```
========================================
Name : Arpit Savarkar
========================================
```

```
========================================
main.c
========================================
```

```c
/*
 * main.c - application entry point
 *
 * Author Howdy Pierce, howdy.pierce@colorado.edu
 */

#include "MKL25Z4.h"
#include <string.h>
#include <stdio.h>
#include <stdbool.h>


#include "sysclock.h"
#include "queue.h"
#include "UART.h"
#include "hexdump.h"
#include "cli.h"
#include "test_queue.h"


int main(void)
{
  sysclock_init();
  Init_UART0(38400);

  test_queue();

  // Application Mode that sets the Command Line Interface
  application_mode();

  return 0 ;
}
```

```
========================================
cli.h
========================================
```

```c
/*
 * cli.h
 *
 *  Created on: Nov 2, 2020
 *      Author: root
 */

#ifndef CLI_H_
#define CLI_H_
```

```
#define CLI_SIZE 2048

/*
 * Parse Command entered over UART to Segment and print appropriate result
 *
 * Parameters:
 *   str : String to Parse
 *
 * Returns:
 *   void
 */
void segment_cmd(char *str);

#endif /* CLI_H_ */

========================================
cli.c
========================================
/*
 * cli.c
 *
 *  Created on: Nov 2, 2020
 *      Author: root
 */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include <stdint.h>

#include "hexdump.h"
#include "cli.h"

typedef void (*separate_cmd)(const char* cmd); // Operating Using Function Pointers

typedef struct {
 const char* head;
 separate_cmd func_name;
} cmd_lookup_t;


/*
 * Prints Appropriate string for parsed command
 *
 * Parameters:
 *   str : String to Parse
 *
 * Returns:
 *   void
 */
static void auth(const char* cmd){
 printf("Arpit Savarkar\r\n");
}


/*
 * Prints Appropriate string for hexDump parsed command
 *
 * Parameters:
 *   str : String to Parse
 *
 * Returns:
 *   void
 */
static void dumpHexDump(const char* cmd){
 uint32_t start;
 size_t len;
 if(sscanf(cmd, "dump %x %i", &start, &len) == 2) {
  hexdump((void*) start, len);
 }
}


/*
 * Prints Appropriate string for unrecognized parsed command
 *
 * Parameters:
 *   str : String to Parse
 *
 * Returns:
 *   void
 */
static void unrecognized(const char* cmd){
 printf("Unknown Command: %s\r\n", cmd);
}


/*
 * Prints All the commands Available
 *
 * Parameters:
 *   str : String to Parse
 *
 * Returns:
 *   void
 */
static void help(const char* cmd){
 printf("Command: Author ; Arguments: <> ; Description: Prints a string with your name.\r\n");
 printf("Command: Dump ; Arguments: <Start>, <Len> ; Description: Prints a hexdump of the memory requested \r\n");
 printf("Command: Info ; Arguments: <> ; Description: Prints Build Information.\r\n");
}


/*
 * Prints Build Information
 *
 * Parameters:
 *   str : String to Parse
 *
 * Returns:
 *   void
 */
static void info(const char* cmd){
 printf("Version %s built on %s at %s \r\n", VERSION_TAG, BUILD_MACHINE, BUILD_DATE);
 printf("Commit: %s \r\n", GIT_LOG);
}

static const cmd_lookup_t cmds[] = {
  {"author", auth},
  {"dump", dumpHexDump},
  {"help", help},
  {"info", info},
  {"", unrecognized}
};


/*
 * Parse Command entered over UART to Segment and print appropriate result
 *
 * Parameters:
 *   str : String to Parse
 *
 * Returns:
 *   void
 */
void segment_cmd(char *str) {

 char temp[CLI_SIZE] = {0};
 char* tmp1 = &temp[0];
 char* tmp2 = &str[0];
 int flag = 1;
 char head[20];

 while(*tmp2 != '\0'){
  if(isspace(*tmp2)) {
   if(flag) {
    tmp2++;
   } else {
    *tmp1++ = ' ';
    flag = 1;
   }
  }
  else {
   *tmp1++ = tolower(*tmp2++);
   flag = 0;
  }
 }
 *tmp2 = '\0';
 sscanf(temp, "%s", head);

 int num = sizeof(cmds) / sizeof(cmd_lookup_t);
 for(int i=0; i<num-1; i++) {
  if(strcmp(head, cmds[i].head) == 0) {
   cmds[i].func_name(temp);
   tmp1 = &temp[0];
   tmp2 = &str[0];
   return;
  }
 }

 unrecognized(temp);
 tmp1 = &temp[0];
 tmp2 = &str[0];
 return;

}
```

```
========================================
hexdump.c
========================================
/*
 * hexdump.c
 *
 *  Created on: Nov 2, 2020
 *      Author: arpit.savarkar@colorado.edu
 *
 *  Implementation inspired by Hexdump Implementation of
 *  Howdy Pierce, howdy.pierce@colorado.edu
 */


#include "hexdump.h"


/*
  *     @brief    Returns a character representation of the parameter
  *
  *     Given   a interger, function returns the character repsentation
 *
  *     @param    str :   Interger to be converted
 *
 *   @return    char
 */
char int_to_hexchar(int x) {
 if (x >=0 && x < 10)
  return '0' + x;
 else if (x >= 10 && x < 16)
  return 'A' + x - 10;
 else
  return '-';
}
```

```
/**
  *    @brief    Hex Dump of a memory location upto a selected number of bytes at a specified memory
 *          location
  *
  *    Prints representing a “dump” of the nbytes of memory starting at loc. Bytes are
 *    printed up to 16 bytes per line, separated by newlines.
 *
  *    @param    loc :   Pointer to a char  data  set where the hex dump would be stored
 *   @param nbytes : Number of bytes upto which the hex values of the memory would be printed
 *
  *    @return    void
  */
void hexdump(const void *loc, size_t nbyte) {


   const uint8_t *buf = (uint8_t*) loc;
   const uint8_t *max = (uint8_t*) loc + nbyte;

   if (nbyte > MAX_HEXDUMP_SIZE) {
    nbyte = MAX_HEXDUMP_SIZE;
   }

   while(buf < max ) {
      putchar(int_to_hexchar(((uint32_t)(buf) & 0xF0000000) >> 28));
   putchar(int_to_hexchar(((uint32_t)(buf) & 0x0F000000) >> 24));
   putchar(int_to_hexchar(((uint32_t)(buf) & 0x00F00000) >> 20));
   putchar(int_to_hexchar(((uint32_t)(buf) & 0x000F0000) >> 16));
   putchar('_');
   putchar(int_to_hexchar(((uint32_t)(buf) & 0x0000F000) >> 12));
   putchar(int_to_hexchar(((uint32_t)(buf) & 0x00000F00) >>  8));
   putchar(int_to_hexchar(((uint32_t)(buf) & 0x000000F0) >>  4));
   putchar(int_to_hexchar((uint32_t)(buf) & 0x0000000F));
   putchar(' ');
   putchar(' ');
   for (int j=0; j < STRIDE && buf+j < max; j++) {
     putchar(int_to_hexchar(buf[j] >> 4));
     putchar(int_to_hexchar(buf[j] & 0x0f));
     putchar(' ');
    }
   buf += STRIDE;
   putchar('\r');
   putchar('\n');

   }
}
=====================================
hexdump.h
=====================================
/*
 * hexdump.h
 *
 *  Created on: Nov 2, 2020
 *      Author: root
 */

#ifndef HEXDUMP_H_
#define HEXDUMP_H_

#include <stdio.h>
#include <stdint.h>
#include <string.h>

#define STRIDE 16
#define MAX_HEXDUMP_SIZE 640

/*
  *     @brief    Returns a character representation of the parameter
  *
  *     Given   a interger, function returns the character repsentation
 *
  *     @param    str :   Interger to be converted
 *
 *   @return    char
 */
char int_to_hexchar(int x);


/**
  *     @brief    Hex Dump of a memory location upto a selected number of bytes at a specified memory
 *          location
  *
  *     Prints representing a "dump" of the nbytes of memory starting at loc. Bytes are
 *    printed up to 16 bytes per line, separated by newlines.
 *
  *     @param    loc :   Pointer to a char  data  set where the hex dump would be stored
 *   @param nbytes : Number of bytes upto which the hex values of the memory would be printed
 *
  *     @return    void
  */
void hexdump(const void *loc, size_t nbytes);

#endif /* HEXDUMP_H_ */

=======================================
queue.c
=======================================
/*
 * queue.c
 *
 *  Created on: Nov 2, 2020
 *      Author: arpit.savarkar@colorado.edu /
 */

#include "MKL25Z4.h"
#include "queue.h"
#include <string.h>
#include <stdbool.h>
```

```
#include <stdio.h>
#include <stdint.h>
#include <assert.h>


/*
 * Initializing the FIFO
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   void
 */
void Q_Init(Q_T * q) {
 q->write = 0;
 q->read = 0;
 q->Full_Status = false;
 q->size = 0;
 for (int i=0; i<MAX_SIZE; i++)
     q->data[i] = '_';  // to simplify our lives when debugging
}


/*
 * Returns the FIFO's capacity
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   The capacity, in bytes, for the FIFO
 */
size_t Q_Capacity(Q_T * q) {
 return MAX_SIZE;
}

/*
 * Helper function to check if the cB is empty
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   bool: True/False
 */
bool Q_Empty(Q_T * q) {
 assert(q);
 return (q->write == q->read);
}

/*
 * Helper function to sanity check the current size of the Buffer
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   bool: True/False
 */
int Q_Size(Q_T * q) {
 assert(q);
 return q->size;
}


/*
 * Helper function to check if the cB is at complete capacity
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   bool: True/False
 */
bool Q_Full(Q_T * q){
 assert(q);
 return (Q_Length(q) == MAX_SIZE);
}


/*
 * Returns the number of bytes currently on the FIFO.
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   Number of bytes currently available to be dequeued from the FIFO
 */
size_t Q_Length(Q_T * q){
 uint32_t masking_state;
 size_t val=0;

 masking_state = __get_PRIMASK();
 START_CRITICAL();

 if(q->Full_Status){
  val = MAX_SIZE;
 }
 else if(q->write >= q->read){
  val = q->write - q->read;
 }
 else {
  val = MAX_SIZE - (q->read - q->write);
 }
 END_CRITICAL(masking_state);
 return val;
```

```
}


/*
 * Enqueues data onto the FIFO, up to the limit of the available FIFO
 * capacity.
 *
 * Parameters:
 *   buf      Pointer to the data
 *   nbyte    Max number of bytes to enqueue
 *   Q_T : Queue Object
 *
 * Returns:
 *   The number of bytes actually enqueued, which could be 0. In case
 * of an error, returns -1.
 */
size_t Q_Enqueue(Q_T * q, const void *buf , size_t nbyte) {
 size_t len1 = 0;
 size_t len2=0;
 uint32_t masking_state;

 if(q->Full_Status) {
  return 0;
 }

 masking_state = __get_PRIMASK();
 START_CRITICAL();

 if(Q_Empty(q)) {
  len1 = nbyte;
  q->write = len1;
  if(nbyte == MAX_SIZE) {
   len1 = MAX_SIZE;
   q->Full_Status = true;
   q->write = 0;
  }
  memcpy(q->data, buf, len1);
  q->read = 0;
  q->size += (len1 + len2);
  END_CRITICAL(masking_state);
  return len1 + len2;
 }

 if(q->read < q->write){
  len1 = min(nbyte, MAX_SIZE - q->write);
  memcpy(q->data + q->write, buf, len1);
  q->write += len1;


  if (q->write < MAX_SIZE) {
   END_CRITICAL(masking_state);
     return len1 + len2;
  }
  q->write = 0;
  if(q->read == 0) {
   q->Full_Status = true;
   q->size += (len1 + len2);
   END_CRITICAL(masking_state);
     return len1 + len2;
  }

  nbyte -= len1;
  buf += len1;
 }


 // 2nd stage
 len2 = min(nbyte, q->read - q->write);
 memcpy(q->data + q->write, buf, len2);
 q->write += len2;

 if(q->write == q->read) {
  q->Full_Status = true;
 }
 q->size+= (len1 + len2);

 END_CRITICAL(masking_state);
 return len1 + len2;

}

/*
 * Attempts to remove ("dequeue") up to nbyte bytes of data from the
 * FIFO. Removed data will be copied into the buffer pointed to by buf.
 *
 * Parameters:
 *   buf      Destination for the dequeued data
 *   nbyte    Bytes of data requested
 *   Q_T : Queue Object
 *
 * Returns:
 *   The number of bytes actually copied, which will be between 0 and
 *  nbyte. In case of an error, returns -1.
 */
size_t Q_Dequeue(Q_T * q, void *buf , size_t nbyte) {

 size_t len1 = 0, len2=0;
 uint32_t masking_state;

 masking_state = __get_PRIMASK();

 START_CRITICAL();

 if(Q_Empty(q) && !q->Full_Status) {
  q->size-= (len1 + len2);
  END_CRITICAL(masking_state);
  return len1 + len2;
 }

 q->Full_Status = false;

 len1 = min(nbyte, MAX_SIZE - q->read);
 if((q->write > q->read) && (len1 > q->write - q->read)) {
  len1 = q->write - q->read;
 }
 memcpy(buf, q->data + q->read, len1);
 q->read += len1;
 if(q->read < MAX_SIZE) {
  q->size-= (len1 + len2);
  END_CRITICAL(masking_state);
  return len1 + len2;
 }


 len2 = min(nbyte - len1, q->write);
 memcpy(buf+len1, q->data, len2);
 q->read = len2;

 END_CRITICAL(masking_state);
 return len1 + len2;

}
```

```
=====================================
queue.h
=====================================
```

```
/*
 * queue.h
 *
 *  Created on: Nov 2, 2020
 *      Author: root
 */

#ifndef QUEUE_H_
#define QUEUE_H_

#include <string.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>
#include <assert.h>

#define MAX_SIZE 256
// critical section macro functions
#define START_CRITICAL() __disable_irq()
#define END_CRITICAL(x) __set_PRIMASK(x)
#define min(x,y) ((x)<(y)?(x):(y))

typedef struct {
 int write;
 int read;
 size_t size;
 bool Full_Status;
 uint8_t data[MAX_SIZE];
} Q_T;


/*
 * Initializing the FIFO
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   void
 */
extern void Q_Init(Q_T * q);


/*
 * Enqueues data onto the FIFO, up to the limit of the available FIFO
 * capacity.
 *
 * Parameters:
 *   buf      Pointer to the data
 *   nbyte    Max number of bytes to enqueue
 *   Q_T : Queue Object
 *
 * Returns:
 *   The number of bytes actually enqueued, which could be 0. In case
 * of an error, returns -1.
 */
extern size_t Q_Enqueue(Q_T * q, const void *buf , size_t nbyte);


/*
 * Attempts to remove ("dequeue") up to nbyte bytes of data from the
 * FIFO. Removed data will be copied into the buffer pointed to by buf.
 *
 * Parameters:
 *   buf      Destination for the dequeued data
 *   nbyte    Bytes of data requested
 *   Q_T : Queue Object
 *
 * Returns:
 *   The number of bytes actually copied, which will be between 0 and
 *  nbyte. In case of an error, returns -1.
 */
extern size_t Q_Dequeue(Q_T * q, void *buf , size_t nbyte);


/*
 * Returns the number of bytes currently on the FIFO.
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   Number of bytes currently available to be dequeued from the FIFO
 */
extern size_t Q_Length(Q_T * q);


/*
 * Returns the FIFO's capacity
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   The capacity, in bytes, for the FIFO
 */
extern size_t Q_Capacity(Q_T * q);


/*
 * Helper function to check if the cB is empty
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   bool: True/False
 */
extern bool Q_Empty(Q_T * q);


/*
 * Helper function to check if the cB is at complete capacity
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   bool: True/False
 */
extern bool Q_Full(Q_T * q);


/*
 * Helper function to sanity check the current size of the Buffer
 *
 * Parameters:
 *   Q_T : Queue Object
 *
 * Returns:
 *   bool: True/False
 */
```

```
extern int Q_Size(Q_T * q);


#endif /* QUEUE_H_ */

========================================
sysclock.h
========================================
/*
 * sysclock.h - configuration routines for KL25Z system clock
 *
 * Author Howdy Pierce, howdy.pierce@colorado.edu
 */

#ifndef _SYSCLOCK_H_
#define _SYSCLOCK_H_

#define SYSCLOCK_FREQUENCY (24000000U)

/*
 * Initializes the system clock. You should call this first in your
 * program.
 */
void sysclock_init();

#endif  // _SYSCLOCK_H_

========================================
sysclock.c
========================================
/*
 * sysclock.c - configuration routines for KL25Z system clock
 *
 * Author Howdy Pierce, howdy.pierce@colorado.edu
 *
 * See section 24 of the KL25Z Reference Manual to understand this code
 *
 * Inspired by https://learningmicro.wordpress.com/configuring-device-clock-and-using-systick-system-tick-timer-module-to-generate-software-timings/

 */

#include "MKL25Z4.h"
#include "sysclock.h"


void
sysclock_init()
{
  // Corresponds to FEI mode as shown in sec 24.4.1

  // Select PLL/FLL as clock source
  MCG->C1 &= ~(MCG_C1_CLKS_MASK);
  MCG->C1 |= MCG_C1_CLKS(0);
  // Use internal reference clock as source for the FLL
  MCG->C1 |= MCG_C1_IREFS(1);

  // Select the FLL (by setting "PLL select" to 0)
  MCG->C6 &= ~(MCG_C6_PLLS_MASK);
  MCG->C6 |= MCG_C6_PLLS(0);

  // Select 24 MHz - see table for MCG_C4[DMX32]
  MCG->C4 &= ~(MCG_C4_DRST_DRS_MASK & MCG_C4_DMX32_MASK);
  MCG->C4 |= MCG_C4_DRST_DRS(0);
  MCG->C4 |= MCG_C4_DMX32(1);
}

========================================
test_queue.c
========================================
/*
 * test_queue.c
 *
 *  Created on: Nov 7, 2525
 *      Author: Arpit Savarkar
 *
 *    This method of testing was adapted from Howdy Pierce, Testing of Linked List Based Queue, updated for circul
ar buffer
 */

#include "test_queue.h"
#include "UART.h"

static int g_tests_passed = 0;
static int g_tests_total = 0;
static int g_skip_tests = 0;

Q_T Q;

/*
  *     @brief    Sets up the testing harness for Circular Buffer
 *
  *     @param    void
 *
 *   @return    void
 */
void queue_test_setup() {
 char *str =
      "To be, or not to be: that is the question: \n"
       "Whether 'tis nobler in the mind to suffer \n"
       "The slings and arrows of outrageous fortune, \n"
       "Or to take arms against a sea of troubles, \n"
       "And by opposing end them? To die, to sleep— \n"
       "No more—and by a sleep to say we end \n"
       "The heart-ache and the thousand natural shocks \n"
       "That flesh is heir to, 'tis a consummation \n"
       "Devoutly to be wish'd. To die, to sleep; \n"
```

```
        "To sleep: perchance to dream: ay, there's the rub; \n"
        "For in that sleep of death what dreams may come \n"
        "When we have shuffled off this mortal coil, \n"
        "Must give us pause. \n"
      ;

// const int strs_len = sizeof(strs) / sizeof(const char *);
 char temp_str[1024];
 const int limit = Q_Capacity(&Q);

 test_assert(sizeof(temp_str) > limit);
 test_assert(limit == 256);

 Q_Init(&Q);

 test_equal(Q_Length(&Q), 0);
 test_equal(Q_Dequeue(&Q, temp_str , limit), 0);
 test_equal(Q_Dequeue(&Q, temp_str , 1), 0);
 test_assert(!Q_Full(&Q));

 test_equal(Q_Enqueue(&Q, str , 5), 5);
 test_assert(!Q_Full(&Q));
 test_equal(Q_Length(&Q), 5);
 test_equal(Q_Dequeue(&Q, temp_str , 5), 5);
 test_equal(strncmp(temp_str, str, 5), 0);
 test_equal(Q_Length(&Q), 0);
 test_assert(!Q_Full(&Q));


 test_equal(Q_Enqueue(&Q, str , 10), 10);
 test_equal(Q_Length(&Q), 10);
 test_equal(Q_Dequeue(&Q, temp_str , 5), 5);
 test_equal(Q_Length(&Q), 5);
 test_equal(Q_Dequeue(&Q, temp_str+5 , 5), 5);
 test_equal(Q_Length(&Q), 0);
 test_equal(strncmp(temp_str, str, 10), 0);
 test_assert(!Q_Full(&Q));

 test_equal(Q_Enqueue(&Q, str , limit), limit);
 test_equal(Q_Length(&Q), limit);
 test_assert(Q_Full(&Q));
 test_equal(Q_Enqueue(&Q, str , 1), 0);
 test_assert(Q_Full(&Q));
 test_equal(Q_Dequeue(&Q, temp_str , limit), limit);
 test_assert(!Q_Full(&Q));
 test_equal(Q_Length(&Q), 0);
 test_equal(strncmp(temp_str, str, limit), 0);
//
 test_equal(Q_Enqueue(&Q, str , 25), 25);
 test_assert(!Q_Full(&Q));
 test_equal(Q_Length(&Q), 25);
 test_equal(Q_Dequeue(&Q, temp_str , 23), 23);
 test_assert(!Q_Full(&Q));
 test_equal(Q_Length(&Q), 2);
 test_equal(strncmp(temp_str, str, 23), 0);

 // Following Implementation for testing was necessary since the removing and adding can take place in fixed sizes
 int val = (limit-2) / 4;
 for(int i = 0; i<4; i++) {
  test_equal(Q_Enqueue(&Q, str + i*val , val), val);
  test_equal(Q_Length(&Q), (i+1)*val +2);
 }
 test_equal(Q_Length(&Q), 4*val +2);
 test_equal(Q_Dequeue(&Q, temp_str , 2), 2);
 test_equal(strncmp(temp_str, str+23, 2), 0);

 for(int i=0; i<val*4; i++) {
  test_equal(Q_Dequeue(&Q, temp_str+i , 1), 1);
  test_equal(Q_Length(&Q), val*4 -i -1);
 }

 test_equal(strncmp(temp_str, str, val*4), 0);
 test_equal(Q_Enqueue(&Q, str , 50), 50);
 test_equal(Q_Enqueue(&Q, str+50 , limit), limit-50);
 test_equal(Q_Length(&Q), limit);
 test_assert(Q_Full(&Q));
 test_equal(Q_Dequeue(&Q, temp_str , limit), limit);
 test_equal(Q_Length(&Q), 0);
 test_equal(strncmp(temp_str, str, limit), 0);

 test_equal(Q_Enqueue(&Q, str , 0), 0);
 test_equal(Q_Length(&Q), 0);

}


/*
  *    @brief    Helper Function to track testing
 *
  *    @param    void
 *
 *   @return    void
 */
void test_queue()
{
  g_tests_passed = 0;
  g_tests_total = 0;
  g_skip_tests = 0;

  queue_test_setup();

  printf("%s: passed %d/%d test cases\r\n", __FUNCTION__,
      g_tests_passed, g_tests_total);

  printf("\r\n");
}

====================================
test_queue.h
=======================================
/*
 * test_queue.h
 *
 *  Created on: Nov 7, 2020
 *      Author: root
 */


#include <string.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>
#include <assert.h>

#include "MKL25Z4.h"
#include "queue.h"

#define max(x,y) ((x) > (y) ? (x) : (y))



#define test_assert(value) {                                            \
  g_tests_total++;                                                      \
  if (!g_skip_tests) {                                                  \
    if (value) {                                                        \
      g_tests_passed++;                                                 \
    } else {                                                            \
      printf("ERROR: test failure at line %d\n", __LINE__);             \
      g_skip_tests = 1;                                                 \
    }                                                                   \
  }                                                                     \
}

#define test_equal(value1, value2) {                                    \
  g_tests_total++;                                                      \
  if (!g_skip_tests) {                                                  \
    long res1 = (long)(value1);                                         \
    long res2 = (long)(value2);                                         \
    if (res1 == res2) {                                                 \
      g_tests_passed++;                                                 \
    } else {                                                            \
      printf("ERROR: test failure at line %d: %ld != %ld\n", __LINE__, res1, res2); \
      g_skip_tests = 1;                                                 \
    }                                                                   \
  }                                                                     \
}


/*
  *    @brief    Sets up the testing harness for Circular Buffer
 *
  *    @param    void
 *
 *  @return    void
 */
void queue_test_setup();

/*
  *     @brief    Helper Function to track testing
 *
  *     @param    void
 *
 *  @return    void
 */
void test_queue();

=====================================
UART.c
=====================================
/*
 * UART.c
 *
 *  Created on: Nov 2, 2020
 *      Author: root
 */
#include "UART.h"
#include "sysclock.h"
#include "queue.h"
#include "cli.h"

Q_T TxQ, RxQ;

int __sys_write(int handle, char* buffer, int count) {
 if(buffer == NULL) {
  return -1;
 }
 while(Q_Full(&TxQ)) {
  ; // Wait for the space to openup
 }

 if(Q_Enqueue(&TxQ, buffer, count) != count) {
  return -1;
 }

 if(!(UART0->C2 & UART0_C2_TIE_MASK)) {
  UART0->C2 |= UART0_C2_TIE(1);
 }

 return 0;
}

int __sys_readc(void) {
 char ch;
 if (Q_Dequeue(&RxQ, &ch, 1) != 1){
  return -1;
 }
 return ch;
}


/*
 * Initializing the UART for BAUD_RATE: 38400, Data Size: 8, Parity: None, Stop Bits: 2
 *
 * Parameters:
 *   baud_rate: uint32_t for the requested baud rate
 *
 * Returns:
 *   void
 */
void Init_UART0(uint32_t baud_rate) {
 uint16_t sbr;

 // Enable clock gating for UART0 and Port A
 SIM->SCGC4 |= SIM_SCGC4_UART0_MASK;
 SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

 // Make sure transmitter and receiver are disabled before init
 UART0->C2 &= ~UART0_C2_TE_MASK & ~UART0_C2_RE_MASK;

 // Set UART clock to 24 MHz clock
 SIM->SOPT2 |= SIM_SOPT2_UART0SRC(1);

 // Set pins to UART0 Rx and Tx
 PORTA->PCR[1] = PORT_PCR_ISF_MASK | PORT_PCR_MUX(2); // Rx
 PORTA->PCR[2] = PORT_PCR_ISF_MASK | PORT_PCR_MUX(2); // Tx

 // Set baud rate and oversampling ratio
 sbr = (uint16_t)((SYS_CLOCK)/(baud_rate * UART_OVERSAMPLE_RATE));
 UART0->BDH &= ~UART0_BDH_SBR_MASK;
 UART0->BDH |= UART0_BDH_SBR(sbr>>8);
 UART0->BDL = UART0_BDL_SBR(sbr);
 UART0->C4 |= UART0_C4_OSR(UART_OVERSAMPLE_RATE-1);

 // Disable interrupts for RX active edge and LIN break detect, select one stop bit
 UART0->BDH |= UART0_BDH_RXEDGIE(0) | UART0_BDH_SBNS(1) | UART0_BDH_LBKDIE(0);

 // Don't enable loopback mode, use 8 data bit mode, don't use parity
 UART0->C1 = UART0_C1_LOOPS(0) | UART0_C1_M(0) | UART0_C1_PE(0) | UART0_C1_PT(0);

 // Don't invert transmit data, don't enable interrupts for errors
 UART0->C3 = UART0_C3_TXINV(0) | UART0_C3_ORIE(0)| UART0_C3_NEIE(0)
   | UART0_C3_FEIE(0) | UART0_C3_PEIE(0);

 // Clear error flags
 UART0->S1 = UART0_S1_OR(1) | UART0_S1_NF(1) | UART0_S1_FE(1) | UART0_S1_PF(1);

 // Send LSB first, do not invert received data
 UART0->S2 = UART0_S2_MSBF(0) | UART0_S2_RXINV(0);

 // Enable interrupts. Listing 8.11 on p. 234
 Q_Init(&TxQ);
 Q_Init(&RxQ);

 NVIC_SetPriority(UART0_IRQn, 2); // 0, 1, 2, or 3
 NVIC_ClearPendingIRQ(UART0_IRQn);
 NVIC_EnableIRQ(UART0_IRQn);

 // Enable receive interrupts but not transmit interrupts yet
 UART0->C2 |= UART_C2_RIE(1);

 // Enable UART receiver and transmitter
 UART0->C2 |= UART0_C2_RE(1) | UART0_C2_TE(1);

}


/*
 * Transmits String over to UART
 *
 * Parameters:
 *   str: String to Transmit over UART
 *  count: The Length of the String to transmit
 * Returns:
 *   void
 */
void Send_String(const void* str, size_t count){
 Q_Enqueue(&TxQ, str, count);

 // start transmitting if it isint already
 if (!(UART0->C2 & UART0_C2_TIE_MASK)) {
  UART0->C2 |= UART0_C2_TIE(1);
 }
}


/*
 * Receive the Data from UART to Receive Buffer to store
 *
 * Parameters:
 *   str: String to Transmit over UART
 *  count: The Length of the String to transmit
 * Returns:
 *   void
 */
size_t Receive_String(void* str, size_t count) {
 return Q_Dequeue(&RxQ, str, count);
}


/*
 * Helper function to Clear the Error flags
 *
 * Parameters:
 *   void
 * Returns:
 *   void
 */
static void clearUARTErrors(void) {
 UART0->S1 = UART0_S1_OR(1) | UART0_S1_NF(1) | UART0_S1_FE(1) | UART0_S1_PF(1);
}

/*
 * UART IRQ_Handler
 *
 * Parameters:
 *   void
 * Returns:
 *   void
 */
void UART0_IRQHandler(void) {

 uint8_t ch;

 if (UART0->S1 & (UART_S1_OR_MASK |UART_S1_NF_MASK | UART_S1_FE_MASK | UART_S1_PF_MA
SK)) {
  clearUARTErrors();
  ch = UART0->D;
 }

 if (UART0->S1 & UART0_S1_RDRF_MASK) {
   // received a character
   ch = UART0->D;
   Q_Enqueue(&RxQ, &ch, 1);
  }

 if ( (UART0->C2 & UART0_C2_TIE_MASK) && // transmitter interrupt enabled
    (UART0->S1 & UART0_S1_TDRE_MASK) ) {

  if(Q_Dequeue(&TxQ, &ch, 1)) {
   UART0->D = ch;
  } else {
   // queue is empty so disable transmitter interrupt
   UART0->C2 &= ~UART0_C2_TIE_MASK;
  }
 }
}

/*
 * Handling Command Line Interface between UART and user (terminal)
 *
 * Parameters:
 *   void
 * Returns:
 *   void
 */
static void manage() {
 char xbur[640];
 char* xb = &xbur[0];
 uint8_t c;
 while(c != 0x0D) {
  while (Q_Size(&RxQ) == 0)
   ;
  Q_Dequeue(&RxQ, &c, 1);
  putchar(c);
   if (c != 0x0D || c != 0x0A) {
   if(c != 0x08) {
    *xb = (char)c;
    xb++;
//    *xb = '\0';
    }
   else {
    printf(" \b");
    xb--;
//    *xb = '\0';
   }
  }
  // start transmitter if it isn't already running
  if (!(UART0->C2 & UART0_C2_TIE_MASK)) {
   UART0->C2 |= UART0_C2_TIE(1);
  }
  if(c == '\r'){
   c = 0x0A; // '\n'
   printf("\r\n");
   break;
  }

 }
 *xb = '\0';

 segment_cmd(xbur);
 xb = &xbur[0];
}


/*
 * Application Mode which handles the coordination between UART and Command line interface
 *
 * Parameters:
 *   void
 * Returns:
 *   void
 */
void application_mode() {
 bool status = true;
 char str[] = "Welcome to BreakfastSerial!\r\n";
 Send_String(str, sizeof(str));
 while(status) {
  printf("? ");
  manage();
 }
}
```

```
========================================
UART.h
========================================
/*
 * UART.h
 *
 *  Created on: Nov 2, 2020
 *      Author: root
 */

#ifndef UART_H_
#define UART_H_

#include "MKL25Z4.h"
#include <string.h>
#include <stdio.h>
#include <stdbool.h>

#define UART_OVERSAMPLE_RATE  (15)
#define BUS_CLOCK     (24e6)
#define SYS_CLOCK    (24e6)

// critical section macro functions
#define START_CRITICAL() __disable_irq()
#define END_CRITICAL(x) __set_PRIMASK(x)


/*
 * Initializing the UART for BAUD_RATE: 38400, Data Size: 8, Parity: None, Stop Bits: 2
 *
 * Parameters:
 *   baud_rate: uint32_t for the requested baud rate
 *
 * Returns:
 *   void
 */
void Init_UART0(uint32_t baud_rate);


/*
 * Transmits String over to UART
 *
 * Parameters:
 *   str: String to Transmit over UART
 *  count: The Length of the String to transmit
 * Returns:
 *   void
 */
void Send_String(const void* str, size_t count);


/*
 * Receive the Data from UART to Receive Buffer to store
 *
 * Parameters:
 *   str: String to Transmit over UART
 *  count: The Length of the String to transmit
 * Returns:
 *   void
 */
size_t Receive_String(void* str, size_t count);


/*
 * Application Mode which handles the coordination between UART and Command line interface
 *
 * Parameters:
 *   void
 * Returns:
 *   void
 */
void application_mode();

#endif /* UART_H_ */
```