```
#include
<stdlib.h>
             #include <stdio.h>
             #include "board.h"
             #include "peripherals.h"
             #include "pin_mux.h"                  //CALL THE HEADER FILES AND SETS OF
             LIBRARY FUNCTIONS
             #include "clock_config.h"
             #include "MKL25Z4.h"
             #include "fsl_debug_console.h"
             #include "logger.h"
             #include "memory_test.h"
             #include "led.h"


             //PATERN GENERATION DEFINES FOR MERRAIN TWISTER RANDOM
             //PATTERN GNERATION FUNCTION
             #define UPPER_MASK           0x8000
             #define LOWER_MASK           0x7fff
             #define TEMPERING_MASK_B     0x9d2c
             #define TEMPERING_MASK_C     0xefc60000
             #define STATE_VECTOR_LENGTH 62
             #define STATE_VECTOR_M      39 /* changes to STATE_VECTOR_LENGTH also require
             changes to this */

             typedef struct tagMTRand                            //defining a structure
             data type of MTRand
             {
                 unsigned int mt[STATE_VECTOR_LENGTH];         //DEFINING THE ARRAY TO
             STORE THE PATTERN
                 int index;                                  //using the index to specify
             the array elements
                 //the value points to
             } Random;

             unsigned int length=4;                     // DEFINING THE SIZE OF THE BLOCK
             TO BE DYNAMICALLY
             //ALLOCATED

             extern unsigned int pat[100];      //INITIALIZING AN ARRAY PAT TO STORE THE
             RANDOM PATTERNS
             //GENERATED BY RANDOM PATTERN GENERATOR
             Random seedRand(unsigned int seed);
```

```
unsigned int genRandLong(Random* rand);  //PROTOTYPES OF ALL FUNCTIONS WHICH
ARE
//INVOLVED IN GENERATING THE RANDOM PATTERN
unsigned int genRand(Random* rand);
void m_seedRand(Random* rand, unsigned int seed);
void gen_pattern(size_t length,unsigned int seed);


int main(void)
{
    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitBootPeripherals();
    /* Init FSL debug console. */
    BOARD_InitDebugConsole();
    //printf("hello");
    PRINTF("Hello World\n\r");

    LED_RED_INIT(1);
    LED_GREEN_INIT(1);                      //INITIALIZING THE LED INIT PINS
    LED_BLUE_INIT(1);






    //MEMORY TEST SUITE



    uint32_t *test_space = allocate_words(length);  //ALLOCATING THE ADDRESS
OF THE BLOCK_POINTER
    //TO TEST SPACE

    //EXECUTING THE MEMORY TESTS WHICH HAS
    //FOLLOWING FUNCTIONS
    write_pattern( test_space,  length,  10); //CALLING WRITE PATTERN WHICH
WRITES PATTERN TO
    //MEMORY LOCATION

    display_memory(test_space,length);   //CALLING THE MEMORY CONTENTS
    PRINTF("\n\r");
    verify_pattern( test_space, length, 10);  //VERIFY PATTERN TO CHECK IF THE
MEMORY CONTENTS
    //MATCH THE CONTENTS OF THE ARRAY WHICH CONTAINS
    // THE RANDOMLY GENERATED PATTERN
    PRINTF("\n\r");



    write_memory(test_space,2,0xFFEE);             //WRITE TO MEMORY VALUE
0XFFEE AT OFFSET 2
    PRINTF("\n\r");
    display_memory(test_space,length);  //DISPLAY THE MEMORY CONTENTS OF THE
BLOCK_POINTER
    PRINTF("\n\r");
    verify_pattern( test_space, length, 10); //VERIFY PATTERN TO CHECK IF THE
MEMORY CONTENTS
    //MATCH THE CONTENTS OF THE ARRAY WHICH CONTAINS

    PRINTF("\n\r");
    PRINTF("\n\r");
    write_pattern( test_space,  length,  10);  //CALLING WRITE PATTERN WHICH
WRITES PATTERN TO
    //MEMORY LOCATION
    PRINTF("\n\r");
    display_memory(test_space,length);     //DISPLAY THE MEMORY CONTENTS OF
THE BLOCK_POINTER
    PRINTF("\n\r");
    verify_pattern( test_space, length, 10); //VERIFY PATTERN TO CHECK IF THE
MEMORY CONTENTS
    //MATCH THE CONTENTS OF THE ARRAY WHICH CONTAINS
    PRINTF("\n\r");

    PRINTF("\n\r");                  //INVERT THE BUFFER_POINTER AT THE
LOCATION WITH 1 OFFSET
    invert(test_space,1);
    PRINTF("\n\r");
    display_memory(test_space,length);  //DISPLAY THE MEMORY CONTENTS OF THE
BLOCK_POINTER
    PRINTF("\n\r");
    verify_pattern(test_space, length, 10); //VERIFY PATTERN TO CHECK IF THE
MEMORY CONTENTS
    //MATCH THE CONTENTS OF THE ARRAY WHICH CONTAINS
    PRINTF("\n\r");

    PRINTF("\n\r");
    invert(test_space,1);         //INVERT THE BUFFER_POINTER AT THE LOCATION
WITH 1 OFFSET
    PRINTF("\n\r");
    display_memory(test_space,length); //DISPLAY THE MEMORY CONTENTS OF THE
BLOCK_POINTER
    PRINTF("\n\r");
    verify_pattern( test_space, length, 10); //VERIFY PATTERN TO CHECK IF THE
MEMORY CONTENTS
    //MATCH THE CONTENTS OF THE ARRAY WHICH CONTAINS

    PRINTF("\n\r");

    PRINTF("\n\r");
    get_address(test_space,3);      //GET ADDRESS OF A PARTICULAR OFFSET FROM
LOCATION
    PRINTF("\n\r");
    free_words(test_space,length); //FREE BLOCK_POINTER
    LED_BLUE_OFF();
    LED_GREEN_ON();               //TURN THE LED GREEN ON
    LED_RED_OFF();

    PRINTF("\n\r");
    printf("\n\r");


    return 0 ;
}






void gen_pattern(size_t length,unsigned int seed)
{
    unsigned int i;            //GENERATE PATTERN FUNCTION WHICH STORES THE
RANDOM PATTERN
    //IN THE PARTICULAR ARRAY

    Random r = seedRand(seed);
    for(i=0; i<length; i++)
    {
        pat[i]=genRand(&r);   //STORE THE RANDOM PATTERN IN ARRAY PAT

    }

}


void m_seedRand(Random* rand, unsigned int seed)
{
    rand->mt[0] = seed & 0xff;        //TAKE THE FIRST ELEMENT OF THE ARRAY MT
OF VARIABLE
    //RAND AND DO BITWISE OPERATIONS TO GET A PARTICULAR SET OF VALUE
    for(rand->index=1; rand->index<STATE_VECTOR_LENGTH; rand->index++)
    {
        rand->mt[rand->index] = (6069 * rand->mt[rand->index-1]) & 0xff;
    }
}

/*
 Creates a new random number generator from a given seed.
*/
Random seedRand(unsigned int seed)
{
    //TAKE PARTICULAR VARIABLE OF STRUCTURE NAMED RAND AND
    //DO THE FOLLOWING OPERATIONS
    Random rand;
    m_seedRand(&rand, seed);  //CALL M_SEED TO DO A PARTICULAR SET OF
OPERATIONS FOR PATTERN
    //GENERATION
    return rand;
}


/**
 * Generates a pseudo-randomly generated long.
 */
unsigned int genRandLong(Random* rand)
{

    unsigned int y;
    static unsigned int mag[2] = {0x0, 0x99};      /* mag[x] = x * 0x99 for x
= 0,1 */
    if(rand->index >= STATE_VECTOR_LENGTH || rand->index < 0)
    {
        /* generate STATE_VECTOR_LENGTH words at a time */
        int kk;
        if(rand->index >= STATE_VECTOR_LENGTH+1 || rand->index < 0)
        {
            //CHECK IF RAND VARIABELS INDEX FALLS BETWEEN A
            //PARTICULAR RANGE AND DO PARTICULAR BITWSIE OPERATIONS
            //ON   THE ARRAY MT OF VARIABLE RAND
            m_seedRand(rand, 43);
        }
        for(kk=0; kk<STATE_VECTOR_LENGTH-STATE_VECTOR_M; kk++)
        {
            y = (rand->mt[kk] & UPPER_MASK) | (rand->mt[kk+1] & LOWER_MASK);
            rand->mt[kk] = rand->mt[kk+STATE_VECTOR_M] ^ (y >> 1) ^ mag[y &
0x1];
        }                                         //CHECK FOR THE ARRAY MT IF
IT FALLS BETWEEN A SET OF VALUES
        //IF IT DOES THEN WE DO BITWISE OPERATION AT THOSE VALUES
        for(; kk<STATE_VECTOR_LENGTH-1; kk++)
        {
            //CHECK FOR THE ARRAY MT IF IT FALLS BETWEEN A SET OF VALUES
            //IF IT DOES THEN WE DO BITWISE OPERATION AT THOSE VALUES
            y = (rand->mt[kk] & UPPER_MASK) | (rand->mt[kk+1] & LOWER_MASK);
            rand->mt[kk] = rand->mt[kk+(STATE_VECTOR_M-STATE_VECTOR_LENGTH)] ^
\
                           (y >> 1) ^ mag[y & 0x1];
        }                                             //CHECK FOR THE ARRAY MT
IF IT FALLS BETWEEN A SET OF VALUES
        //IF IT DOES THEN WE DO BITWISE OPERATION AT THOSE VALUES
        y = (rand->mt[STATE_VECTOR_LENGTH-1] & UPPER_MASK) | (rand->mt[0] &
LOWER_MASK);
        rand->mt[STATE_VECTOR_LENGTH-1] = rand->mt[STATE_VECTOR_M-1] ^ (y >>
1) ^ mag[y & 0x1];
        rand->index = 0;
    }
    y = rand->mt[rand->index++];                          //DO THE MERRIAN
TWISTER ALGORITHM ON THE SET OF VALUES OF THE ARRAY
    y ^= (y >> 11);
    y ^= (y << 7) & TEMPERING_MASK_B;
    y ^= (y << 15) & TEMPERING_MASK_C;
```

```
        y ^= (y >> 18);
        return y;
    }

    /**
     * Generates a pseudo-randomly generated double in the range [0..1].
     */
    unsigned int genRand(Random* rand)                       //CONVERT THE
    GENRANDLONG INTO AN UNSIGNED  FORM AND GET THE 8 BIT
    //RANDOM PATTERN
    {
        return(genRandLong(rand) / 0xfffff);
    }
```

```
#include
"led.h"

           uint32_t constant_value=400000;      //TAKE A CONSTANT VALUE OF 400000 AND DELAY
           IT FOR THE
                                                                             //THE
           REQUIRED TIME
           uint32_t j=0;
           void Delay(uint32_t time)
           {                                                          //WASTE THE
           CLOCK CYCLE FOR PARTICULAR VALUE TO GET THE
                                                                             //DELAY
           OF THE LED
           uint32_t clock_ticks=(constant_value* time)/(500);
           for (j=0;j<clock_ticks;j++);
           }
           void led_blue()
           {
```

```
        LED_BLUE_ON();    // TURN THE BLUE LED ON FOR INITIAL PERIOD
        Delay(200);
        LED_BLUE_OFF();  // TURN IT OFF AFTER THE DELAY
        Delay(100);
}
void led_red()
{

        LED_RED_ON();     // TURN THE BLUE LED ON FOR ERRORS DURING MEMORY TESTS
        Delay(200);
        LED_RED_OFF();   // TURN IT OFF AFTER THE DELAY
        Delay(100);
}
void led_green()
{
        LED_GREEN_ON();   // TURN THE GREEN LED ON AFTER SUCCESFULLY COMPLETING
THE OPERTION
}
```

```
#ifndef
_LED_H_
          #define _LED_H_
          #include <stdlib.h>
          #include <stdio.h>
          #include <stdint.h>
          #include "board.h"
          #include "peripherals.h"
          #include "pin_mux.h"
          #include "clock_config.h"
          #include "MKL25Z4.h"
          #include "fsl_debug_console.h"

          void led_red();
          void led_green();
          void led_blue();
          void Delay(uint32_t time);
          #endif
```

```
#include
"logger.h"

             int log_enable=1;
             int log_disable=0;
             int status=0;
             int log_status(int status)
             {
                 if (status==log_enable)
                 {
                                                                       // CHECK FOR THE
             STATUS OF LOGS
                                                                              //IF ENABLED
             RETURN THE STATUS
                     return (status);
                 }
                 else                              //IF DISABLED RETURN THE STATUS
                 {

                     return (status);
                 }
             }


             //log_status(log_enable);
             void log_data(uint32_t * Block_Pointer,uint8_t length)
             {
                 if (log_status(status))
                 {

                     int i;                //PRINT THE DATA AND ADDRESS IN HEXADECIMAL
                     for(i=0; i<length; i++)
                     {
                         PRINTF("  DATA AT %p ADDRESS IS %02x \n\r",(Block_Pointer)
             +i,Block_Pointer[i]);

                     }
                 }

             }
```

```
void log_get_address(uint32_t * loc, int offset)
{
    if (log_status(status))                // PRINT THE ADDRESS AT A
PARTICULAR OFFSET
    {

        PRINTF("  ADDRESS AT OFFSET %d is %p \n\r",offset,loc+offset);
    }
}

void log_invert(uint32_t * loc, int offset, uint32_t inverted)
{
    if (log_status(status))
    {
        //INVERT DATA VALE AND ADDRESS PRINTED

        PRINTF(" INVERTED data at %p address is %08x
\n\r",(loc+offset),inverted);
    }
}

void log_invert_block(uint32_t * loc,uint8_t i,uint32_t inverted)
{
    if (log_status(status))               //INVERT BLOCK VALE AND ADDRESS
PRINTED
    {
        PRINTF(" INVERTED data at %p address is %08x \n\r",(loc+i),inverted);
    }

}



void log_verify_pattern(uint32_t * loc,uint8_t i)
{
    if (log_status(status))              //PRINT THE ADDRESS WHICH HAD ERROR
ON VERIFYING
    {
        PRINTF("\n\rAddress %p contains value %p\n\r",&loc[i],loc[i]);
    }
}
```

```
void log_free_memory()
{
    if (log_status(status))             //PRINT THE STATUS FREE THE MEMORY FOR
UART
    {
        PRINTF("\n\rFREE THE MEMORY\n\r");
    }
}

void log_string(uint8_t string)
{
    if (log_status(status))
    {
        if (string==1)
            PRINTF("\n\r WRITE PATTERN TO MEMORY\n\r");
        else if (string==2)
            PRINTF("\n\r DISPLAY MEMORY PATTERN\n\r");    //DISPLAY THE
PARTICULAR MESSAGE

                      //BASED ON THE
                                                 // FUNCTION WHAT IS CALLED
        else if (string==3)
            PRINTF("\n\rINVERT DATA OF A MEMORY LOCATION\n\r ");
        else if (string==4)
            PRINTF("\n\rWRITE DATA TO MEMORY\n\r");
        else if (string==5)
            PRINTF("\n\rVERIFYING RANDOM PATTERN TO MEMORY\n\r");
        else if (string==6)
            PRINTF("\n\rFREE THE MEMORY\n\r");
    }

}
int logger_error_code(int log_err)
{
    if (log_status(status))
    {
                               //PRINT THE PRINTF FUNCTIONS IF RUNNING CODE ON
UART
        if (log_err == 0)
            return    (PRINTF("\n\r Operation Completed \n\r"));
        else if (log_err  == 1)
            return (PRINTF("\n\r Warning: Buffer has not been allocated yet
\n\r"));
```

```
        else if (log_err  == 2)
            return    (PRINTF("\n\r Memory Out of Bounds attempted - Exited
with error code 2 \n\r"));
        else if (log_err ==3)
            return (PRINTF("\n\rError on verifying \n\r")); //PRINT ALL ERRORS
AS THEY COME
        else if (log_err==4)
            return    (PRINTF("\n\rOperation Successful \n\r"));



    }
}
```

```
#ifndef
_LOGGER_H_
             #define _LOGGER_H_
             #include <stdlib.h>
             #include <stdio.h>
             #include <stdint.h>
             #include "board.h"
             #include "peripherals.h"
             #include "pin_mux.h"
             #include "clock_config.h"
             #include "MKL25Z4.h"
             #include "fsl_debug_console.h"
             void log_data(uint32_t * Block_Pointer,uint8_t length);
             int log_status(int status);
             int logger_error_code(int log_err);
             void log_get_address(uint32_t * loc, int offset);
             void log_invert(uint32_t * loc, int offset, uint32_t inverted);
             void log_invert_block(uint32_t * loc,uint8_t i,uint32_t inverted);
             //void log_write_pattern(uint32_t * loc, size_t length, int8_t seed,uint32_t
             pat);
             void log_verify_pattern(uint32_t * loc,uint8_t i);
             void log_free_memory();
             void log_string(uint8_t string);
             #endif
```

```
#include
"memory_test.h"
                  #include "logger.h"
                  #include"led.h"          //CALL THE HEADER FILES CONTAINING THE FUNCTIONS
                  AND LIBRARY FUNCTIONS

                  static unsigned int err; //DECLARE err AS STATIC SO THAT VALUE IS
                  RETAINED WITH
                                                                     //EVERY FUNCTION CALL
                  unsigned int block2[1000];   //DECLARE THE BLOCK2 SIZE WHICH CONTAINS THE
                  CONTENTS OF MEMORY
                  unsigned int pat[100];
                  int error_code(int err)
                  {


                      if (err == 0)
                      {
                         led_blue();
                         logger_error_code(err);   //IF ERROR CODE IS 0, PRINT OPERATION
                  STARTED AND TURN BLUE LED
                                                                            //RUN ON
                  CONSOLE AND TERMINAL
                          return (printf("\n\r Operation Started \n\r"));
                      }
                      else if (err == 1)
                      {
                         led_red();            //IF ERROR CODE IS 1, PRINT WARNING AND TURN
                  RED LED
                                                          //RUN ON CONSOLE AND TERMINAL
                         logger_error_code(err);

                          return (printf("\n\r Warning: Buffer has not been allocated yet
                  \n\r"));
                      }
                      else if (err == 2)
                      {
```

```
       led_red();          //IF ERROR CODE IS 2, PRINT MEMORY OUT OF
BOUND AND TURN RED LED
                                                          //RUN ON
CONSOLE AND TERMINAL
       logger_error_code(err);

        return (printf("\n\r Memory Out of Bounds attempted - Exited with
error code 2 \n\r"));
    }
    else if (err==3)
    {
       led_red();           //IF ERROR CODE IS 3, PRINT ERROR ON
VERIFYING AND TURN RED LED
                                           //RUN ON CONSOLE AND TERMINAL
       logger_error_code(err);
        return (printf("\n\rERROR ON VERIFYING \n\r"));
    }
    else if (err==4)
    {
       led_green();         //IF ERROR CODE IS 4, PRINT OPERATION
SUCCESSFUL AND TURN GREEN LED
                                          //RUN ON CONSOLE AND TERMINAL
       logger_error_code(err);
        return (printf("\n\rOperation Successful \n\r"));
    }



}
uint32_t * allocate_words(unsigned int length)
{

    uint32_t *Block_Pointer;
    int i;                             //ALLOCATING MEMORY TO BLOCK
POINTER 16 BYTES OF SPACE
    Block_Pointer = (uint32_t *)malloc(length*4);
    if (Block_Pointer!=NULL)
    {                                          //IF BLOCK IS ALLOCATED
PROPERLY, NO ERROR
        err=0;
        error_code(err);
    }
    else
    {
        err=2;                 //IF BUFFER IS OVER LIMIT,ERROR PRINT
MEMORY BOUND EXCEEDED
        error_code(err);
    }


    for(i= 0; i<length; i++)

       /*Initialize all memory blocks to 0*/
    {
        *(Block_Pointer+i)=0;
    }

    return (Block_Pointer);
}


void free_words(uint32_t *Block_Pointer,unsigned int length)
{
    int i;
    log_string(6);                   //PRINT  FREE MEMORY
    printf("\n\rFREE THE MEMORY\n\r");
    for (i=0; i<length; i++)     //CHECK IF MEMORY HAS BEEN SUCCESSFULLY
ALLOCATED
    {
        if (Block_Pointer!=NULL)    //IF YES FREE THE BLOCK AND
INITIALIZE TO NULL
        {   free(Block_Pointer);
            Block_Pointer = NULL;
        }

        else
        {
              err=1;               //IF NO GIVE WARNING FREE THE BLOCK
AND INITIALIZE TO NULL
            free(Block_Pointer);
            Block_Pointer = NULL;
        }
    }


}
```

```
//Read from memory
uint32_t * display_memory(uint32_t *Block_Pointer,uint8_t
Data_words_to_read )
{
    int i;
    printf("\n\r DISPLAY MEMORY PATTERN\n\r");
    log_string(2);                        //DISPLAY THE MEMORY PATTERN
STRING
    for(i=0; i<Data_words_to_read; i++)
    {                                                            // PRINT
THE VALUE AT THOSE ADDRESS

        printf("  Data at %p address is %x \n\r",(Block_Pointer)
+i,Block_Pointer[i]);
    }

    log_data(Block_Pointer, Data_words_to_read); //PRINT IN UART
    return (Block_Pointer);
}

void write_memory(uint32_t * loc,uint8_t offset, uint32_t value)
{
    log_string(4);                       //WRITE MEMORY STRING PRINTED
    printf("\n\rWRITE DATA TO MEMORY\n\r");
    loc[0+offset] = value;          // A PARTICULAR VALUE IS WRITTEN TO
THE LOCATION
}



void invert_block(uint32_t * loc, size_t length)
{
    int i,data,inverted;
    for(i=0; i<length; i++)       //INVERT THE ENTIRE BLOCK BY USING XOR
OPERATION AND RUNNING
                                                          //FOR LOOP FROM
START TO END
    {
        data=*(loc+i);
        inverted = 0xFFFFFFFF^data;
```

```
       // Exor with 1 togges the bit
        *(loc+i) = inverted;
        printf(" inverted data at %p address is %08x
\n\r",(loc+i),inverted);

        log_invert_block(loc,i,inverted); //PRINT THE INVERTED VALUE IN
UART
    }

}
void invert(uint32_t * loc, size_t offset)
{
    int data,inverted;
    log_string(3);               //PRINTING STRING INVERT DATA
    printf("\n\rINVERT DATA OF A MEMORY LOCATION\n\r ");

    data=loc[offset];
    inverted = 0xFFFFFFFF^data;  //INVERTING THE PARTICULAR VALUE USING
XOR

    loc[offset] = inverted;

    printf(" \n\rInverted data at %p address is %08x
\n\r",(loc+offset),inverted);
    printf("\n\r");
    log_invert(loc,offset,inverted);   //PRINT THE INVERTED VALUE IN UART
}

uint32_t *  get_address(uint32_t * loc, int offset)
{
    log_get_address(loc,offset);           //GET ADDRESS AT OFFSET FROM
LOCATION
    printf("  Address at offset %d is %p \n\r",offset,loc+offset);
    return (loc+offset);

}

void write_pattern(uint32_t * loc, size_t length, int8_t seed)
{
    int i;
    gen_pattern(length, seed);   // CALL THE RANDOM PATTERN GENERATION
FUNCTION
```

```
    printf("\n\r WRITE PATTERN TO MEMORY\n\r");
    log_string(1);              // PRINT THE STRING WRITRE PATTERN
    for (i=0; i<length; i++)
    {
        loc[i]=pat[i];        //WRITE RANDOM PATTERN VALUE TO THE
BLOCK_POINTER
        printf("%p\n\r",loc[i]);
    }

}


uint32_t * verify_pattern(uint32_t * loc, size_t length, int8_t seed)
{
    int i;

    gen_pattern(length, seed);         // CALL THE RANDOM PATTERN
GENERATION FUNCTION TO VERIFY

    log_string(5);               // PRINT VERIFY PATTERN STRING
    printf("\n\rVERIFYING RANDOM PATTERN TO MEMORY\n\r");
    for (i=0; i<length; i++)
    {
        block2[i]=pat[i];        // COPY THE RANDOM PATTERN INTO ARRAY
BLOCK2

    }
    for (i=0; i<length; i++)
    {
        if(block2[i]==loc[i])     //VERIFY IF MEMORY PATTERN MATCHES
PATTER AT THE LOCATION
        {
            LED_BLUE_OFF();
            LED_RED_OFF();
        }
        else
        {
            err=3;               //IF PATTERN DOES NOT MATCHG AT A
LOCATION PRINT THE ADDRESS
                                            //AND PARTICULAR VALUE AT
THAT ADDRESS
            error_code(err);
            printf("\n\rAddress %p contains value
%p\n\r",&loc[i],loc[i]);
            log_verify_pattern(loc,i);
            LED_BLUE_OFF();
            LED_RED_ON();
        }
    }

    return (loc);

}
```

```
#ifndef
_MEMORY_TEST_H_
                  #define _MEMORY_TEST_H_
                  #include <stdlib.h>
                  #include <stdio.h>
                  #include <stdint.h>
                  #include "board.h"
                  #include "peripherals.h"
                  #include "pin_mux.h"
                  #include "clock_config.h"
                  #include "MKL25Z4.h"
                  #include "fsl_debug_console.h"
                  void write_pattern(uint32_t * loc, size_t length, int8_t seed);
                  uint32_t *verify_pattern(uint32_t * loc, size_t length, int8_t seed);
                  uint32_t *  get_address(uint32_t * loc, int offset);
                  void gen_pattern(unsigned int length,unsigned int seed);
                  void invert(uint32_t * loc, size_t offset);
                  void invert_block(uint32_t * loc, size_t length);
                  void write_memory(uint32_t * loc,uint8_t offset, uint32_t value);
                  uint32_t * display_memory(uint32_t *Block_Pointer,uint8_t
                  Data_words_to_read );
                  void free_words(uint32_t *Block_Pointer,unsigned int length);
                  uint32_t * allocate_words(unsigned int length);
                  int error_code(int err);
                  //void log_data(uint32_t * Block_Pointer,uint8_t length);
                  #endif
```