Code

main.c

```
/*
 * main.c - application entry point
 *
 * Author Howdy Pierce, howdy.pierce@colorado.edu
 */
#include "sysclock.h"
#include "uart.h"
#include "queue.h"
#include "test_queue.h"
#include "hexdump.h"

int main(void)
{
  sysclock_init();

  // TODO: initialize the UART here
  Init_UART0(BAUD_RATE);

  test_cbfifo();

  char str[] = "Welcome to BreakfastSerial!\r\n";
  send_String(str, sizeof(str));
  // enter infinite loop
  while (1) {
        printf("? ");
        cmd_accumulate();
  }
  return 0 ;
}
```

Queue.c

```
/*
 * @File              cbfifo.c
 * @Brief             The file contains functions related to handling of the
Circular Buffer
 *                          implementation of the FIFO. It includes function to
enqueue and dequeue
 *                          elements in the FIFO and functions to destroy it and
return the length
 *                          and capacity of the queue.
 * @Author            Ruchit Naik
 * @Date              03-Nov-2021
 *
 * @InstituteUniversity of Colorado, Boulder
 * @Course            ECEN 8513: Principles of Embedded Software
 *
 * @Attribute
 */

#include "queue.h"

/*//Initializing the structure to use the elements while calling function on
cbfifo
```

```
CBfifo CBfifo_t ={
             .size = 0,
             .readp = 0,
             .writep = 0,
             .length = 0,
};*/

void cbfifo_Init(CBfifo * queue){
             queue->size = 0;
             queue->readp = 0;
             queue->writep = 0;
             for(int i = 0; i<MAXSIZE; i++){
                    queue->data[i] = 0;
             }
             queue->full_queue = false;
}




bool cbfifo_full(CBfifo *queue){
      return (cbfifo_length(queue) == MAXSIZE);
}

bool cbfifo_empty(CBfifo *queue){
      return (cbfifo_length(queue) == 0);
}

int cbfifo_size(CBfifo *queue){
      return queue->size;
}



/*
 * @Function cbfifo_length
 * @Param          none
 * @Returns        Number of bytes currently available to be dequeued from the
FIFO
 * @Description    Returns the number of bytes currently on the FIFO.
 */
size_t cbfifo_length(CBfifo *queue){
      /*uint8_t length = 0;
      length = (CBfifo_t.writep - CBfifo_t.readp) & (MAXSIZE - 1);
      return length;*/
      size_t value = 0;
      if(queue->full_queue){
             value = MAXSIZE;
      }
      else if(queue->writep >= queue->readp){
             value = queue->writep - queue->readp;
      }
      else{
             value = MAXSIZE - (queue->readp - queue->writep);
      }
      return value;
}
```

```
/*
 * @Function cbfifo_enqueue
 * @Param               buf - The pointer to the location from where data is to be
entered in
 *                                 the fifo
 *                            nbytes - Number of bytes to be enqueued in fifo
 * @Returns             The number of bytes actually copied, which will be between 0
and nbyte.
 * @Description         Enqueues data onto the FIFO, up to the limit of the available
FIFO
 *                            capacity.
 */
size_t cbfifo_enqueue(CBfifo *queue, const void *buf, size_t nbyte){
      size_t len1 = 0;
      size_t len2 = 0;

      if(queue->full_queue){
            return 0;
      }

      if(cbfifo_empty(queue)){
            len1 = nbyte;
            queue->writep = len1;
            if(nbyte == MAXSIZE){
                  len1 = MAXSIZE;
                  queue->full_queue = true;
                  queue->writep = 0;
            }
            memcpy(queue->data, buf, len1);
            queue->readp = 0;
            queue->size = queue->size + len1 + len2;
            return len1 + len2;
      }

      if(queue->readp < queue->writep){
            len1 = min(nbyte, MAXSIZE - queue->writep);
            memcpy(queue->data + queue->writep, buf, len1);
            queue->writep = queue->writep + len1;

            if(queue->writep < MAXSIZE){
                  return len1 + len2;
            }

            queue->writep = 0;
            if(queue->readp == 0){
                  queue->size = queue->size + (len1 + len2);
            //Updating the queue size after enqueue
                  queue->full_queue = true;
                  return len1 + len2;
            }

            nbyte = nbyte - len1;
            buf = buf + len1;
                                        //Updating the buf pointer over the buffer
roll-over point
```

```
        }

        //After roll-over stage
        len2 = min(nbyte, queue->readp - queue->writep);
        memcpy(queue->data + queue->writep, buf, len2);
        queue->writep = queue->writep + len2;
                                //Furthering updating the write pointer

        if(queue->writep == queue->readp){
                queue->full_queue = true;
        }

        queue->size = queue->size + len1 + len2;

        return (len1 + len2);
}



/*
 * @Function cbfifo_dequeue
 * @Param               buf - The pointer to the location where the dequeued data is
to be fetched
 *                              nbytes - Number of bytes to be dequeued from fifo
 * @Returns             The number of bytes actually copied, which will be between 0
and nbyte.
 * @Description         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.
 */
size_t cbfifo_dequeue(CBfifo *queue, void *buf, size_t nbyte){
        size_t len1 = 0;
        size_t len2 = 0;

        if(cbfifo_empty(queue) && !queue->full_queue){
                queue->size = queue->size - (len1 + len2);
                return 0;
        }

        queue->full_queue = false;

        len1 = min(nbyte, MAXSIZE - queue->readp);
        if((queue->writep > queue->readp) && (len1 > queue->writep - queue->readp))
{
                len1 = queue->writep - queue->readp;
        }
        memcpy(buf, queue->data + queue->readp, len1);
        queue->readp = queue->readp + len1;
                                        //Updating the read pointer

        if(queue->readp < MAXSIZE){
                queue->size = queue->size - (len1 + len2);
                                                //Updating size of the buffer
                return (len1 + len2);
        }

        //Handling the roll-over condition of the buffer
```

```
      len2 = min(nbyte - len1, queue->writep);
                          //check the remaining length of the buffer
      memcpy(buf+len1, queue->data, len2);
                          //Return the dequeue data to the return buffer
      queue->readp = len2;

      return (len1 + len2);
}



/*
 * @Function cbfifo_capacity
 * @Param           none
 * @Returns         the current capacity of the fifo in bytes
 * @Description     Returns the current capacity of the fifo
 */
size_t cbfifo_capacity(){
      return MAXSIZE;
}
```

Queue.h

```
/*
 * queue.h - a fixed-size FIFO implemented via a circular buffer
 *
 * Author: Howdy Pierce, howdy.pierce@colorado.edu
 *
 */

#ifndef QUEUE_H_
#define QUEUE_H_

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

#define MAXSIZE           256                         //Static size of cbfifo
#define min(x,y)    ((x)<(y)?(x):(y))

/**
 * The structure stores the necessary elements to track the states of the circular
 * buffer FIFO
 */
typedef struct CBfifo{
      uint8_t data[MAXSIZE];
      uint8_t size;
      int readp;
      int writep;
      bool full_queue;
}CBfifo;



void cbfifo_Init(CBfifo *queue);
```

```
/*
 * 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
 *
 * Returns:
 *   The number of bytes actually enqueued, which could be 0. In case
 * of an error, returns -1.
 */
size_t cbfifo_enqueue(CBfifo *queue, 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
 *
 * Returns:
 *   The number of bytes actually copied, which will be between 0 and
 * nbyte.
 *
 * To further explain the behavior: If the FIFO's current length is 24
 * bytes, and the caller requests 30 bytes, cbfifo_dequeue should
 * return the 24 bytes it has, and the new FIFO length will be 0. If
 * the FIFO is empty (current length is 0 bytes), a request to dequeue
 * any number of bytes will result in a return of 0 from
 * cbfifo_dequeue.
 */
size_t cbfifo_dequeue(CBfifo *queue, void *buf, size_t nbyte);


/*
 * Returns the number of bytes currently on the FIFO.
 *
 * Parameters:
 *   none
 *
 * Returns:
 *   Number of bytes currently available to be dequeued from the FIFO
 */
size_t cbfifo_length();


/*
 * Returns the FIFO's capacity
 *
 * Parameters:
 *   none
 *
 * Returns:
 *   The capacity, in bytes, for the FIFO
 */
```

```
size_t cbfifo_capacity();

bool cbfifo_full(CBfifo *queue);

bool cbfifo_empty(CBfifo *queue);

int cbfifo_size(CBfifo *queue);

#endif /* QUEUE_H_ */
```

Uart.c

```
/*
 * @File            uart.c
 * @Brief           The file contains functions related to handling the UART on
the KL25Z board.
 *                          It also includes the IRQ Hanlder for the UART Tx and Rx
and function to bind
 *                          stdio function with the UART using REDLIB library.
 * @Author          Ruchit Naik
 * @Date            03-Nov-2021
 *
 * @InstituteUniversity of Colorado, Boulder
 * @Course          ECEN 8513: Principles of Embedded Software
 *
 * @Attribute
 */

#include "uart.h"
#include "queue.h"

CBfifo RxQ, TxQ;

int __sys_write(int handle, char* buffer, int count){
	if(buffer == NULL){
		//Return error if null character is passed
		return -1;
	}

	while(cbfifo_full(&TxQ)){
		;			//Wait if transmitter buffer is full
	}

	if(cbfifo_enqueue(&TxQ, buffer, count) != count){
		//Error in enqueue which is propogated further
		return -1;
	}
	if(!(UART0->C2 & UART0_C2_TIE_MASK)){
		UART0->C2 |= UART0_C2_TIE(1);
	}
	return 0;
}

int __sys_readc(void){
	char chatr;
	if(cbfifo_dequeue(&RxQ, &chatr, 1) != 1){
		return -1;
```

```
        }
        if((chatr == '\r') || (chatr == '\r')){
            chatr = '\r';
            printf("%c", chatr);
            chatr = '\n';
            printf("%c", chatr);
        }
        else{
            printf("%c", chatr);
        }
        return chatr;
    }

    void Init_UART0(uint32_t baud_rate){
        uint16_t sbr;

        //Enable clock gating for UART00 and PORTA
        SIM->SCGC4 |= SIM_SCGC4_UART0_MASK;
        SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

        //Disabling transmitter and receiver before init
        UART0->C2 &= ~UART0_C2_TE_MASK & ~UART0_C2_RE_MASK;

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

        //Set pins to UART0 Tx and Rx
        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 bau rate and oversampling ratio
        sbr = (uint16_t)((SYS_CLOCK)/(baud_rate * UART_OVERSAMPLE_RATE));
        UART0->BDH &= ~UART_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 two
    stop bit
        UART0->BDH |= UART0_BDH_RXEDGIE(0) | UART0_BDH_SBNS(STOP_CONFIG) |
    UART_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(BIT_MODE) |
    UART0_C1_PE(PARITY_ENABLE) | UART0_C1_PT(0);
        //Don't invert transit data, do enable interrupt 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);

        //Configure to send LSB first
        UART0->S2 = UART0_S2_MSBF(0) | UART0_S2_RXINV(0);

        //Initializing Queue for further use
        cbfifo_Init(&TxQ);
        cbfifo_Init(&RxQ);

        //Enable UART0 interrupts
        NVIC_SetPriority(UART0_IRQn, 2);
        NVIC_ClearPendingIRQ(UART0_IRQn);
        NVIC_EnableIRQ(UART0_IRQn);

        //Enable receiver interrupts
        UART0->C2 |= UART_C2_RIE(1);

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


    void UART0_IRQHandler(void){
        uint8_t chatr;
        if(UART0->S1 & (UART_S1_OR_MASK | UART_S1_NF_MASK |
                            UART_S1_FE_MASK | UART_S1_PF_MASK)){
            //Clearing the error flags
            UART0->S1 |= UART0_S1_OR_MASK | UART0_S1_NF_MASK |
                            UART0_S1_FE_MASK | UART0_S1_PF_MASK;
            //Read the data register
            chatr = UART0->D;
        }

        if(UART0->S1 & UART0_S1_RDRF_MASK){
            //Entered here when character is received
            chatr = UART0->D;
            if(!cbfifo_full(&RxQ)){
                //If Rx queue not full
                cbfifo_enqueue(&RxQ, &chatr, 1);
            }
        }

        if((UART0->C2 & UART0_C2_TIE_MASK) && (UART0->C2 & UART0_S1_TDRE_MASK)){
            //Entered here when Tx buffer is empty. Can send another character
            if(cbfifo_dequeue(&TxQ, &chatr, 1) == 1){
                UART0->D = chatr;
            }
            else{
                //Queue empty -> Tx interrupts disabled
                UART0->C2 &= ~UART0_C2_TIE_MASK;
            }
        }
    }


    void send_String(const void* str, size_t count){
        cbfifo_enqueue(&TxQ, str, count);

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

```
size_t receive_String(void* str, size_t count){
	return cbfifo_dequeue(&RxQ, str, count);
}

void cmd_accumulate(void){
	char acc_buf[640];
	char *ptr_acc = &acc_buf[0];
	uint8_t ch;

	while(ch != '\r'){				//In loop until terminating
character is received
		while(cbfifo_empty(&RxQ)){
			;					//Wait if Rx queue is
still empty to handle the user input commands
		}

		cbfifo_dequeue(&RxQ, &ch, 1);
		putchar(ch);
		if((ch != '\r') && (ch != '\n')){
			//Not handling backspace here.
			*ptr_acc = (char)ch;
			ptr_acc++;
			*ptr_acc = '\0';		//Adding terminating char at the
end, it is overwritten when we receive next char
		}

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

		if(ch == '\r'){
			ch = '\n';
			printf("\r\n");
			break;
		}
	}

	Process_Message(acc_buf);		//Segmenting the received cmd into token
to handle function calls
	ptr_acc = &acc_buf[0];			//Resetting the pointer back to
initial location for next accumulation
}
```

Uart.h

```
/*
 * @File		uart.c
 * @Brief		The file contains functions related to handling the UART on
the KL25Z board.
 *			It also includes the IRQ Hanlder for the UART Tx and Rx
and function to bind
 *			stdio function with the UART using REDLIB library.
 * @Author		Ruchit Naik
 * @Date		03-Nov-2021
 *
 * @InstituteUniversity of Colorado, Boulder
 * @Course		ECEN 8513: Principles of Embedded Software
 *
 * @Attribute
 */

#ifndef UART_H_
#define UART_H_

#include <stdint.h>
#include "MKL25Z4.h"
#include "cli.h"

#define UART_OVERSAMPLE_RATE     15
#define SYS_CLOCK                      (24e6)

/*******************************************************
 * UART Configuration
 ******************************************************/
#define      BAUD_RATE                        38400
#define DATA_SIZE                      8
#define PARITY                                None
#define      STOP_BITS                        2

#if (DATA_SIZE == 8)
#define BIT_MODE          0
#else
#define BIT_MODE          1
#endif

#if (PARITY == None)
#define PARITY_ENABLE     0
#else
#define PARITY_ENABLE     1
#endif

#if (STOP_BITS == 1)
#define STOP_CONFIG       0
#else
#define STOP_CONFIG       1
#endif

void send_String(const void* str, size_t count);
size_t receive_String(void* str, size_t count);
void Init_UART0(uint32_t baud_rate);
void cmd_accumulate(void);

#endif /* UART_H_ */
```

Hexdump.c

```
/*
 * @File            hexdump.c
 * @Brief           The file contains functions related to handling the
hexadecimal dump of the RAM.
 *                        It contains the function to print hexdump based on the
given origin on the memory
 *                        and size.
 * @Author          Ruchit Naik
 * @Date            07-Nov-2021
 *
 * @InstituteUniversity of Colorado, Boulder
 * @Course          ECEN 8513: Principles of Embedded Software
 *
 * @Attribute
 */
#include "hexdump.h"


/**
 * @Function: int_to_hexchar()
 * @Parameters:  x - Integer which needs to converted hexdecimal value of the
corresponding character
 * @Description: The function converts the given number to hexadecimal character.
It returns the char
 *                          conversion of the interger value given as input to the
function.
 */
char int_to_hexchar(int ch){
      if (ch >=0 && ch < 10)
             return '0' + ch;
      else if (ch >= 10 && ch < 16)
             return 'A' + ch - 10;
      else
             return '-';
}



/**
 * @Function:       hexdump()
 * @Parameters:  *loc   - It is the pointer to the location from where the hex
dump is to be executed
 *                          nbytes - It is the number of bytes which to be dumped
from the location pointed by *loc
 * @Description: The function returns the hex dump from the *loc pointer in the
memory till the bytes given
 *                          by the user. It would return the hex dump at in the
buffer or array to which *str points to.
 *                          Function would return empty *str in the case of error
when *str is not large enough to accomodate
 *                          the entire hex dump.
 */
void hexdump(const void *loc, size_t nbyte){
        const uint8_t *ptr = (uint8_t*) loc;
        const uint8_t *max = (uint8_t*) loc + nbyte;

        if (nbyte > MAX_HEXDUMP_SIZE) {
               //Handling error condition if larger hexdump is requested
               nbyte = MAX_HEXDUMP_SIZE;
        }

        while(ptr < max ) {
                 putchar(int_to_hexchar(((uint32_t)(ptr) & 0xF0000000) >> 28));
                   putchar(int_to_hexchar(((uint32_t)(ptr) & 0x0F000000) >> 24));
                   putchar(int_to_hexchar(((uint32_t)(ptr) & 0x00F00000) >> 20));
                   putchar(int_to_hexchar(((uint32_t)(ptr) & 0x000F0000) >> 16));
                   putchar('_');
                   putchar(int_to_hexchar(((uint32_t)(ptr) & 0x0000F000) >> 12));
                   putchar(int_to_hexchar(((uint32_t)(ptr) & 0x00000F00) >>  8));
```

```
                    putchar(int_to_hexchar(((uint32_t)(ptr) & 0x000000F0) >>  4));
                    putchar(int_to_hexchar((uint32_t)(ptr) & 0x0000000F));
                    putchar(' ');
                    putchar(' ');
                    for (int j=0; j < BYTES_PER_LINE && ptr+j < max; j++) {
                      putchar(int_to_hexchar(ptr[j] >> 4));
                      putchar(int_to_hexchar(ptr[j] & 0x0f));
                      putchar(' ');
                          }
                    ptr += BYTES_PER_LINE;
                    putchar('\r');
                    putchar('\n');

        }
}
```

Hexdump.h

```
/*
 * @File            hexdump.h
 * @Brief           The file contains functions related to handling the
hexadecimal dump of the RAM.
 *                          It contains the function to print hexdump based on the
given origin on the memory
 *                          and size.
 * @Author          Ruchit Naik
 * @Date            07-Nov-2021
 *
 * @InstituteUniversity of Colorado, Boulder
 * @Course          ECEN 8513: Principles of Embedded Software
 *
 * @Attribute
 */

#ifndef HEXDUMP_H_
#define HEXDUMP_H_

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

#define BYTES_PER_LINE                  16
#define MAX_HEXDUMP_SIZE          640

/**
 * @Function:       hexdump()
 * @Parameters:  *loc    - It is the pointer to the location from where the hex
dump is to be executed
 *                           nbytes - It is the number of bytes which to be dumped
from the location pointed by *loc
 * @Description: The function returns the hex dump from the *loc pointer in the
memory till the bytes given
 *                           by the user. It would return the hex dump at in the
buffer or array to which *str points to.
 *                           Function would return empty *str in the case of error
when *str is not large enough to accomodate
 *                           the entire hex dump.
 */
void hexdump(const void *loc, size_t nbyte);
```

```
/**
 * @Function: int_to_hexchar()
 * @Parameters:  x - Integer which needs to converted hexdecimal value of the
corresponding character
 * @Description: The function converts the given number to hexadecimal character.
It returns the char
 *                          conversion of the interger value given as input to the
function.
 */
char int_to_hexchar(int x);

#endif /* HEXDUMP_H_ */
```

cli.c

```
/*
 * @File           cli.c
 * @Brief          The file contains functions related to handling the UART on
the KL25Z board.
 *                         It also includes the IRQ Hanlder for the UART Tx and Rx
and function to bind
 *                         stdio function with the UART using REDLIB library.
 * @Author         Ruchit Naik
 * @Date           08-Nov-2021
 *
 * @InstituteUniversity of Colorado, Boulder
 * @Course         ECEN 8513: Principles of Embedded Software
 *
 * @Attribute
 */

#include "cli.h"

typedef void (*command_handler_t)(int, char *argv[]);

//Look-up table data structure
typedef struct{
	const char *name;
	command_handler_t handler;
	const char *help_string;
}command_table_t;


static void handle_author(int argc, char *argv[]){
	printf("Ruchit Naik \r\n");
}


static void handle_unknown(int argc, char *argv[]){
	printf("Invalid Command\r\n");
}


static void handle_dump(int argc, char *argv[]){
	uint32_t origin = 0;
	size_t len = 0;
```

```
    if(argc != 3){
        handle_unknown(argc, argv);
        return;
    }
    sscanf(*(&argv[1]), "%x", &origin);
    sscanf(*(&argv[2]), "%i", &len);
    hexdump((void*)origin, len);
}


static void handle_help(int argc, char *argv[]){
    printf("Command: Author | Arg: <> | Brief: Prints a string with your
name.\r\n");
    printf("Command: Dump | Arg: <Start>, <Len> | Brief: Prints a hexdump of
the memory requested; <Start> in hex; <Len> any format.\r\n");
    printf("Command: Info | Arg: <> | Brief: Prints Build Information.\r\n");
}

static void handle_info(int argc, char *argv[]){
//  printf("Version %s built on %s at %s \r\n", VERSION_TAG, BUILD_MACHINE,
BUILD_DATE);
//  printf("Commit: %s \r\n", GIT_LOG);
    printf("info coming up\r\n");
}



//Cmd Look-up table
static const command_table_t command[] = {
        {"author", handle_author, "Command: Author | Arg: <> | Brief: Prints
a string with your name.\r\n"},
        {"dump", handle_dump, "Command: Dump | Arg: <Start>, <Len> | Brief:
Prints a hexdump of the memory requested; <Start> in hex; <Len> any format.\r\n"},
        {"help", handle_help, "Command: Help | Arg: <> | Brief: Prints the
user help info for all the commands in the lookup table.\r\n"},
        {"info", handle_info, "Command: Info | Arg: <> | Brief: Prints Build
Information.\r\n"},
        {"", handle_unknown}
};

static const int num_commands = sizeof(command)/sizeof(command_table_t);


void Process_Message(char *input){
    char *ptr = &input[0];
    char *end;

    //To find the end pointer
    for(end = input; *end != '\0'; end++);

    //Tokenize input in place
    bool in_token = false;
    char *argv[10];
    int argc = 0;
    memset(argv, 0, sizeof(argv));         //initializing argv with 0

    for(ptr = input; ptr < end; ptr++){
            if(*ptr == ' '){                       //Check on spaces
```

```
                                if(!in_token){
                                        *ptr = ' ';                             //Ignore
spaces if not a token
                                }
                                else{
                                        *ptr = '\0';                    //Fill up the space
after token with \0
                                        in_token = false;
                                }
                        }
                        else{                                                   //Managing the
token from the input string token
                                if(!in_token){
                                        argv[argc] = ptr;                       //pointing to
first argument on the accumulated buffer
                                        argc++;
        //Increment argc for next argument
                                }
                                in_token = true;
                        }
        }

        argv[argc] = NULL;
        if (argc == 0){                                                 //No command entered
                return;
        }

        //TODO: Dispatch argc/argv to handler
        for(int i=0; i<num_commands; i++){
                if(strcasecmp(argv[0], command[i].name) == 0){
                        command[i].handler(argc, argv);
        //calls corresponding function handler
                        ptr = &input[0];
                        return;
                }
        }

        handle_unknown(argc, argv);
        ptr = &input[0];
        return;
}
```

cli.h

```
/*
 * @File                cli.h
 * @Brief               The file contains functions related to handling the UART on
the KL25Z board.
 *                              It also includes the IRQ Hanlder for the UART Tx and Rx
and function to bind
 *                              stdio function with the UART using REDLIB library.
 * @Author              Ruchit Naik
 * @Date                08-Nov-2021
 *
 * @InstituteUniversity of Colorado, Boulder
 * @Course              ECEN 8513: Principles of Embedded Software
 *
 * @Attribute
 */

#ifndef CLI_H_
#define CLI_H_

#include <stdbool.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <stdint.h>
#include "hexdump.h"


void Process_Message(char *input);



#endif /* CLI_H_ */
```