (from a constant string array in your main code) specifying each memory test to be run in the script
- By parsing each line in the script, such as **A,32** or **P,32,143** you will identify a command to run, and then execute that command with any provided arguments
- You should have an #IFDEF option for your script parser to either run the entire script (printing a string after each command to the UART) or to wait for a key press before each command is executed

Required Test Script – note, comments below are not required to be included in the script or parsed, only the actual commands

A,32	**//Allocate a 32 byte region in the heap**
M	**//Verify that memory is allocated (True)**
P,32,143	**//Write a pattern using the seed 143 to the first 32 bytes of the allocated memory**
D,32	**//Display the first 32 bytes of allocated memory**
D,34	**//Display the first 34 bytes of allocated memory (Out of range error)**
C,32,143	**//Compare the first 32 bytes of allocated memory to the pattern with seed 143 (True)**
W,12,2,0xFFEE	**//Write the hex value 0xFFEE to 2 bytes of allocated memory starting at byte 12**
D,32	**//Display the first 32 bytes of allocated memory**
C,32,143	**//Compare the first 32 bytes of allocated memory to the pattern with seed 143 (False)**
P,16,127	**//Write a pattern using the seed 127 to the first 16 bytes of the allocated memory**
D,16	**//Display the first 16 bytes of allocated memory**
C,16,127	**//Compare the first 16 bytes of allocated memory to the pattern with seed 127 (True)**
I,35,2	**//Invert all bits in 2 bytes of allocated memory starting at byte 35 (Out of range error)**
I,9,2	**//Invert all bits in 2 bytes of allocated memory starting at byte 9**

D,16	**//Display the first 16 bytes of allocated memory**
C,16,127	**//Compare the first 16 bytes of allocated memory to the pattern with seed 127 (False)**
I,9,2	**//Invert all bits in 2 bytes of allocated memory starting at byte 9**
C,16,127	**//Compare the first 16 bytes of allocated memory to the pattern with seed 127 (True)**
D,16	**//Display the first 16 bytes of allocated memory**
F	**//Free the allocated memory in the heap**
M	**//Verify that memory is allocated (False)**

Memory Test Functions

- You will create a set of eight utilities for memory testing. These test functions will be used against a dynamically allocated memory region using the heap storage on your KL25Z
- You must determine appropriate function signatures and include files for dealing with heap allocated memory pointers and offsets
- Functions should do bounds checking and warn the user if they attempt to read or write memory outside the allocated range (or if no memory block has been allocated). Functions will return an error code appropriate to the function to indicate issues have occurred (or not). Use an enum to specify the error codes:

```
typedef enum mem_status
        {
                SUCCESS = 0,    // no error
                FAILED          // failure case
        } mem_status;
```

  You may extend the error codes definitions from this basic enum as you wish.
- Each test should be in its own .c file, and included in the main program for use
- Memory test functions will include (mapping to script commands are shown in [brackets]):
  - *allocate_bytes* – [**A,n**] – Allocate a block of n bytes of memory. A maximum memory block size should be checked, with an error thrown for an invalid allocation request (execution can continue after the error) – the routine should return a pointer to the allocated memory. You may use malloc on the KL25Z to simplify this element of the project.
  - *free_allocated* – [**F**] – Free an allocated block of memory (using free()). If free_memory is called with no memory having been allocated, a warning message should be issued, but execution can continue.
  - *verify_memory* – [**M**] – Verify that an allocated block of memory is available for testing with a Boolean return
  - *display_memory* – [**D,n**] – Display the contents of a memory region by returning the contents of memory at the location – arguments to the function include a memory address and a number of bytes to display from the beginning of the memory region. The display should show 4 bytes of data at a time in the form: address of first byte in heap, hex value of 4 bytes, binary values of 4 bytes:
    0x200010F0 0xF0F0E0E0 0b11110000 0b11110000 0b11100000 0b11100000
  - *write_memory* – [**W,m,n,v**] – Specifies an address (m bytes offset into allocated region), a length in bytes (n), and a value (n bytes) to write. The memory at that location is modified accordingly.

- *invert_block* – [**I,m,n**] – Specifies an address (m bytes offset into allocated region) and a number of bytes (n). All bits in this region should be inverted using XOR.
- *write_pattern* – [**P,n,s] –** Specifies an address, n number of bytes, and a seed value s. The seed value and the number of bytes will be provided to a pattern generator utility function, which will return a byte array, where each byte has a random value based on the seed. The bytes returned will then be written into memory starting at the specified address. (Note: the data type for your seed value is based on your pattern generation approach.)
- *compare_pattern* – [**C,n,s**] - Specifies an address, n number of bytes, and a seed value. Using the seed value and the pattern generator utility, generate a byte array with random values based on the seed. Check whether the newly generated pattern in the byte array returned matches the existing byte pattern in memory at the specified address. The function should return a Boolean result for the test.

Pattern Generator

- For the pattern creation, create a standalone .c file with the pattern generation function.
  - *gen_pattern* – Your pattern generation function will accept a number of bytes and a seed value and return a byte array. (Note: the data type for your seed value may vary based on the algorithm you use.) You may pass in a pre-allocated byte array for the routine to use for storing the generated pattern.
  - You may not use any library functions (e.g. random()) to generate this pattern. You may use pseudo-random number generation code you find elsewhere but you must provide attribution and the code should be commented to explain how it works.
  - The random number generating function you create must be able to use a seed value (you may decide on the data type) to create a pseudo-random generation of 8-bit unsigned integers, each of which will be stored in a byte array to be returned. Any time the same seed value is provided to this function, the exact same pseudo-random values should be generated for the byte array. You'll want to test to ensure that this is the case.

Logger

- You will create a simple logger (logger.c/logger.h) for displaying debug messages or data generated by the memory tests. (Initially this logger will not use timestamps and will be run in the main program thread, in later projects we will extend and enhance this design.) You should provide the option of enabling or disabling the logger via #IFDEF. If disabled, messages sent to the logger by other routines are ignored. If enabled, the logger will send incoming messages to your program's output along with any normal displays. You may design appropriate signatures for these functions.
- When running on the KL25Z, output should be directed to the UART and displayed in a terminal window. The logger should initialize to a disabled state.
- Logging functions include:
  - Log_enable – begin printing log messages when called
  - Log_disable – ignore any log messages until re-enabled
  - Log_status – returns a flag to indicate whether the logger is enabled or disabled
  - Log_data – display in hexadecimal an address and contents of a memory location, arguments are a pointer to a sequence of bytes and a specified length
  - Log_string – display a string

- Log_integer – display an integer
- The previous three commands should include a new line after each display

LED Control

- You will create an LED control function (led_control.c/led_control.h) that will be able to turn the KL25Z LED either red, green, blue, or off. This function should only be used when running your memory tests on the KL25Z (with logger enabled or disabled). At initialization, turn the LED off. When the memory test starts, the LED should be set to blue. For each command, if an error occurs during the test or a Boolean returns false, the LED should be set to red. If the command runs as expected or a Boolean is true, the LED should be set to green. The LED changes should be clear when the test script is run in a stepped fashion between commands. You may add a delay after LED activation for the non-stepped script run if you want to see the LED change (optional).

Development Environment/Limits

For this project, you will develop using the MCUXpresso IDE environment as shared in class demonstration. See the SAs for any assistance you need in your development environment.

Your code should follow the ESE C Style Guide as closely as possible.

When compiling use -Wall and -Werror compiler flags at a minimum. Your code should have no compilation errors or warnings.

Note that this project is intended to be a bare metal implementation, meaning, you will not use advanced SDK or RTOS routines. You may use include files generated from MCUXpresso tools such as pin_mux.h. You may also use “board.h” and “MKL25Z4.h”. You may not use the “fsl_” include file that provide higher-level SDK-based functions for LED or Slider control. You may use standard C libraries.

Part 2 (10 points) – Create a UML Sequence Diagram for the above test cycle

Create a UML Sequence diagram that shows each of the activities in the test cycle above. List the main modules in your program across the top of the diagram and show message flow between them for one complete test cycle.

If you don’t have a tool for creating UML diagrams, I suggest Draw.IO or Lucidchart, but you can also do the diagram by hand (as long as they are readable for grading). In any case, the diagram must be thorough, complete, and readable, and submitted in your GitHub repo as a PDF.

Part 3 – Extra Credit (10 points) – Create a Unit Test set (unitTest.c/unitTest.h) for the application.

Using µcUnit as presented in class, create a minimum of ten example unit tests for some of the memory test functions. These tests should confirm some normal and abnormal responses for some subset of the memory test functions. Execute the unit tests in a test script. The tests should be run on your development environment, and output from the tests should be captured in a unitTestResults.out file (can be created from cut and paste or output piping, does not require file output).

Project Submission

The project is due on **Tuesday 3/3** prior to class and will be submitted on Canvas. The project will also be demonstrated interactively with the class staff. Appointment slots will be posted for demonstrations. The Canvas submission will consist of two parts:

Part 1 is a single GitHub repo URL which will be accessed by graders to review code and documentation. This will consist of any C code or include files, captured output files, and a README Markdown document that includes:

- A title (PES Project 3 Readme)
- Names of your team
- Description: description of the repo contents
- Observations: A description of any issues or difficulties you encountered on the project and how they were addressed, or any assumptions you made when met with incomplete specifications
- Installation/execution notes: for others who may use the code – this should include compilation instructions for the SAs to more easily grade
- The repo should also contain the PDF with the UML Sequence diagram from above
- Please include a Git tag on your final submission to allow the SAs to be clear about what was submitted for grading

Part 2 will be a PDF containing all C code and README documentation – the PDF is used specifically for plagiarism checks: your code should be your team's alone, developed by your team. You should provide a URL for any significant code taken from a third party source, and you should not take code artifacts from other student teams. However, you may consult with other teams, the SAs, and the instructor in reviewing concepts and crafting solutions to problems (and may wish to credit them in documentation).

Grading

Points will be awarded as follows:

- 30 for the correctness of demonstrated code (execution of both cross-compiled code versions) **– code will be demonstrated with class staff after the due date**
- 25 for the construction of the code (including following style guide elements, required elements, and the quality of solution)
- 10 points for the README
- 10 points for the UML diagram
- And there is up to 10 points of extra credit in this submission for unit testing

Assignments will be accepted late for one week. There is no late penalty within 4 hours of the due date/time. In the next 24 hours, the penalty for a late submission is 5%. After that the late penalty increases to 15% of the grade. After the one week point, assignments will not be accepted. The team may submit the assignment using a late pass from each of the team members. Only one late pass can be submitted per project, and it will extend the due date/time 24 hours.