```
/*
 * binary_log.c
 *
 *  Created on: Mar 8, 2017
 *      Author: jacob
 */

#include "binary_log.h"

CircBufStatus BinLogBufferInit(CircBuf* CB, uint32_t size){

        if(!CB) return PTR_ERROR_BUF;
        if(size == 0) return INIT_FAILURE;
        CB->buffer = (BinLog**)malloc(sizeof(BinLog*) * size);
        if(!(CB->buffer)) return HEAP_FULL;
        CB->head = CB->buffer;
        CB->tail = CB->buffer;
        CB->length = size;
        CB->count = 0;
        return SUCCESS_BUF;
}

CircBufStatus BinLogBufferAdd(CircBuf* CB, BinLog* item){

        BinLog** temp_ptr;

        if(!CB || !(CB->buffer)) return PTR_ERROR_BUF;
        if(CB->count == CB->length) return OVERWRITE;

        if(CB->count > 0)
                CB->head = ((BinLog**)CB->head < (BinLog**)CB->buffer + CB->length -
1 ? ((BinLog**)CB->head) + 1 : CB->buffer);

        temp_ptr = (BinLog**)CB->head;
        *temp_ptr = item;
        (CB->count)++;
        return SUCCESS_BUF;
}


CircBufStatus BinLogBufferRemove(CircBuf* CB, BinLog** item){

        if(!CB || !(CB->buffer) ) return PTR_ERROR_BUF;
        if(CB->count == 0){
                return ITEM_REMOVE_FAILURE;
        }
        if(CB->count == 1){ // return to empty state
                if(item) **item = **(((BinLog**)CB->tail));
                free(*((BinLog**)(CB->tail)));
                CB->tail = CB->buffer;
                CB->head = CB->buffer;
                CB->count = 0;
                return SUCCESS_BUF;
        }
        if(item) **item = **((BinLog**)CB->tail);
        free(*((BinLog**)(CB->tail)));
        CB->tail = (((BinLog**)CB->tail) < ((BinLog**)CB->buffer) + CB->length - 1 ?
                        ((BinLog**)CB->tail) + 1 : CB->buffer);
        (CB->count)--;
        return SUCCESS_BUF;
}
```

```
CircBufStatus BinLogBufferFull(CircBuf* CB){

        if(!CB) return PTR_ERROR_BUF;
        if(CB->count == CB->length) return BUFFER_FULL;
        else return BUFFER_NOT_FULL;
}

CircBufStatus BinLogBufferEmpty(CircBuf* CB){

        if(!CB) return PTR_ERROR_BUF;
        if(CB->count == 0) return BUFFER_EMPTY;
        else return BUFFER_NOT_EMPTY;
}

uint32_t BinLogBufferCount(CircBuf* CB){
        if(!CB) return PTR_ERROR_BUF;
        else return CB->count;
}

CircBufStatus BinLogBufferPeek(CircBuf* CB, BinLog** item_n, uint32_t n){
// returns nth oldest item
        if(!CB || !item_n || !(CB->buffer)) return PTR_ERROR_BUF;
        if(n > CB->count || n < 1) return INVALID_PEEK;

        *item_n = (((BinLog**)CB->tail) + n - 1 > ((BinLog**)CB->buffer) + (CB-
>length - 1) ?
                        *(((BinLog**)CB->tail) - CB->length + n - 1) : *
(((BinLog**)CB->tail) + n - 1));
        return SUCCESS_BUF;
}

CircBufStatus BinLogBufferClear(CircBuf* CB){

        uint32_t i;
        if(!CB || !(CB->buffer)) return PTR_ERROR_BUF;
        for(i = 0; i < CB->length; ++i)
                BinLogBufferRemove(CB, NULL);
        return SUCCESS_BUF;
}


CircBufStatus BinLogBufferDestroy(CircBuf* CB){

        uint32_t i;
        if(!CB || !(CB->buffer)) return PTR_ERROR_BUF;
        for(i = 0; i < CB->length; ++i)
                BinLogBufferRemove(CB, NULL);
        free(CB->buffer);
        CB->length = 0;
        CB->count = 0;
        CB->head = NULL;
        CB->tail = NULL;
        CB->buffer = NULL;
        return SUCCESS_BUF;
}

BinLogStatus BinLogCreate(BinLog** BL, BinLogID ID, uint8_t* payload, uint32_t length)
{

        *BL = (BinLog*) malloc(sizeof(BinLog));
        if(*BL == NULL) return BINLOG_HEAP_FULL;
        (*BL)->ID = ID;
        (*BL)->size = length;
        my_memmove(payload, (*BL)->payload, length);
        return BINLOG_SUCCESS;
}

BinLogStatus BinLogEvent(CircBuf* CB, BinLogID ID, uint8_t* payload, uint32_t length){

        BinLog* BL;
        if(BinLogCreate(&BL, ID, payload, length) == BINLOG_HEAP_FULL);
                return BINLOG_HEAP_FULL;
        if(BinLogBufferAdd(CB, BL) == OVERWRITE) return BINLOGBUF_FULL;
        else BINLOG_SUCCESS;
}

BinLogStatus BinLogChar(CircBuf* CB, BinLogID ID, uint8_t character){

        // search for existing ID
        uint32_t i = 1;
        BinLog* BL = (BinLog*)malloc(sizeof(BinLog));
        if(BL == NULL) return BINLOG_HEAP_FULL;

        while(i <= CB->count){
                BinLogBufferPeek(CB, &BL, i);
                if(BL->ID == ID){
                        BL->payload[BL->size % MAX_BINLOG_PAYLOAD_SIZE] = character;
                        ++(BL->size);
                        if(BinLogBufferAdd(CB, BL) == OVERWRITE) return
BINLOGBUF_FULL;
                        else return BINLOGCHAR_NO_EVENT_CREATED;
                }
                else{
                        ++i;
                }
        }
        // create new BinLog item if ID doesn't exist in CB
        BinLogEvent(CB, ID, &character, 1);
        return BINLOGCHAR_EVENT_CREATED;
}

BinLogStatus BinLogSendData(CircBuf* CB, BinLogID ID){

        uint32_t i = 1;
        BinLog* BL = (BinLog*)malloc(sizeof(BinLog));
        if(BL == NULL) return BINLOG_HEAP_FULL;

        while(i <= CB->count){
                BinLogBufferPeek(CB, &BL, i);
                if(BL->ID == ID){
                        log_string("\nNumber of characters: ");
                        log_integer(BL->size);
                        if(ID != DATA_MISC_COUNT) {
                                log_string("\nCharacters received: ");
                                uart_send_byte_n(BL->payload, BL->size);
                        }
                        free(BL);
                        return BINLOG_CHARS_FOUND;
                }
                else{
                        ++i;
                }
        }
        log_string("\nNumber of characters: 0");
        log_string("\nCharacters received: none");
        free(BL);
        return BINLOG_NO_CHARS_FOUND;
}
/*
 * binary_log.h
 *
 *  Created on: Mar 8, 2017
 *      Author: jacob
 */

#ifndef SOURCES_BINARY_LOG_H_
#define SOURCES_BINARY_LOG_H_

#include <stdlib.h>
#include <stdint.h>
#include "circbuf.h"
#include "defines.h"
#include "memory.h"
#include "uart.h"
#include "uartbuf.h"
#include "log.h"

// Macro functions obtained using ascii table.
#define IS_ALPHA(X) ((X >= 'A' && X <= 'Z') || (X >= 'a' && X <= 'z'))

#define IS_NUMERIC(X) (X >= '0' && X <= '9')

#define IS_PUNCTUATION(X) ((X >= '!' && X <= 0x47) || (X >= ':' && X <= '@') || (X >=
'[' && X <= '`') || (X >= '{' && X <= '~'))

#define IS_CTL(X) ((X >= 0 && X <= 0x20) || (X == 0x7F))

typedef enum BinLogID_e{
        LOGGER_INITIALIZED,
        GPIO_INITIALIZED,
        SYSTEM_INITIALIZED,
        SYSTEM_HALTED,
        INFO,
        WARNING,
        ERROR,
        DATA_RECEIVED,
        DATA_ANALYSIS_STARTED,
        DATA_ALPHA_COUNT,
        DATA_NUMERIC_COUNT,
        DATA_PUNCTUATION_COUNT,
        DATA_MISC_COUNT,
        DATA_ANALYSIS_COMPLETED
}BinLogID;

typedef enum BinLogStatus_e{
        BINLOG_SUCCESS,
        BINLOG_HEAP_FULL,
        BINLOG_PTR_ERROR,
        BINLOGBUF_FULL,
        BINLOGCHAR_EVENT_CREATED,
        BINLOGCHAR_NO_EVENT_CREATED,
        BINLOG_CHARS_FOUND,
        BINLOG_NO_CHARS_FOUND
}BinLogStatus;

typedef struct BinLog_t{
        BinLogID ID;
        uint32_t size;
        uint8_t payload[MAX_BINLOG_PAYLOAD_SIZE];
}BinLog;


/******************************************************
* BinLogStatus BinLogEvent(CircBuf* CB, BinLogID ID, uint8_t* payload, uint32_t
length)
*       Description: This function is used to create a BinLog item and initialize it
*               it with an ID and payload given by the function's parameters. The
newly created
*               BinLog item is then added into the CircBuf pointed at by CB.
*       Parameters:
*               - CircBuf* CB: This parameter should be a pointer to an initialized
*                       CircBuf.
*               - BinLogID ID: This parameter is used to indicate the ID of the
*                       BinLog that is to be created.
*               - uint8_t* payload: This parameter is a pointer to the payload
*                       that is to be copied into the newly created BinLog. The value
0
*                       be passed
*               - uint32_t length: This parameter specifies the number of bytes
pointed
*                       at by payload
*       Possible Return Values:
*               - BINLOG_SUCCESS: BinLog was able to be created and added to CB
*               - BINLOG_HEAP_FULL: BinLog unable to be created.
*               - BINLOGBUF_FULL: BinLog created but not added to CB.
******************************************************/
BinLogStatus BinLogEvent(CircBuf* CB, BinLogID ID, uint8_t* payload, uint32_t length);


/******************************************************
 * BinLogStatus BinLogCreate(BinLog** BL , BinLogID ID, uint8_t* payload, uint32_t
length)
 *      Description: Creates BinLog on heap containing data specified by parameters
 *      Parameters:
 *              - BinLog** BL: points at pointer to newly created BinLog
 *              - BinLogID: specifies which type of BinLog is to be created
 *              - uint8_t* payload: data to be added into new BinLog
 *              - uint32_t length: size of payload. number of bytes pointed at by
payload.
 *      Possible Return Values:
 *              - BINLOG_HEAP_FULL: BinLog unable to be created
 *              - BINLOG_SUCCESS: BinLog successfully created
******************************************************/
BinLogStatus BinLogCreate(BinLog** BL , BinLogID ID, uint8_t* payload, uint32_t
length);


/******************************************************
* BinLogStatus BinLogChar(CircBuf* CB, BinLogID ID, uint8_t character)
*       Description: This function is used to log character data that is received
*               via UART. It searches for an existing BinLog with an ID matching the
ID parameter.
*               If one is found the character is added into that BinLog's payload. If
it is not found
*               a new BinLog is made and added to the BinLogBuf.
*       Parameters:
*               - CircBuf* CB: This parameter should be a pointer to an initialized
*                       CircBuf.
*               - BinLogID ID: This parameter is used to indicate the ID of the
*                       BinLog that is to be added to or created.
*               - uint8_t character: This is the data to be added to the payload
*       Possible Return Values:
*               - BINLOG_EVENT_CREATED: A BinLog with a matching ID was not found,
*               and so a new BinLog was created and added to CB successfully.
*               - BINLOG_HEAP_FULL: BinLog unable to be created.
*               - BINLOGBUF_FULL: BinLog created but not added to CB.
*               - BinLog_NO_EVENT_CREATED: A BinLog with a matching ID was found in
CB,
*               character was added to the back of the BinLog's payload.
*****************************************************/
BinLogStatus BinLogChar(CircBuf* CB, BinLogID ID, uint8_t character);


/*****************************************************
* BinLogStatus BinLogSendData(CircBuf* CB, BinLogID ID)
*       Description: This function is used to send out character data that has been
logged
*       in CB. Calls functions from log.h.
*       Parameters:
*               - CircBuf* CB: This parameter should be a pointer to an initialized
*                       CircBuf.
*               - BinLogID ID: This parameter is used to indicate the ID of the
desired
*                       character type. ID = DATA_ALPHA_COUNT prints alphabetic
characters received,
*                       ID = DATA_PUNCTUATION_COUNT prints punctuation characters and
count,
*                       ID = DATA_NUMERIC_COUNT prints all numerical characters
received and count,
*                       ID = DATA_MISC_COUNT prints the count of control characters
received
*       Possible Return Values:
*               - BINLOG_HEAP_FULL: Space unable to be allocated for function.
*               - BINLOG_CHARS_FOUND: Found matching ID in CB and data of that type
is sent out
*                       along with its count (except for MISC, only count sent)
*               - BINLOG_NO_CHARS_FOUND: No BinLog with a matching ID was found.
String printed
*               out to let user know that no data could be found matching that type.
*****************************************************/
BinLogStatus BinLogSendData(CircBuf* CB, BinLogID ID);


/*****************************************************
* CircBufStatus BinLogBufferInit(BinLogBuf* CB, uint32_t size)
*       Description: This function is used to initialize
*               a BinLogBuf. The buffer member is set to point
*               at an  array of type BinLog* that can hold size
*               elements. If the heap is full, function returns a
*               HEAP_FULL error. The end result of a successful
*               call to this function is an empty BinLogBuf.
*       Parameters:
*               - BinLogBuf* CB: This parameter is a pointer to a
*               BinLogBuf. Multiple calls to BufferInit() using
*               the same CB pointer should not be made without calls
*               to BufferDestroy() between them. The function will
*               return a PTR_ERROR if CB is NULL.
*               - uint32_t size: This parameter is the number of elements
*               in the buffer. The value 0 is not allowed and will cause
*               the function to return an INIT_FAILURE error.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer, size is > 0, and
*               heap has enough space to allocate size*sizeof(BinLog*)
*               bytes.
*               - PTR_ERROR: CB is invalid (NULL)
*               - INIT_FAILURE: size is equal to 0
*               - HEAP_FULL: Unable to allocate size*sizeof(BinLog*)
*               bytes from the heap.
******************************************************/
CircBufStatus BinLogBufferInit(CircBuf* LB, uint32_t size);


/******************************************************
* CircBufStatus BinLogBufferAdd(BinLogBuf* CB, BinLogBuf_data_t item)
*       Description: This function is used to add parameter item
*               into an initialized buffer pointed at by CB. This
*               function will by default overwrite the oldest entry
*               if the BinLogBuf is full,
*       Parameters:
*               - BinLogBuf* CB: This parameter should be a valid pointer
*               to a BinLogBuf. After a successful call to this function
*               the BinLogBuf pointed at by CB will now contain a new item.
*               - BinLog* item: This is the value to be added into
*               the BinLogBuf.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to a non-full initialized
*               BinLogBuf, and the item was able to be added.
*               - OVERWRITE: CB is a valid pointer to a full and initialized
*               BinLogBuf and the item has been added by overwriting the previous
*               oldest entry.
*               - PTR_ERROR: CB is a non-valid pointer, or points to a non-initialized
*               BinLogBuf. The item has not been added.
******************************************************/
CircBufStatus BinLogBufferAdd(CircBuf* CB, BinLog* item);


/******************************************************
* CircBufStatus BinLogBufferRemove(BinLogBuf* CB)
*       Description: This function is used to remove the oldest item
*               previously inside of a BinLogBuf pointed at by CB. If successful
*               the item parameter will point at the entry that has just
*               been removed.
*       Parameters:
*               - BinLogBuf* CB: This parameter should be a valid pointer
*               to an initialized, non-empty BinLogBuf.
*               - BinLog** item: This parameter should be a valid
*               pointer which upon successful completion of the function
*               will be pointing at a copy of the removed item. The NULL
*               pointer may be passed in for item if the removed value is
*               of no interest.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to a non-empty initialized
*               BinLogBuf, and the item was able to be removed after being copied
*               into the memory location pointed at by item.
*               - ITEM_REMOVE_FAILURE: CB points at an empty BinLogBuf. Nothing can
*               be removed and the value of *item is the same as before.
*               - PTR_ERROR: CB is an invalid pointer or points to an
*               uninitialized BinLogBuf. The function
*               will return without having done any work.
******************************************************/
CircBufStatus BinLogBufferRemove(CircBuf* CB, BinLog** item);
```

```
/******************************************************
* CircBufStatus BinLogBufferFull(BinLogBuf* CB)
*       Description: This function can be used to check if
*                a BinLogBuf is full.
*       Parameters:
*               BinLogBuf* CB: This parameter should be a valid pointer
*               to an initialized BinLogBuf
*       Possible Return Values:
*               - BUFFER_FULL: CB is a valid pointer to an initialized
*               and full BinLogBuf.
*               - BUFFER_NOT_FULL: CB is a valid pointer to an
*               initialized an non-full BinLogBuf
*               - PTR_ERROR: CB is an invalid pointer, or points
*               to an uninitialized BinLogBuf
******************************************************/
CircBufStatus BinLogBufferFull(CircBuf* CB);


/******************************************************
* CircBufStatus BinLogBufferEmpty(BinLogBuf* CB)
*       Description: This function can be used to check if a
*               BinLogBuf is empty.
*       Parameters:
*               - BinLogBuf* CB: This parameter should be a valid pointer
*               to an initialized BinLogBuf
*       Possible Return Values:
*               - BUFFER_EMPTY: CB is a valid pointer to an initialized
*               and empty BinLogBuf
*               - BUFFER_NOT_EMPTY: CB is a valid pointer to an
*               initialized and non-empty BinLogBuf
                _ PTR_ERROR: CB is an invalid pointer or points at an
*               uninitialized BinLogBuf.
******************************************************/
CircBufStatus BinLogBufferEmpty(CircBuf* CB);


/******************************************************
* CircBufStatus BufferPeek(BinLogBuf* CB, BinLog** item_n, uint32_t n)
*       Description: This function is used to return the nth item
*               inside of a BinLogBuf
*       Parameters:
*               - BinLogBuf* CB: This parameter should be a valid pointer
*               to an initialized BinLogBuf containing at least n items.
*               - BinLog** item_n: This parameter should be a valid
*               pointer that upon successful completion of the function call
*               will point at the nth item in the BinLogBuf.
*               - uint32_t n: This parameter indicates which item should be peeked at.
*               n is one-based, so n = 1 returns the first value in the buffer.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to an initialized BinLogBuf with at
least
*               n items inside of it. item_n will point at a copy of the nth item in
the
*               BinLogBuf.
*               - INVALID_PEEK: CB is a valid pointer to an initialized BinLogBuf
which contains
*               less than n items or n < 1.
*               - PTR_ERROR: CB is invalid or points at an uninitialized BinLogBuf or
item_n
*               is an invalid pointer.
******************************************************/
```

```
CircBufStatus BinLogBufferPeek(CircBuf* CB, BinLog** item_n, uint32_t n);


/******************************************************
* CircBufStatus BinLogBufferClear(CircBuf* CB)
*       Description: This function is used to empty the contents
*               of CB. The dynamic memory is returned to the heap.
*       Parameters:
*               - BinLogBuf* CB: This parameter should be a valid pointer
*               to an initialized BinLogBuf. After a call to this function CB
*               will return to its state immediately after
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to an initialized BinLogBuf with at
least
*               n items inside of it. item_n will point at a copy of the nth item in
the
*               BinLogBuf.
*               - INVALID_PEEK: CB is a valid pointer to an initialized BinLogBuf
which contains
*               less than n items or n < 1.
*               - PTR_ERROR: CB is invalid or points at an uninitialized BinLogBuf or
item_n
*               is an invalid pointer.
******************************************************/
CircBufStatus BinLogBufferClear(CircBuf* CB);


/******************************************************
* CircBufStatus BinLogBufferDestroy(BinLogBuf* CB)
*       Description: This function destroys a BinLogBuf and returns
*               its memory back the heap. To use this buffer again a call
*               to BufferInitialize() must be made.
*       Parameters:
*               - BinLogBuf* CB: This should be a pointer to an initialized
*               BinLogBuf.
*       Possible Return Values:
*               - SUCCESS: The previously valid BinLogBuf pointed at by CB
*               has been destroyed, and its dynamic memory has been returned
*               for later use,
*               - PTR_ERROR: CB is an invalid pointer or points at an uninitialized
*               BinLogBuf. No work is done in this case.
******************************************************/
CircBufStatus BinLogBufferDestroy(CircBuf* CB);


/******************************************************
* CircBufStatus BinLogBufferCount(BinLogBuf* CB)
*       Description: This function returns the number of items in
*               BinLogBuf
*       Parameters:
*               - BinLogBuf* CB: This should be a pointer to an initialized
*               BinLogBuf.
*       Possible Return Values:
*               - <number of items in buffer>: Returns for valid initialized buffer
*               - PTR_ERROR: CB is NULL
******************************************************/
uint32_t BinLogBufferCount(CircBuf* CB);

#endif /* SOURCES_BINARY_LOG_H_ */
/*
 * circbuf.c
 *
 *  Created on: Mar 10, 2017
 *      Author: jonathanwingfield
 */

#include "circbuf.h"

CircBufStatus BufferInit(CircBuf* CB, uint32_t size){

        if(!CB) return PTR_ERROR_BUF;
        if(size == 0) return INIT_FAILURE;
        CB->buffer = (CircBufData_t*)malloc(sizeof(CircBufData_t) * size);
        if(!(CB->buffer)) return HEAP_FULL;
        CB->head = CB->buffer;
        CB->tail = CB->buffer;
        CB->length = size;
        CB->count = 0;
        return SUCCESS_BUF;
}

CircBufStatus BufferAdd(CircBuf* CB, CircBufData_t item){

        if(!CB || !(CB->buffer)) return PTR_ERROR_BUF;

        CircBufData_t* temp_ptr = (CircBufData_t*)CB->head;

        if(CB->count == CB->length) return OVERWRITE;

        if(CB->count > 0)
                CB->head = ((CircBufData_t*)CB->head < (CircBufData_t*)CB->buffer +
CB->length - 1 ?
                                ((CircBufData_t*)CB->head) + 1 : CB->buffer);

        *temp_ptr = item;
        (CB->count)++;
        return SUCCESS_BUF;
}


CircBufStatus BufferRemove(CircBuf* CB, CircBufData_t* item){

        if(!CB || !(CB->buffer) ) return PTR_ERROR_BUF;
        if(CB->count == 0){
                return ITEM_REMOVE_FAILURE;
        }
        if(CB->count == 1){ // return to empty state
                if(item) *item = *((CircBufData_t*)CB->tail);
                CB->tail = CB->buffer;
                CB->head = CB->buffer;
                CB->count = 0;
                return SUCCESS_BUF;
        }
        if(item) *item = *((CircBufData_t*)CB->tail);
        CB->tail = (((CircBufData_t*)CB->tail) < ((CircBufData_t*)CB->buffer) + CB-
>length - 1 ?
                        ((CircBufData_t*)CB->tail) + 1 : CB->buffer);
        (CB->count)--;
        return SUCCESS_BUF;
}

CircBufStatus BufferFull(CircBuf* CB){

        if(!CB) return PTR_ERROR_BUF;
        if(CB->count == CB->length) return BUFFER_FULL;
        else return BUFFER_NOT_FULL;
}

CircBufStatus BufferEmpty(CircBuf* CB){

        if(!CB) return PTR_ERROR_BUF;
        if(CB->count == 0) return BUFFER_EMPTY;
        else return BUFFER_NOT_EMPTY;
}

uint32_t BufferCount(CircBuf* CB){
        if(!CB) return PTR_ERROR_BUF;
        else return CB->count;
}

CircBufStatus BufferPeek(CircBuf* CB, CircBufData_t* item_n, uint32_t n){
// returns nth oldest item
        if(!CB || !item_n || !(CB->buffer)) return PTR_ERROR_BUF;
        if(n > CB->count || n < 1) return INVALID_PEEK;
        *item_n = (((CircBufData_t*)CB->tail) + n - 1 > ((CircBufData_t*)CB->buffer)
+ (CB->length - 1) ?
                        *((CircBufData_t*)(CB->tail - CB->length + n - 1)) : *
((CircBufData_t*)(CB->tail + n - 1)));
        return SUCCESS_BUF;
}

CircBufStatus BufferDestroy(CircBuf* CB){

        if(!CB || !(CB->buffer)) return PTR_ERROR_BUF;
        free(CB->buffer);
        CB->length = 0;
        CB->count = 0;
        CB->head = NULL;
        CB->tail = NULL;
        CB->buffer = NULL;
        return SUCCESS_BUF;
}
/*
 * circbuf.h
 *
 *  Created on: Mar 10, 2017
 *      Author: jonathanwingfield
 */

#ifndef CIRCBUF_H_
#define CIRCBUF_H_

#include <stdint.h>
#include <stdlib.h>

typedef uint8_t CircBufData_t;

typedef struct CircBuf_t{
        void* buffer;
        void* head;
        void* tail;
        uint32_t length;
        uint32_t count;
}CircBuf;

typedef enum CircBufStatus_e{
        SUCCESS_BUF,
        INIT_FAILURE,
        HEAP_FULL,
        INVALID_PEEK,
        BUFFER_DESTROY_FAILURE,
        ITEM_ADD_FAILURE,
        BUFFER_FULL,
        BUFFER_NOT_FULL,
        BUFFER_EMPTY,
        BUFFER_NOT_EMPTY,
        ITEM_REMOVE_FAILURE,
        OVERWRITE,
        PTR_ERROR_BUF
}CircBufStatus;

/*******************************************************
* CircBufStatus BufferInit(CircBuf* CB, uint32_t size)
*       Description: This function is used to initialize
*               a CircBuf. The buffer member is set to point
*               at an  array of type CircBufData_t that can hold size
*               elements. If the heap is full, function returns a
*               HEAP_FULL error. The end result of a successful
*               call to this function is an empty CircBuf.
*       Parameters:
*               - CircBuf* CB: This parameter is a pointer to a
*               CircBuf. Multiple calls to BufferInit() using
*               the same CB pointer should not be made without calls
*               to BufferDestroy() between them. The function will
*               return a PTR_ERROR if CB is NULL.
*               - uint32_t size: This parameter is the number of elements
*               in the buffer. The value 0 is not allowed and will cause
*               the function to return an INIT_FAILURE error.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer, size is > 0, and
*               heap has enough space to allocate size*sizeof(CircBufData_t)
*               bytes.
*               - PTR_ERROR: CB is invalid (NULL)
*               - INIT_FAILURE: size is equal to 0
*               - HEAP_FULL: Unable to allocate size*sizeof(CircBufData_t)
*               bytes from the heap.
*******************************************************/
CircBufStatus BufferInit(CircBuf* CB, uint32_t size);

/*******************************************************
* CircBufStatus BufferAdd(CircBuf* CB, CircBuf_data_t item)
*       Description: This function is used to add parameter item
*               into an initialized buffer pointed at by CB. This
*               function will by default overwrite the oldest entry
*               if the CircBuf is full,
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to a CircBuf. After a successful call to this function
*               the CircBuf pointed at by CB will now contain a new item.
*               - CircBufData_t item: This is the value to be added into
*               the CircBuf.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to a non-full initialized
*               CircBuf, and the item was able to be added.
*               - OVERWRITE: CB is a valid pointer to a full and initalized
*               CircBuf and the item has been added by overwriting the previous
*               oldest entry.
*               - PTR_ERROR: CB is a non-valid pointer, or points to a non-initialized
*               CircBuf. The item has not been added.
*******************************************************/
CircBufStatus BufferAdd(CircBuf* CB, CircBufData_t item);

/*******************************************************
* CircBufStatus BufferRemove(CircBuf* CB)
*       Description: This function is used to remove the oldest item
*               previously inside of a CircBuf pointed at by CB. If successful
*               the item parameter will point at the entry that has just
*               been removed.
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to an initialized, non-empty CircBuf.
*               - CircBufData_t* item: This parameter should be a valid
*               pointer which upon successful completion of the function
*               will be pointing at a copy of the removed item. The NULL
*               pointer may be passed in for item if the removed value is
*               of no interest.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to a non-empty initialized
*               CircBuf, and the item was able to be removed after being copied
*               into the memory location pointed at by item.
*               - ITEM_REMOVE_FAILURE: CB points at an empty CircBuf. Nothing can
*               be removed and the value of *item is the same as before.
*               - PTR_ERROR: CB is an invalid pointer or points to an
*               uninitialized CircBuf. The function
*               will return without having done any work.
*******************************************************/
CircBufStatus BufferRemove(CircBuf* CB, CircBufData_t* item);

/*******************************************************
* CircBufStatus BufferFull(CircBuf* CB)
*       Description: This function can be used to check if
*                a CircBuf is full.
*       Parameters:
*               CircBuf* CB: This parameter should be a valid pointer
*               to an initialized CircBuf
*       Possible Return Values:
*               - BUFFER_FULL: CB is a valid pointer to an initialized
*               and full CircBuf.
*               - BUFFER_NOT_FULL: CB is a valid pointer to an
*               initialized an non-full CircBuf
*               - PTR_ERROR: CB is an invalid pointer, or points
*               to an uninitialized CircBuf
*******************************************************/
CircBufStatus BufferFull(CircBuf* CB);

/*******************************************************
* CircBufStatus BufferEmpty(CircBuf* CB)
*       Description: This function can be used to check if a
*               CircBuf is empty.
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to an initialized CircBuf
*       Possible Return Values:
*               - BUFFER_EMPTY: CB is a valid pointer to an initialized
*               and empty CircBuf
*               - BUFFER_NOT_EMPTY: CB is a valid pointer to an
*               initialized and non-empty CircBuf
                _ PTR_ERROR: CB is an invalid pointer or points at an
*               uninitialized CircBuf.
*******************************************************/
```

```
CircBufStatus BufferEmpty(CircBuf* CB);

/******************************************************
* CircBufStatus BufferPeek(CircBuf* CB, CircBufData_t* item_n, uint32_t n)
*       Description: This function is used to return the nth item
*               inside of a CircBuf
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to an initialized CircBuf containing at least n items.
*               - CircBufData_t* item_n: This parameter should be a valid
*               pointer that upon successful completion of the function call
*               will point at the nth item in the CircBuf.
*               - uint32_t n: This parameter indicates which item should be peeked at.
*               n is one-based, so n = 1 returns the first value in the buffer.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to an initialized CircBuf with at
least
*               n items inside of it. item_n will point at a copy of the nth item in
the
*               CircBuf.
*               - INVALID_PEEK: CB is a valid pointer to an initialized CircBuf which
contains
*               less than n items or n < 1.
*               - PTR_ERROR: CB is invalid or points at an uninitialized CircBuf or
item_n
*               is an invalid pointer.
******************************************************/
CircBufStatus BufferPeek(CircBuf* CB, CircBufData_t* item_n, uint32_t n);

/******************************************************
* CircBufStatus BufferDestroy(CircBuf* CB)
*       Description: This function destroys a CircBuf and returns
*               its memory back the heap. To use this buffer again a call
*               to BufferInitialize() must be made.
*       Parameters:
*               - CircBuf* CB: This should be a pointer to an initialized
*               CircBuf.
*       Possible Return Values:
*               - SUCCESS: The previously valid CircBuf pointed at by CB
*               has been destroyed, and its dynamic memory has been returned
*               for later use,
*               - PTR_ERROR: CB is an invalid pointer or points at an uninitialized
*               CircBuf. No work is done in this case.
******************************************************/
CircBufStatus BufferDestroy(CircBuf* CB);

/******************************************************
* CircBufStatus BufferCount(CircBuf* CB)
*       Description: This function returns the number of items in
*               CircBuf
*       Parameters:
*               - CircBuf* CB: This should be a pointer to an initialized
*               CircBuf.
*       Possible Return Values:
*               - <number of items in buffer>: Returns for valid initialized buffer
*               - PTR_ERROR: CB is NULL
******************************************************/
uint32_t BufferCount(CircBuf* CB);

#endif /* CIRCBUF_H_ */
#include "data.h"
```

```
#define IS_NUM(X) (0x30 <= X && X <= 0x39)
#define NUM_2_ASCII(X) (X + 0x30)
#define ASCII_2_NUM(X) (X - 0x30)

int8_t* my_itoa(int8_t* str, int32_t data, int32_t base)
{
        const int8_t* digits = (int8_t *) ("0123456789abcdefghijklmnopqrstuvwxyz");
        uint8_t i = 0;
        uint32_t u_data;

        if(!str || base > 36 || base < 2) return 0;

        if(data < 0 && base == 10)
        {
                *(str + i++) = '-';
                u_data = ((uint32_t)-data);
                if(data == 0)
                {// overflow. not very elegant solution but faster than do-while loop
                        *(str + i++) = '2';
                        *(str + i++) = '1';
                        *(str + i++) = '4';
                        *(str + i++) = '7';
                        *(str + i++) = '4';
                        *(str + i++) = '8';
                        *(str + i++) = '3';
                        *(str + i++) = '6';
                        *(str + i++) = '4';
                        *(str + i++) = '8';
                        *(str + i++) = '\0';
                        return str;
                }
        }
        else u_data = (uint32_t)data;

        do
        {
                *(str + i++) = digits[u_data % base];
                u_data /= base;
        }while(u_data);

        if(str[0] == '-')
                my_reverse((uint8_t *)(str + 1), i - 1);
        else
                my_reverse((uint8_t *)str, i);

        *(str + i) = '\0';
        return str;
}

int32_t my_atoi(int8_t* str)
{
        uint32_t i;
        int8_t sign = 1;
        int32_t value = 0;
        if(!str) return -1;
        for(i = 0; *(str + i) != '\0'; ++i)
        {
                if(IS_NUM(*(str + i)))
                {
                        value = 10*value + ASCII_2_NUM(*(str + i));
                }
                else if(!value && *(str + i) == '-') sign = -1;
        }
        return sign*value;
}

int8_t big_to_little32(uint32_t* data, uint32_t length)
{
        uint32_t i;
        if(!data) return PTR_ERROR;

        for(i = 0; i < length; i++)
                my_reverse((((uint8_t*)data) + 4*i), 4);

        return SUCCESS;
}

int8_t little_to_big32(uint32_t* data, uint32_t length)
{
        uint32_t i;
        if(!data) return PTR_ERROR;

        for(i = 0; i < length; i++)
                my_reverse((((uint8_t*)data) + 4*i),  4);

        return SUCCESS;
}

void print_memory(uint8_t* start, uint32_t length)
{
        if(!start) return;
#ifndef FRDM
        uint32_t i;
        for(i = 0; i < length; ++i)
                printf("%x ", *(start + i));
#endif



}
#ifndef __DATA_H__
#define __DATA_H__

// standard includes
#include <stdint.h>
// local includes
#include "memory.h"
#include "defines.h"

#ifndef FRDM
#include <stdio.h>
#endif

/*****************************************************************
*       File: data.h
*
*       Dependencies: stdio.h, stdint.h, data.c, memory.h, memory.c
*
*       Description: This file contains several functions useful for
*       manipulating data, including functions to convert integers to
*       ascii strings and vice versa, functions to convert little endian
*       to big endian and vice versa, and a function to print the contents
*       of a section of memory. The functions are documented in more detail
*       below. This code is meant to be portable for multiple architectures,
*       and because of that the print_memory function can be disabled due
*       to its use of the printf() function.
*
****************************************************************/



/****************************************************************
* int8_t* my_itoa(int8_t* str, int32_t data, int32_t base);
*
*       Description: Converts signed 32-bit integer data to null terminated
*       ascii character string representation with radix given by parameter
*       base. Base 2 - Base 36 are supported (who the hell is using Base 36?).
*       Alphabetic and numeric characters are used to give 36 unique characters.
*       Verifies str is valid (!= NULL). It is up to the user to ensure that str
*       points at a buffer large enough to hold the resulting character string.
*       The size of the buffer pointed at by str should generally be at least
*       ceiling(logb(data)/log10(data)), where logb() is the logarithm whose
*       base is base. The return value is the same as str.
*
*       Parameters:
*               - int8_t* str: Pointer to buffer used to store ascii character string.
*                 It is up to the user to ensure the size of the buffer is large
enough.
*                 The buffer must be large enough to hold numerical data in addition
to 1
*                 sign byte (only if base == 10 and data < 0) and a null-terminator
byte.
*               - int32_t data: Value to be converted into string representation.
*               - int32_t base: Radix of resultant character string. Can range from
2-36.
*
*       Return value: int8_t*. The return value points at the same address as str.
*               In case of NULL string being passed in for str or illegal base values
used,
*               return value will be NULL (0).
*
*****************************************************************/
int8_t* my_itoa(int8_t* str, int32_t data, int32_t base);



/****************************************************************
* int32_t my_atoi(int8_t* str);
*
*       Description: Converts null-terminated ascii character string pointed at by
*        str into a signed 32-bit integer representation. Returns 0 if str is invalid
*       (NULL). Ignores all non-numeric characters except for a '+' or a '-' if either
*       is found before any numeric characters. Can handle values in range -2147483648
*       to +2147483647. Values outside of this range will lead to unpredicatble
results.
*
*       Parameters:
*               - int8_t* str: Pointer to null-terminated character string to be
converted
*                 into signed 32-bit integer. Checked for valid address before used.
*
*       Return value: int32_t. This is the resultant integer value obtained from the
*               conversion process. Will be 0 if str is NULL. Takes on values from
-2^31
*               to 2^31 - 1.
*
```

```
*****************************************************************/
int32_t my_atoi(int8_t* str);



/****************************************************************
* int8_t big_to_little32(uint32_t* data, uint32_t length);
*
*       Description: Converts big-endian chunk of memory pointed at by data into
little
*       endian format. Assumes word length of 32-bits. The size of the memory region
is
*       given by the parameter length. Returns 0 in case of SUCCESS_DATA and non-0
value for
*       failure.
*
*
*       Parameters:
*               - uint32_t* data: Pointer to base of big-endian memory region.
*               - uint32_t length: Number of bytes to convert endianness of.
*
*       Return value: int8_t. Return value used to indicate SUCCESS_DATA/failure of
function.
*               0 used to indicate SUCCESS_DATA, non-0 value used to indicate failure.
*
*****************************************************************/
int8_t big_to_little32(uint32_t* data, uint32_t length);



/****************************************************************
* int8_t little_to_big32(uint32_t* data, uint32_t length);
*
*       Description: Converts little-endian chunk of memory pointed at by data into
big
*       endian format. Assumes word length of 32-bits. The size of the memory region
is
*       given by the parameter length. Returns 0 in case of SUCCESS_DATA and non-o
value for
*       failure.
*
*       Parameters:
*               - uint32_t* data: Pointer to base of little-endian memory region.
*               - uint32_t length: Number of bytes to convert endianness of..
*
*       Return value: int8_t. Return value used to indicate SUCCESS_DATA/failure of
function.
*               0 used to indicate SUCCESS_DATA, non-0 value used to indicate failure.
*
*****************************************************************/
int8_t little_to_big32(uint32_t* data, uint32_t length);



/****************************************************************
* void print_memory(uint8_t* start, uint32_t length);
*
*       Description: Prints the contents of memory to stdout using printf(). The
*       base of the memory region is pointed at by start, the number of bytes printed
*       is given by the length parameter. The output is presented in hex format.
*
*       Parameters:
```

```
*               - uint8_t* start: Pointer to start of memory region to be printed out.
*               - uint32_t length: Number of bytes to be printed out.
*
*       Return value: None.
*
*****************************************************************/
void print_memory(uint8_t* start, uint32_t length);



#endif /* __DATA_H__ */
/*
 * defines.h
 *
 *  Created on: Mar 1, 2017
 *      Author: jacob
 */

#ifndef SOURCES_DEFINES_H_
#define SOURCES_DEFINES_H_

#define PTR_ERROR (int8_t) -1
#define SUCCESS (int8_t) 0

#define INTERRUPTS
#define DEFAULT_UARTBUF_SIZE 256
#define DEFAULT_BINLOGBUF_SIZE 128

#define SET_FLAG(FLAG)      FLAG = 1
#define CLEAR_FLAG(FLAG)    FLAG = 0
#define FLAG_IS_SET(FLAG)   FLAG == 1
#define FLAG_IS_CLEAR(FLAG) FLAG == 0

#define MAX_BINLOG_PAYLOAD_SIZE 16

#define B_LOGGER

#define heartbeat_configure()   do{ \
                                                                        SIM_SCGC5 |=
SIM_SCGC5_PORTB_MASK; \
                                                                        PORTB_PCR18 =
PORT_PCR_MUX(1); \
                                                                        GPIOB_PDDR |=
(1<<18); \
                                                                        GPIOB_PDOR &=
~(1<<18); \
                                                                }while(0)

#define heartbeat()             GPIOB_PTOR |= (1<<18)

#endif /* SOURCES_DEFINES_H_ */
/*
 * log.c
 *
 *  Created on: Mar 1, 2017
 *      Author: jonathanwingfield
 */

#include "log.h"

extern CircBuf TXBuf;
```

```
log_error log_data(uint8_t* data, uint32_t length){

#ifdef FRDM
        uart_send_byte_n(data,length);
#endif
#ifndef FRDM
        uint8_t i = 0;
        for(i = 0; i < length; ++i){
                printf("%c", data[i]);
        }
#endif

}

log_error log_string(uint8_t* string){
#ifdef FRDM
        while(*string != '\0')
                uart_send_byte(string++);
#endif
#ifndef FRDM
        printf("%s", string);
#endif
}

log_error log_integer(int32_t num){

        int8_t my_str[11];
        my_itoa(my_str, num, 10);
#ifdef FRDM
        log_string(my_str);
#endif
#ifndef FRDM
        printf("%d", num);
#endif
}

log_error log_flush(void){

        while(BufferEmpty(&TXBuf) == BUFFER_EMPTY);
}
/*
 * log.h
 *
 *  Created on: Mar 1, 2017
 *      Author: jonathanwingfield
 */

#ifndef SOURCES_LOG_H_
#define SOURCES_LOG_H_

#include "uart.h"
#include "data.h"
#include "memory.h"
#include "circbuf.h"
#include "defines.h"

typedef enum log_error_t {
        SUCCESS_LOG,
        NOT_SURE_WHICH_OTHER_CONDITIONS_TO_INCLUDE_YET
} log_error;

log_error log_data(uint8_t* data, uint32_t length);
log_error log_string(uint8_t* string);

log_error log_integer(int32_t num);

log_error log_flush(void);

#endif /* SOURCES_LOG_H_ */
/*
 * Copyright (c) 2015, Freescale Semiconductor, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * o Redistributions of source code must retain the above copyright notice, this list
 *   of conditions and the following disclaimer.
 *
 * o Redistributions in binary form must reproduce the above copyright notice, this
 *   list of conditions and the following disclaimer in the documentation and/or
 *   other materials provided with the distribution.
 *
 * o Neither the name of Freescale Semiconductor, Inc. nor the names of its
 *   contributors may be used to endorse or promote products derived from this
 *   software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
#include <stdint.h>
#include "MKL25Z4.h"
#include "uart.h"
#include "circbuf.h"
#include "uartbuf.h"
#ifdef DEBUG
#include "defines.h"
#include "log.h"
#endif
#ifdef B_LOGGER
#include "binary_log.h"
uint32_t data_flag;
#endif
#include "memory.h"
#include "data.h"

void main(void)
{
        #ifdef FRDM
        #ifdef DEBUG
        uart_configure();
        #ifdef B_LOGGER
        CircBuf BLB;
        BinLogBufferInit(&BLB, DEFAULT_BINLOGBUF_SIZE);
        BinLogEvent(&BLB, LOGGER_INITIALIZED, 0, 0);
        BinLogEvent(&BLB,GPIO_INITIALIZED, 0, 0);
        #ifdef INTERRUPTS
        NVIC_EnableIRQ(UART0_IRQn);
        __enable_irq();
        #endif
        BinLogEvent(&BLB, SYSTEM_INITIALIZED, 0, 0);
        #endif
        #endif
        #endif

        while(1){

                #ifdef B_LOGGER
                uint8_t data = 0, count = 0;
                while(FLAG_IS_CLEAR(data_flag));
                CLEAR_FLAG(data_flag);
                uart_receive_byte(&data);
                count++;

                BinLogEvent(&BLB, DATA_RECEIVED, &data, 1);
                BinLogEvent(&BLB, DATA_ANALYSIS_STARTED, 0 , 0);

                if(IS_ALPHA(data)) BinLogChar(&BLB, DATA_ALPHA_COUNT, data);

                else if(IS_NUMERIC(data)) BinLogChar(&BLB, DATA_NUMERIC_COUNT, data);

                else if(IS_PUNCTUATION(data)) BinLogChar(&BLB,
DATA_PUNCTUATION_COUNT, data);

                else BinLogChar(&BLB, DATA_MISC_COUNT, data);

                BinLogEvent(&BLB, DATA_ANALYSIS_COMPLETED, 0, 0);

                if(count == 16){
                        count = 0;

                        log_string("\n\nAlphabetic Characters\n");
                        BinLogSendData(&BLB, DATA_ALPHA_COUNT);
                        log_string("\n\nPunctuation Characters\n");
                        BinLogSendData(&BLB, DATA_PUNCTUATION_COUNT);
                        log_string("\n\nNumeric Characters\n");
                        BinLogSendData(&BLB, DATA_NUMERIC_COUNT);
                        log_string("\n\nMisc Characters\n");
                        BinLogSendData(&BLB, DATA_MISC_COUNT);
                        BinLogBufferClear(&BLB);
                }

                #endif
                #ifndef B_LOGGER
                uint8_t data[] = {'a','b','c','\0'};
                log_string(data);
                log_flush();
                uint8_t i;
                for(i = 0; i < 4; i++) {
                        *(data + i) = i + 65;
                }
                log_data(data, 4);
            log_flush();
                int32_t ui;
                for(ui = -10; ui < 11; ui++) {
                        log_integer(ui);
                }
            log_flush();
                while(1);
                #endif

        }
}
#include "memory.h"

int8_t my_memmove(uint8_t* src, uint8_t* dst, uint32_t length)
{

        uint32_t i;
        if( !src || !dst ) return PTR_ERROR;

        if(src > dst)
        {
                for(i = 0; i < length; i++)
                        *(dst + i) = *(src + i);
        }
        else if(src < dst)
        {
                for(i = length; i > 0; i--)
                        *(dst + i - 1) =  *(src + i - 1);
        }
        return SUCCESS;

}
int8_t my_memset(uint8_t* src, uint32_t length, uint8_t value)
{
        uint32_t i;
        if( !src ) return PTR_ERROR;

        for(i = 0; i < length; ++i)
                *(src + i) = value;

        return SUCCESS;
}

int8_t my_memzero(uint8_t* src, uint32_t length)
{
        uint32_t i;
        if( !src ) return PTR_ERROR;

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

        return SUCCESS;
}

int8_t my_reverse(uint8_t* src, uint32_t length)
{
        if( !src ) return PTR_ERROR;
  uint32_t i;
        uint8_t temp;

        for(i = 0; i < length >> 1; ++i)
        {
                temp = *(src + i);
                *(src + i)= *(src + length - 1 - i);
                *(src + length -1 - i) = temp;
        }
        return SUCCESS;
```

```
}
#ifndef __MEMORY_H__
#define __MEMORY_H__

// standard includes
#include <stdint.h>
#include "defines.h"

/***********************************************************************************
*       File: memory.h
*
*       Dependencies: stdint.h, memory.c
*
*       Description: Contains prototypes for 4 different functions used to manipulate
*       memory. Implementations of the functions is in memory.c. Detailed descriptions
*       of each function given below.
*
***********************************************************************************/


/***********************************************************************************
* int8_t my_memmove(uint8_t* src, uint8_t* dst, uint32_t length)
*
*       Description: Copies length number of bytes from memory location pointed
*               to by src to memory location pointed to by dst. Works properly for
*               case where pointers overlap, that is |src - dest| < length. Neither
*               src nor dst should be NULL, function returns PTR_ERROR(-1) if either
*               pointer is NULL. Returns SUCCESS(0) after successfully moving all
bytes.
*
*       Parameters:
*               - uint8_t* src: Pointer which holds memory address of source data.
*               - uint8_t* dst: Pointer which holds memory address of destination
data.
*               - uint32_t length: Value which indicates number of bytes to be
transferred
*               from source to destination.
*
*       Return value: int8_t. Function will return PTR_ERROR if either src or dest is
*               a NULL pointer. Returns SUCCESS if neither pointer is NULL and length
bytes
*               are copied.
*
************************************************************************************/
int8_t my_memmove(uint8_t* src, uint8_t* dst, uint32_t length);



/***********************************************************************************
*       int8_t my_memset(uint8_t* src, uint32_t length, uint8_t value)
*
*       Description: Sets chunk of memory equal to value. The size of the chunk of
memory
*               is determined by the value of length. The base memory address is
determined by
*               src. Function return value indicates SUCCESS/FAILURE of operation.
*
*       Parameters:
*               - uint8_t* src: Pointer which holds base memory address of memory to
be set..
*               - uint32_t length: Value which indicates number of bytes to be
written into
```

```
*                 with value.
*               - uint8_t value: Holds the value that each byte of memory will be 
assigned to.
*
*       Return value: int8_t. Function will return PTR_ERROR if src is a NULL 
pointer.
*               Returns SUCCESS after length bytes are written into with value.
*
************************************************************************************/
int8_t my_memset(uint8_t* src, uint32_t length, uint8_t value);



/***********************************************************************************
*int8_t my_memzero(uint8_t* src, uint32_t length)
*
*       Description: Sets chunk of memory equal to 0. The size of the chunk of memory 
*               is determined by the value of length. The base memory address is 
determined by 
*               src. Function return value indicates SUCCESS/FAILURE of operation.
*
*       Parameters:
*               - uint8_t* src: Pointer which holds base memory address of memory to 
be set.
*               - uint32_t length: Value which indicates number of bytes to be zeroed 
out..
*
*       Return value: int8_t. Function will return PTR_ERROR if src is a NULL 
pointer.
*               Returns SUCCESS after length bytes are zeroed out.
*
************************************************************************************/
int8_t my_memzero(uint8_t* src, uint32_t length);



/***********************************************************************************
*       int8_t my_reverse(uint8_t* src, uint32_t length)
*
*       Description: Reverses chunk of memory of size length where base address is 
held
*               in src. Checks to make sure src is a valid pointer ( != NULL). After 
call to
*               function memory is reversed. That is, *(src + length) --> *src,
*               *src --> *(src + length) and so on.
*
*       Parameters:
*               - uint8_t* src: Pointer which holds base memory address of memory to 
be reversed.
*               - uint32_t length: Value which indicates number of bytes to be 
reversed.
*
*       Return value: int8_t. Function will return PTR_ERROR if src is a NULL 
pointer.
*               Returns SUCCESS after length bytes are reversed.
*
************************************************************************************/
int8_t my_reverse(uint8_t* src, uint32_t length);


#endif /* __MEMORY_H__ */
#include "mock_circbuf.h"
```

```
#include <stdint.h>
#ifndef __MOCK_CIRCBUF_H__
#define __MOCK_CIRCBUF_H__


#endif
#include "mock_data.h"
#include <stdint.h>
#ifndef __MOCK_DAtA_H__
#define __MOCK_DAtA_H__


#endif
#include "mock_memory.h"

int8_t __wrap_my_memmove(uint8_t* src, uint8_t* dst, uint32_t length) {
  check_expected_ptr(src);
  check_expected_ptr(dst);

  return (int8_t) mock();
}

int8_t __wrap_my_memset(uint8_t* src, uint32_t length, uint8_t value) {
  check_expected_ptr(src);

  return (int8_t) mock();
}

int8_t __wrap_my_memzero(uint8_t* src, uint32_t length) {
  check_expected_ptr(src);

  return (int8_t) mock();
}

int8_t __wrap_my_reverse(uint8_t* src, uint32_t length) {
  int i;
  for(i = 0; i < length; i++) {
    src[i] = (uint8_t)mock();
  }
  return (int8_t) mock();
}
#ifndef __MOCK_MEMORY_H
#define __MOCK_MEMORY_H

#include <stdbool.h>
#include <stdarg.h>
#include <setjmp.h>
#include <stdlib.h>
#include <cmocka.h>
#include <stdint.h>
#include "defines.h"

int8_t __wrap_my_memmove(uint8_t* src, uint8_t* dst, uint32_t length);
int8_t __wrap_my_memset(uint8_t* src, uint32_t length, uint8_t value);
int8_t __wrap_my_memzero(uint8_t* src, uint32_t length);
int8_t __wrap_my_reverse(uint8_t* src, uint32_t length);


#endif
#include <stdbool.h>
#include <stdarg.h>
#include <setjmp.h>
```

```
#include <stdlib.h>
#include <cmocka.h>

#include <stdio.h>

#include "circbuf.h"
#include "data.h"
#include "memory.h"
#include "defines.h"

#include "mock_memory.h"


#include <stdint.h>

#ifdef MEMORY
// Will test that memmove returns properly when passed a null pointer input
static void test_invalid_pointer_memmove(void **state) {
  uint8_t * src = NULL;
  uint8_t dst[] = {0xAA, 0xBB, 0xCC, 0xDD, 0xEE};
  uint32_t length = 3;

  // Test with NULL src
  int ret1 = (int) my_memmove(src, dst, length);

  src = dst;
  uint8_t * ndst = NULL;

  // Test with NULL ndst
  int ret2 = (int) my_memmove(src, ndst, length);

  assert_int_equal(ret1, PTR_ERROR);
  assert_int_equal(ret2, PTR_ERROR);
}

// Tests that memmove works correctly on memory regions with no overlap
static void test_overlap_memmove(void **state) {
  uint8_t src[] = {0xAA, 0xBB, 0xCC, 0xDD};
  uint8_t dst[] = {0xEE,0xFF,0x00,0x11};
  uint32_t length = 4;

  // test mocve with no overlap
  int ret = (int) my_memmove(src, dst, length);
  uint32_t i;
  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < length; i ++) {
    assert_true(dst[i] == src[i]);
  }
}

// Tests that memmove works correctly when the source region is a subset of the
destination region
static void test_SRC_DST_overlap_memmove(void **state) {
  uint8_t dst[] = {0xAA, 0xBB, 0xCC, 0xDD};
  uint8_t * src = dst + 1;
  src[3] = 0x99;
  uint32_t length = 4;
  // Need temporary holder to compare destination to as src will be overwritten
  uint8_t hld[] = {0xBB, 0xCC, 0xDD, 0x99};


  // Test when src is in dst
  int ret = (int) my_memmove(src, dst, length);
  uint32_t i;
  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < length; i ++) {
    assert_true(dst[i] == hld[i]);

  }
}

// Tests that memmove works correctly when the destination region is  a subset of the
source region
static void test_DST_SRC_overlap_memmove(void **state) {
  uint8_t src[] = {0xAA, 0xBB, 0xCC, 0xDD};
  uint8_t * dst = src + 1;
  dst[3] = 0xEE;
  uint32_t length = 4;
  // Need temporary holder to compare destination to as src will be overwritten
  uint8_t hld[] = {0xAA, 0xBB, 0xCC, 0xDD};

  // test when dst is in src
  int ret = (int) my_memmove(src, dst, length);
  uint32_t i;
  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < length; i ++) {
    assert_true(dst[i] == hld[i]);
  }
}

// Tests that memset returns correctly with null pointer input
static void test_invalid_pointer_memset(void **state) {
  uint8_t * src = NULL;
  uint32_t length = 4;
  uint8_t value = 17;

  int ret = (int) my_memset(src, length, value);

  assert_int_equal(ret, PTR_ERROR);
}

// Tests that memset works correctly for entire array and susbset of array
static void test_check_set_memset(void **state) {
  uint8_t * src;
  uint8_t arr[] = {9,8,7,6,5,0};
  src = arr;
  uint32_t length = 6;
  uint8_t value = 4;
  int i = 0;

  int ret1 = (int) my_memset(src, length, value);
  for(i = 0; i < length; i++) {
    assert_true(*(src+i) == value);
  }

  value = 0;
  length = 2;
  src = src + 3;

  int ret2 = (int) my_memset(src, length, value);
  for(i = 0; i < length; i++) {
    assert_true(*(src+i) == value);
  }
  assert_int_equal(ret1, SUCCESS);
  assert_int_equal(ret2, SUCCESS);
}

// Tests that memzero returns correctly with null pointer input
static void test_invalid_pointer_memzero(void **state) {
  uint8_t * src = NULL;
  uint32_t length = 5;

  int ret = (int) my_memzero(src, length);

  assert_int_equal(ret, PTR_ERROR);
}

// Tests that memzero works correctly for  entire array and subset of array
static void test_check_set_memzero(void **state) {
  uint8_t * src = NULL;
  uint8_t arr[] = {1,2,3,4,5,6,7};
  src = arr;
  uint32_t length = 5;
  int i;

  int ret1 = (int) my_memzero(src, length);
  for(i = 0; i < length; i++) {
    assert_true(*(src+i) == 0);
  }

  length = 2;
  uint8_t ard[] = {1,2,3,4,5,6,7};
  src = ard;
  src = src + 3;

  int ret2 = (int) my_memzero(src, length);
  for(i = 0; i < length; i++) {
    assert_true(*(src+i) == 0);
  }

  assert_int_equal(ret1, SUCCESS);
  assert_int_equal(ret2, SUCCESS);
}

// Test that reverse returns correctly with null pointer input
static void test_invalid_pointer_reverse(void **state) {
  uint8_t * src = NULL;
  uint32_t length = 5;

  int ret = (int) my_reverse(src, length);

  assert_int_equal(ret, PTR_ERROR);
}

// Tests that reverse functions correctly for odd numbered arrays
static void test_odd_reverse(void **state) {
  uint8_t src[] = {1,2,3,4,5};
  uint32_t length = 5;

  uint8_t comp[] = {5,4,3,2,1};
  int i;

  int ret = (int) my_reverse(src, length);

  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < length; i++) {
    assert_true(src[i] == comp[i]);
  }
}

// Test that reverse functions correctly for even numbered arrays
static void test_even_reverse(void **state) {
  uint8_t src[] = {1,2,3,4};
  uint32_t length = 4;

  uint8_t comp[] = {4,3,2,1};
  int i;

  int ret = (int) my_reverse(src, length);

  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < length; i++) {
    assert_true(src[i] == comp[i]);
  }
}

// Tests that reverse correctly revereses input arrays
static void test_check_characters_reverse(void **state) {
  uint8_t src[256];
  uint32_t length = 256;
  int i;
  for(i = 0; i < (int)length; i++) {
    src[i] = i;
  }

  uint8_t comp[256];
  for(i = 0; i < (int)length; i++) {
    comp[i] = 255 - i;
  }

  int ret = (int) my_reverse(src, length);

  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < length; i++) {
    assert_true(src[i] == comp[i]);
  }
}
#endif
/**********************************************/
#ifdef DATA
// Test that big to little returns correctly with null pointer input
static void test_invalid_pointer_BtL(void **state) {
  uint8_t * src = NULL;
  uint32_t length = 1;

  int ret = (int) big_to_little32((uint32_t *)src, length);
  assert_int_equal(ret, PTR_ERROR);
}

// Tests that big to little correctly converts inputs
static void test_valid_converstion_BtL(void **state) {
  uint8_t * src = NULL;
  uint8_t arr[] = {0xAA,0xBB,0xCC,0xDD,0xEE,0xFF,0x00,0x11};
  uint8_t ar2[] = {0xDD,0xCC,0xBB,0xAA,0x11,0x00,0xFF,0xEE};
  src = arr;
  uint32_t length = 2;
  int i;
  int j = 0;
  for(i = 1; i <= length; i++) {
    while(j < i*4) {
      will_return(__wrap_my_reverse, ar2[j]);
      j++;
    }
    will_return(__wrap_my_reverse, SUCCESS);
  }

  int ret = big_to_little32((uint32_t *)src, length);

  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < 4 * length; i++) {
    assert_true(src[i] == ar2[i]);
  }
}

// Test that little to big returns correctly with null pointer input
static void test_invalid_pointer_LtB(void **state) {
  uint8_t * src = NULL;
  uint32_t length = 1;

  int ret = (int) little_to_big32((uint32_t *)src, length);
  assert_int_equal(ret, PTR_ERROR);
}

// Tests that little to big correctly converts inputs
static void test_valid_conversion_LtB(void **state) {
  uint8_t * src = NULL;
  uint8_t arr[] = {0xAA,0xBB,0xCC,0xDD,0xEE,0xFF,0x00,0x11};
  uint8_t ar2[] = {0xDD,0xCC,0xBB,0xAA,0x11,0x00,0xFF,0xEE};
  src = arr;
  uint32_t length = 2;
  int i;
  int j = 0;
  for(i = 1; i <= length; i++) {
    while(j < i*4) {
      will_return(__wrap_my_reverse, ar2[j]);
      j++;
    }
    will_return(__wrap_my_reverse, SUCCESS);
  }

  int ret = little_to_big32((uint32_t *)src, length);

  assert_int_equal(ret, SUCCESS);
  for(i = 0; i < 4 * length; i++) {
    assert_true(src[i] == ar2[i]);
  }
}
#endif
/************************************************/
#ifdef CIRCBUF
// Test that bufferinit functions correctly for various inputs
static void test_allocate_free(void **state) {
  CircBuf buf;
  uint32_t size = 0;

  int ret = (int) BufferInit(NULL, size);
  assert_int_equal(ret, PTR_ERROR_BUF);

  ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, INIT_FAILURE);

  size = 256;
  ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = (int) BufferDestroy(NULL);
  assert_int_equal(ret, PTR_ERROR_BUF);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Test that all functions in circbuf return correctly with null pointer input
static void test_invalid_pointer_circbuf(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  int ret;
  CircBufData_t data = 28;
  CircBufData_t * ptr = NULL;
  uint32_t n = 2;

  ret = (int) BufferAdd(NULL, data);
  assert_int_equal(ret, PTR_ERROR_BUF);

  ret = (int) BufferRemove(NULL, ptr);
  assert_int_equal(ret, PTR_ERROR_BUF);

  ret = (int) BufferFull(NULL);
  assert_int_equal(ret, PTR_ERROR_BUF);

  ret = (int) BufferEmpty(NULL);
  assert_int_equal(ret, PTR_ERROR_BUF);

  ret = (int) BufferPeek(NULL, ptr, n);
  assert_int_equal(ret, PTR_ERROR_BUF);

  ret = (int) BufferDestroy(NULL);
  assert_int_equal(ret, PTR_ERROR_BUF);
/*******************************************/
  ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = (int) BufferAdd(&buf, data);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = (int) BufferRemove(&buf, ptr);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = (int) BufferFull(&buf);
  assert_int_equal(ret, BUFFER_NOT_FULL);

  ret = (int) BufferEmpty(&buf);
  assert_int_equal(ret, BUFFER_EMPTY);

  ret = (int) BufferPeek(&buf, ptr, n);
  assert_int_equal(ret, PTR_ERROR_BUF);
/*******************************************/
  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}
```

```
// Tests that buffer is initialized
static void test_non_init_buff(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  assert_true(buf.buffer);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Tests that removed items are identical to the added items
static void test_add_remove(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  CircBufData_t data = 26;
  CircBufData_t ptr;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  BufferAdd(&buf, data);
  BufferRemove(&buf, &ptr);

  assert_true(ptr == data);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Tests that bufferfull returns correctly when buffer is full
static void test_buffer_full(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = (int) BufferFull(&buf);
  assert_int_equal(ret, BUFFER_NOT_FULL);

  int i;
  for(i = 0; i < size; i++) {
    BufferAdd(&buf, (CircBufData_t)i);
  }

  ret = (int) BufferFull(&buf);
  assert_int_equal(ret, BUFFER_FULL);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Tests that bufferempty returns correctly based on buffer state
static void test_buffer_empty(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = BufferEmpty(&buf);
  assert_int_equal(ret, BUFFER_EMPTY);
  int i;
  for(i = 0; i < size; i++) {
    BufferAdd(&buf, (CircBufData_t)i);
  }

  ret = (int) BufferEmpty(&buf);
  assert_int_equal(ret, BUFFER_NOT_EMPTY);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Test that buffer adds across wrap
static void test_wrap_add(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  CircBufData_t * ptr = NULL;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  int i;
  // Fill buffer
  for(i = 0; i < size; i++) {
    BufferAdd(&buf, (CircBufData_t)i);
  }
  // Remove one item
  BufferRemove(&buf, ptr);

  // Add at wrap point
  ret = (int) BufferAdd(&buf, (CircBufData_t)i);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Tests that buffer removes across wrap
static void test_wrap_remove(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  CircBufData_t * ptr = NULL;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  int i;
  // Fill buffer
  for(i = 0; i < size; i++) {
    BufferAdd(&buf, (CircBufData_t)i);
  }
  // remove one item
  BufferRemove(&buf, ptr);

  // Add at wrap point
  BufferAdd(&buf, (CircBufData_t)i);
  // Remove all but one
  for(i = 0; i < size - 1; i++) {
    BufferRemove(&buf, ptr);
  }

  // Remove at wrap point
  ret = (int) BufferRemove(&buf, ptr);
  assert_int_equal(ret, SUCCESS_BUF);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Test that buffer doesn't overwrite
static void test_over_fill(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  int i;
  // Fill buffer
  for(i = 0; i < size; i++) {
    BufferAdd(&buf, (CircBufData_t)i);
  }

  // Attempt to add another item
  ret = (int) BufferAdd(&buf, (CircBufData_t) i);
  assert_int_equal(ret, OVERWRITE);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}

// Tests that buffer deosn't remove from an empty buffer
static void test_over_empty(void **state) {
  CircBuf buf;
  uint32_t size = 256;
  CircBufData_t * ptr = NULL;
  int ret = (int) BufferInit(&buf, size);
  assert_int_equal(ret, SUCCESS_BUF);

  // Attempt to remove non-existent item
  ret = (int) BufferRemove(&buf, ptr);
  assert_int_equal(ret, ITEM_REMOVE_FAILURE);

  ret = (int) BufferDestroy(&buf);
  assert_int_equal(ret, SUCCESS_BUF);
}
#endif

int main() {
  const struct CMUnitTest tests[] = {
#ifdef MEMORY
    cmocka_unit_test(test_invalid_pointer_memmove),
    cmocka_unit_test(test_overlap_memmove),
    cmocka_unit_test(test_SRC_DST_overlap_memmove),
    cmocka_unit_test(test_DST_SRC_overlap_memmove),
    cmocka_unit_test(test_invalid_pointer_memset),
    cmocka_unit_test(test_check_set_memset),
    cmocka_unit_test(test_invalid_pointer_memzero),
    cmocka_unit_test(test_check_set_memzero),
    cmocka_unit_test(test_invalid_pointer_reverse),
    cmocka_unit_test(test_odd_reverse),
    cmocka_unit_test(test_even_reverse),
    cmocka_unit_test(test_check_characters_reverse),
#endif
#ifdef DATA
    cmocka_unit_test(test_invalid_pointer_BtL),
    cmocka_unit_test(test_valid_converstion_BtL),
    cmocka_unit_test(test_invalid_pointer_LtB),
    cmocka_unit_test(test_valid_conversion_LtB),
#endif
#ifdef CIRCBUF
    cmocka_unit_test(test_allocate_free),
    cmocka_unit_test(test_invalid_pointer_circbuf),
    cmocka_unit_test(test_non_init_buff),
    cmocka_unit_test(test_add_remove),
    cmocka_unit_test(test_buffer_full),
    cmocka_unit_test(test_buffer_empty),
    cmocka_unit_test(test_wrap_add),
    cmocka_unit_test(test_wrap_remove),
    cmocka_unit_test(test_over_fill),
    cmocka_unit_test(test_over_empty),
#endif
  };

  return cmocka_run_group_tests(tests, NULL, NULL);
}
#include "uartbuf.h"

CircBufStatus UARTBufferInit(CircBuf* CB, uint32_t size){

        if(!CB) return PTR_ERROR_BUF;
        if(size == 0) return INIT_FAILURE;
        CB->buffer = (uint8_t*)malloc(sizeof(uint8_t) * size);
        if(!(CB->buffer)) return HEAP_FULL;
        CB->head = CB->buffer;
        CB->tail = CB->buffer;
        CB->length = size;
        CB->count = 0;
        return SUCCESS_BUF;
}

CircBufStatus UARTBufferAdd(CircBuf* CB, uint8_t item){

        uint8_t* temp_ptr;

        if(!CB || !(CB->buffer)) return PTR_ERROR_BUF;
        if(CB->count == CB->length) return OVERWRITE;

        if(CB->count > 0)
                CB->head = ((uint8_t*)CB->head < (uint8_t*)CB->buffer + CB->length -
1 ? ((uint8_t*)CB->head) + 1 : CB->buffer);

        temp_ptr = (uint8_t*)CB->head;
        *temp_ptr = item;
        (CB->count)++;
        return SUCCESS_BUF;
}


CircBufStatus UARTBufferRemove(CircBuf* CB, uint8_t* item){

        if(!CB || !(CB->buffer) ) return PTR_ERROR_BUF;
        if(CB->count == 0){
                return ITEM_REMOVE_FAILURE;
        }
        if(CB->count == 1){ // return to empty state
                if(item) *item = *((uint8_t*)CB->tail);
                CB->tail = CB->buffer;
                CB->head = CB->buffer;
                CB->count = 0;
                return SUCCESS_BUF;
        }

        if(item) *item = *((uint8_t*)CB->tail);
        CB->tail = (((uint8_t*)CB->tail) < ((uint8_t*)CB->buffer) + CB->length - 1 ?
((uint8_t*)CB->tail) + 1 : CB->buffer);
        (CB->count)--;
        return SUCCESS_BUF;
}

CircBufStatus UARTBufferFull(CircBuf* CB){

        if(!CB) return PTR_ERROR_BUF;
        if(CB->count == CB->length) return BUFFER_FULL;
        else return BUFFER_NOT_FULL;
}

CircBufStatus UARTBufferEmpty(CircBuf* CB){

        if(!CB) return PTR_ERROR_BUF;
        if(CB->count == 0) return BUFFER_EMPTY;
        else return BUFFER_NOT_EMPTY;
}

uint32_t UARTBufferCount(CircBuf* CB){
        if(!CB) return PTR_ERROR_BUF;
        else return CB->count;
}

CircBufStatus UARTBufferPeek(CircBuf* CB, uint8_t* item_n, uint32_t n){
// returns nth oldest item
        if(!CB || !item_n || !(CB->buffer)) return PTR_ERROR_BUF;
        if(n > CB->count || n < 1) return INVALID_PEEK;
        *item_n = (((uint8_t*)CB->tail) + n - 1 > ((uint8_t*)CB->buffer) + (CB-
>length - 1) ?
                        *((uint8_t*)(CB->tail - CB->length + n - 1)) : *
(((uint8_t*)CB->tail) + n - 1));
        return SUCCESS_BUF;
}

CircBufStatus UARTBufferDestroy(CircBuf* CB){

        if(!CB || !(CB->buffer)) return PTR_ERROR_BUF;
        free(CB->buffer);
        CB->length = 0;
        CB->count = 0;
        CB->head = NULL;
        CB->tail = NULL;
        CB->buffer = NULL;
        return SUCCESS_BUF;
}
#ifndef __UartBuf_H__
#define __UartBuf_H__

#include <stdint.h>
#include <stdlib.h>
#include "circbuf.h"

/*******************************************************
* CircBufStatus UARTBufferInit(CircBuf* CB, uint32_t size)
*       Description: This function is used to initialize
*               a CircBuf. The UARTBuffer member is set to point
*               at an  array of type uint8_t that can hold size
*               elements. If the heap is full, function returns a
*               HEAP_FULL error. The end result of a successful
*               call to this function is an empty CircBuf.
*       Parameters:
*               - CircBuf* CB: This parameter is a pointer to a
*               CircBuf. Multiple calls to UARTBufferInit() using
*               the same CB pointer should not be made without calls
*               to UARTBufferDestroy() between them. The function will
*               return a PTR_ERROR if CB is NULL.
*               - uint32_t size: This parameter is the number of elements
*               in the UARTBuffer. The value 0 is not allowed and will cause
*               the function to return an INIT_FAILURE error.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer, size is > 0, and
*               heap has enough space to allocate size*sizeof(uint8_t)
*               bytes.
*               - PTR_ERROR: CB is invalid (NULL)
*               - INIT_FAILURE: size is equal to 0
*               - HEAP_FULL: Unable to allocate size*sizeof(uint8_t)
*               bytes from the heap.
*****************************************************/
CircBufStatus UARTBufferInit(CircBuf* CB, uint32_t size);

/*****************************************************
* CircBufStatus UARTBufferAdd(CircBuf* CB, CircBuf_data_t item)
*       Description: This function is used to add parameter item
*               into an initialized UARTBuffer pointed at by CB. This
*               function will by default overwrite the oldest entry
*               if the CircBuf is full,
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to a CircBuf. After a successful call to this function
*               the CircBuf pointed at by CB will now contain a new item.
*               - uint8_t item: This is the value to be added into
*               the CircBuf.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to a non-full initialized
*               CircBuf, and the item was able to be added.
*               - OVERWRITE: CB is a valid pointer to a full and initalized
*               CircBuf and the item has been added by overwriting the previous
*               oldest entry.
*               - PTR_ERROR: CB is a non-valid pointer, or points to a non-initialized
*               CircBuf. The item has not been added.
*****************************************************/
CircBufStatus UARTBufferAdd(CircBuf* CB, uint8_t item);

/*****************************************************
* CircBufStatus UARTBufferRemove(CircBuf* CB)
*       Description: This function is used to remove the oldest item
*               previously inside of a CircBuf pointed at by CB. If successful
*               the item parameter will point at the entry that has just
*               been removed.
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to an initialized, non-empty CircBuf.
*               - uint8_t* item: This parameter should be a valid
*               pointer which upon successful completion of the function
*               will be pointing at a copy of the removed item. The NULL
*               pointer may be passed in for item if the removed value is
*               of no interest.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to a non-empty initialized
*               CircBuf, and the item was able to be removed after being copied
*               into the memory location pointed at by item.
*               - ITEM_REMOVE_FAILURE: CB points at an empty CircBuf. Nothing can
*               be removed and the value of *item is the same as before.
*               - PTR_ERROR: CB is an invalid pointer or points to an
*               uninitialized CircBuf. The function
*               will return without having done any work.
*****************************************************/
CircBufStatus UARTBufferRemove(CircBuf* CB, uint8_t* item);

/*****************************************************
* CircBufStatus UARTBufferFull(CircBuf* CB)
*       Description: This function can be used to check if
*                a CircBuf is full.
*       Parameters:
*               CircBuf* CB: This parameter should be a valid pointer
*               to an initialized CircBuf
*       Possible Return Values:
*               - UARTBuffer_FULL: CB is a valid pointer to an initialized
*               and full CircBuf.
*               - UARTBuffer_NOT_FULL: CB is a valid pointer to an
*               initialized an non-full CircBuf
*               - PTR_ERROR: CB is an invalid pointer, or points
*               to an uninitialized CircBuf
*****************************************************/
CircBufStatus UARTBufferFull(CircBuf* CB);

/*****************************************************
* CircBufStatus UARTBufferEmpty(CircBuf* CB)
*       Description: This function can be used to check if a
*               CircBuf is empty.
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to an initialized CircBuf
*       Possible Return Values:
*               - UARTBuffer_EMPTY: CB is a valid pointer to an initialized
*               and empty CircBuf
*               - UARTBuffer_NOT_EMPTY: CB is a valid pointer to an
*               initialized and non-empty CircBuf
                _ PTR_ERROR: CB is an invalid pointer or points at an
*               uninitialized CircBuf.
*****************************************************/
CircBufStatus UARTBufferEmpty(CircBuf* CB);

/*****************************************************
* CircBufStatus UARTBufferPeek(CircBuf* CB, uint8_t* item_n, uint32_t n)
*       Description: This function is used to return the nth item
*               inside of a CircBuf
*       Parameters:
*               - CircBuf* CB: This parameter should be a valid pointer
*               to an initialized CircBuf containing at least n items.
*               - uint8_t* item_n: This parameter should be a valid
*               pointer that upon successful completion of the function call
*               will point at the nth item in the CircBuf.
*               - uint32_t n: This parameter indicates which item should be peeked at.
*               n is one-based, so n = 1 returns the first value in the UARTBuffer.
*       Possible Return Values:
*               - SUCCESS: CB is a valid pointer to an initialized CircBuf with at
least
```

```
*               n items inside of it. item_n will point at a copy of the nth item in
the
*               CircBuf.
*               - INVALID_PEEK: CB is a valid pointer to an initialized CircBuf which
contains
*               less than n items or n < 1.
*               - PTR_ERROR: CB is invalid or points at an uninitialized CircBuf or
item_n
*               is an invalid pointer.
*****************************************************/
CircBufStatus UARTBufferPeek(CircBuf* CB, uint8_t* item_n, uint32_t n);

/*****************************************************
* CircBufStatus UARTBufferDestroy(CircBuf* CB)
*       Description: This function destroys a CircBuf and returns
*               its memory back the heap. To use this UARTBuffer again a call
*               to UARTBufferInitialize() must be made.
*       Parameters:
*               - CircBuf* CB: This should be a pointer to an initialized
*               CircBuf.
*       Possible Return Values:
*               - SUCCESS: The previously valid CircBuf pointed at by CB
*               has been destroyed, and its dynamic memory has been returned
*               for later use,
*               - PTR_ERROR: CB is an invalid pointer or points at an uninitialized
*               CircBuf. No work is done in this case.
*****************************************************/
CircBufStatus UARTBufferDestroy(CircBuf* CB);

/*****************************************************
*        UARTBufferCount(CircBuf* CB)
*       Description: This function returns the number of items in
*               CircBuf
*       Parameters:
*               - CircBuf* CB: This should be a pointer to an initialized
*               CircBuf.
*       Possible Return Values:
*               - <number of items in UARTBuffer>: Returns for valid initialized
UARTBuffer
*               - PTR_ERROR: CB is NULL
*****************************************************/
uint32_t UARTBufferCount(CircBuf* CB);

#endif /* __UartBuf_H__ */
/*
 * uart.c
 *
 *  Created on: Feb 28, 2017
 *      Author: jacob
 */

#include "uart.h"

CircBuf TXBuf, RXBuf;

UART_RETURN uart_configure(void) {

        SIM_SOPT2 |= SIM_SOPT2_PLLFLLSEL(0);
        SIM_SOPT2 |= SIM_SOPT2_UART0SRC(1);
        SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;

        SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;
        PORTA_PCR1 = PORT_PCR_MUX(2);
        PORTA_PCR2 = PORT_PCR_MUX(2);

        UART0_C2  = 0x00;
        UART0_C1  = 0x00;
        UART0_C3  = 0x00;
        UART0_C4  = 0x00;
        UART0_BDH = 0x00;
        UART0_BDL = 0x16;

#ifdef INTERRUPTS
        if(UARTBufferInit(&TXBuf, DEFAULT_UARTBUF_SIZE) != SUCCESS_BUF)
                return UART_INIT_FAILURE;
        if(UARTBufferInit(&RXBuf, DEFAULT_UARTBUF_SIZE) != SUCCESS_BUF)
                return UART_INIT_FAILURE;

        UART0_C2 = UART0_C2_RIE_MASK;

#endif

        UART0_C2 |= UART_C2_RE_MASK | UART0_C2_TE_MASK;
        return UART_INIT_SUCCESS;
}


UART_RETURN uart_send_byte(uint8_t* data) {

#ifdef INTERRUPTS
        while(UARTBufferFull(&TXBuf) == BUFFER_FULL);
        if(UARTBufferAdd(&TXBuf, *data) != SUCCESS_BUF) return
UART_SEND_BUFADD_FAILURE;
        UART0_C2 |= UART0_C2_TIE_MASK;
        return UART_SEND_BUFADD_SUCCESS;
#endif

#ifndef INTERRUPTS
        if(data == NULL) return UART_SEND_FAILURE;
        while(!(UART0_S1 & UART_S1_TDRE_MASK)); // wait for transmit buffer to empty
        UART0_D = *data;
        return UART_SEND_SUCCESS;
#endif

}

UART_RETURN uart_send_byte_n(uint8_t* data, uint32_t length) {

        uint32_t i;
#ifdef INTERRUPTS
        for(i = 0; i < length; ++i){
                if(UARTBufferAdd(&TXBuf, *(data+i)) != SUCCESS_BUF)
                        return UART_SEND_BUFADD_FAILURE;
        }
        if(length) UART0_C2 |= UART0_C2_TIE_MASK;
        return UART_SEND_BUFADD_SUCCESS;
#endif

#ifndef INTERRUPTS
        for(i = 0; i < length; ++i){
                if(uart_send_byte(data + i) == UART_SEND_FAILURE);
                return UART_SEND_FAILURE;
        }
        return UART_SEND_SUCCESS;
```

```
#endif

}

UART_RETURN uart_receive_byte(uint8_t* buffer) {

        if(buffer == NULL) return UART_RECEIVE_FAILURE;

#ifdef INTERRUPTS
        while(UARTBufferEmpty(&RXBuf) == BUFFER_EMPTY); // wait for receive buffer to
be full
        UARTBufferRemove(&RXBuf, buffer);
#endif
#ifndef INTERRUPTS
        while(!(UART0_S1 & UART_S1_RDRF_MASK)); // wait for receive buffer to be full
        *buffer = UART0_D;
#endif
        return UART_RECEIVE_SUCCESS;
}

UART_RETURN uart_receive_byte_n(uint8_t* buffer, uint32_t length) {

        uint32_t i;
        if(buffer == NULL) return UART_RECEIVE_FAILURE;

        for(i = 0; i < length; ++i){
                uart_receive_byte(buffer + i);
        }
        return UART_RECEIVE_SUCCESS;
}

extern void UART0_IRQHandler(void){

        uint8_t data = 0;

        if(UART0_S1 & UART_S1_TDRE_MASK){
                if(UARTBufferCount(&TXBuf) >= 1){
                        UARTBufferRemove(&TXBuf, &data);
                        UART0_D = data;
                }
                else UART0_C2 &= ~UART0_C2_TIE_MASK;
        }
        if(UART0_S1 & UART_S1_RDRF_MASK){
                if(UARTBufferFull(&RXBuf) == BUFFER_NOT_FULL){
                        data = UART0_D;
                        UARTBufferAdd(&RXBuf, data);
                        #ifdef B_LOGGER
                        SET_FLAG(data_flag);
                        #endif
                }
        }
}
/*
 * uart.h
 *
 *  Created on: Feb 28, 2017
 *      Author: jacob
 */
#ifndef SOURCES_UART_H_
#define SOURCES_UART_H_

#include "MKL25Z4.h"
```

```
#include "defines.h"
#include "uartbuf.h"

extern uint32_t data_flag;

typedef enum {
        UART_INIT_SUCCESS,
        UART_INIT_FAILURE,
        UART_SEND_SUCCESS,
        UART_SEND_FAILURE,
        UART_SEND_BUFADD_SUCCESS,
        UART_SEND_BUFADD_FAILURE,
        UART_RECEIVE_SUCCESS,
        UART_RECEIVE_FAILURE
}UART_RETURN;


/***************************************************************
 * UART_RETURN uart_configure(void)
 *      Description: Configures UART0 for 57600 BR, 8-N-1 UART comm.
 *      INTERRUPT switch being set initializes UART TX and RX buffers
 *      and enables RX interrupts.
 *      Parameters:
 *              - none
 *      Possible Return Values:
 *              - UART_INIT_SUCCESS: UART succesfully initialized
 *              - UART_INIT_FAILURE: when INTERRUTPS switch set,
 *                      occurs if Buffer for UART are unable to be created
 **************************************************************/
UART_RETURN uart_configure(void);


/***************************************************************
 * UART_RETURN uart_send_byte(uint8_t* data)
 *      Description: sends 8 bits of data out of UART0
 *      Parameters:
 *              - uint8_t* data: ptr to byte of data to be sent
 *      Possible Return Values:
 *              - UART_SEND_SUCCESS: data successfully sent
 *              - UART_SEND_FAILURE: data is NULL, no data sent
 *              - UART_SEND_BUFADD_SUCCESS: when INTERRUPTS switch is set,
 *                      indicates that data has been buffered.
 *              - UART_SEND_BUFADD_FAILURE: when INTERRUPTS switch is set,
 *                      indicates that data was not able to be buffered.
 **************************************************************/
UART_RETURN uart_send_byte(uint8_t* data);


/***************************************************************
 * UART_RETURN uart_send_byte_n(uint8_t* data, uint32_t length)
 *      Description: sends n bytes of data out of UART0.
 *      If interrupts are enabled, function blocks until there is room in
 *      in the TXBuf.
 *      Parameters:
 *              - uint8_t* data: pointer to data being sent
 *              - uint32_t length: number of bytes pointed at by data
 *      Possible Return Values: *
 *              - UART_SEND_SUCCESS: data successfully sent
 *              - UART_SEND_FAILURE: data is NULL, no data sent
 *              - UART_SEND_BUFADD_SUCCESS: when INTERRUPTS switch is set,
 *                      indicates that data has been buffered.
 *              - UART_SEND_BUFADD_FAILURE: when INTERRUPTS switch is set,
 *                      indicates that data was not able to be buffered.
 **************************************************************/
UART_RETURN uart_send_byte_n(uint8_t* data, uint32_t length);


/***************************************************************
 * UART_RETURN uart_receive_byte(uint8_t* buffer)
 *      Description: receives 8 bits of data from UART0 and stores it into buffer.
Blocks
 *      until byte is received.
 *      Parameters:
 *              - uint8_t* buffer: pointer to location where data is going to be
stored
 *      Possible Return Values:
 *              - UART_RECEIVE_FAILURE: buffer is NULL
 *              - UART_RECEIVE_SUCCESS: data written into buffer
 **************************************************************/
UART_RETURN uart_receive_byte(uint8_t* buffer);


/***************************************************************
 * UART_RETURN uart_receive_byte_n(uint8_t* buffer, uint32_t length)
 *      Description: receives n bytes of data from UART0 and stores into buffer.
 *      This function blocks until length bytes are received.
 *      Parameters:
 *              - uint8_t* buffer: pointer to location where data is stored.
 *              - uint2_t length: number of bytes to be read.
 *      Possible Return Values:
 *              - UART_RECEIVE_FAILURE: buffer is NULL
 *              - UART_RECEIVE_SUCCESS: length bytes received successfully.
 **************************************************************/
UART_RETURN uart_receive_byte_n(uint8_t* data, uint32_t length);

#endif /* SOURCES_UART_H_ */
```