CBFIFO.c

```
/********************************************************************************
*Copyright (C) 2020 by Arpit Savarkar
*Redistribution, modification or use of this software insource or binary
*forms is permitted as long as the files maintain this copyright. Users are
*permitted to modify this and use it to learn about the field of embedded
*software. Arpit Savarkar and the University of Colorado are not liable for
*any misuse of this material.
*
********************************************************************************/
/**
* @file cbfifo.c
* @brief An abstraction to maintain and instantiate Ciruclar Buffer
*
* This file provides functions and abstractions for handling and
* manipulating Circular Buffer
*
* @author Arpit Savarkar
* @date September 10 2020
* @version 1.0
*
*
Sources of Reference :
Online Links : https://embeddedartistry.com/blog/2017/05/17/creating-a-circular-buffer-in-c-and-c/
Textbooks : Embedded Systems Fundamentals with Arm Cortex-M based MicroControllers
I would like to thank the SA's of the course Rakesh Kumar, Saket Penurkar and Howdy Pierece
for their
support to debug the Cirular Buffer Implementation
*/

#ifndef _CBFIFO_C_
#define _CBFIFO_C_

#include "cbfifo.h"


// Checks for Global Bool Status
bool created = false;

// Definition
typedef struct cbfifo_s {
uint8_t * buff;
size_t head, tail;
size_t size;
```

```
bool full_status;
size_t storedbytes;
} cbfifo_t;

cbfifo_t my_fifo;
cbfifo_t* fifo = &my_fifo;
uint8_t CBbuffer[SIZE];


// Helper Function
bool cbfifo_empty()
{
assert(fifo);
return (!fifo->full_status && (fifo->head == fifo->tail));
}

// Helper Function
static void update_head_tail()
{
assert(fifo);
if(fifo->full_status) {
fifo->tail = (fifo->tail + 1) % fifo->size;
}
fifo->head = (fifo->head + 1) % fifo->size;
fifo->full_status = (fifo->head == fifo->tail);
fifo->storedbytes = cbfifo_length();
}

// Helper Function
static void reset_tail()
{
assert(fifo);
// Updates the full status
fifo->full_status = false;
// Since it is a cirular buffer if it resizes to
// back to position zero if the tail size overFlows
fifo->tail = (fifo->tail + 1) % fifo->size;
}

void cbfifo_create() {
// // Assigns memory pointer for the Circular Buffer
// fifo = (cbfifo_t*)malloc(sizeof(cbfifo_t));
//Contiguious Dynamic Memory allocation of upto SIZE
// fifo-> buff = (uint8_t *)malloc(SIZE * sizeof(uint8_t));
fifo-> buff = CBbuffer;
// Dynamic Memory allocation failure handling
for(int i = 0; i < SIZE; i++)
```

```
fifo-> buff[i] = 0;
if(fifo-> buff == NULL) {
exit(0);
}
// SIZE of buffer
fifo-> size = SIZE;
// Pointer to keep track of the size of the bytes in
// Circular Buffer
fifo->storedbytes = 0;

// Helper Pointers for circular buffer
fifo->head = 0;
fifo->tail = 0;

fifo->full_status = false;
created = true;
}

// Helper Function to enque data per byte
void helper_cbenque(void *buf, size_t nbyte)
{
if (buf && fifo->buff) {
// Typecasting to 8 bits
uint8_t *data = (uint8_t*) buf;
assert(fifo);
/* If the Size if not full continue to add on the byte
corresponding to head and update the head and the
tail pointer */
if(!fifo->full_status) {

// Moves the base pointer upto nbytes
for(int i =0; i< nbyte ; i++) {
fifo->buff[fifo->head] = *(uint8_t*) (data + i);
update_head_tail();
}
}
}

}


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

//Asserts that the base struct is created which handles
//The byte storage
if (!created) {
cbfifo_create();
}
// Checks for assertions
if (buf && created && nbyte>=0 && !fifo->full_status) {

// Checks if the bytes to be inserted exceeds the
// max capacity of the Circular Buffer
if(cbfifo_length() + nbyte > fifo->size) {
// Error Handling
return -1;
}
else {
// Helper Function call to Enqueue
helper_cbenque(buf, nbyte);
}
return (fifo->storedbytes);
}
else {
return -1;
}
}


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

```
uint8_t *buffer = (uint8_t*) buf;
size_t len=0;
assert(fifo && buffer);
for(uint8_t i=0; i < nbyte; i++) {
// Cannot Dequeue from an empty buffer
if(!cbfifo_empty(fifo)) {
// Stored bytes checks the size of the
// Buffer
if(fifo->storedbytes <= 0) {
// cbfifo_free();
return i;
}
// Dequues from the front where the tail is
*(uint8_t*) (buffer + i) = fifo->buff[fifo->tail];
// Updated tail status
reset_tail(fifo);
// Stores the current length
fifo->storedbytes = cbfifo_length();
len++;
}
}
// Returns the number of bytes Dequeued
return len;
}


/*
* Returns the number of bytes currently on the FIFO.
*
* Parameters:
* none
*
* Returns:
* Number of bytes currently available to be dequeued from the FIFO
*/
size_t cbfifo_length() {
assert(fifo);
size_t size = fifo->size;
if(!fifo->full_status) {
if(fifo->head >= fifo->tail) {
// When Head is ahead of Tail
size = (fifo->head - fifo->tail);
}
else {
// When Tail is ahead of head
size = fifo->size + fifo->head - fifo->tail;
}
}
// Size of the circular buffer
return size;
}


/*
* Returns the FIFO's capacity
*
* Parameters:
* none
*
* Returns:
* The capacity, in bytes, for the FIFO
*/
size_t cbfifo_capacity() {
// The Max capacity of the circular buffer
return fifo->size;
}



#endif // _CBFIFO_C_
```

CBFIFO.H

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

#ifndef _CBFIFO_H_
#define _CBFIFO_H_

#include <stdlib.h> // for size_t
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>
#include <stdio.h>
```

```c
#define SIZE 128


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


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


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


/*
* Returns the FIFO's capacity
```

```
*
* Parameters:
* none
*
* Returns:
* The capacity, in bytes, for the FIFO
*/
size_t cbfifo_capacity();

/*
* Helper function to check if the cB is empty
*
* Parameters:
* none
*
* Returns:
* none
*/
bool cbfifo_empty();

/*
* Helper function to update head and tail pointer to keep track of the CB
*
* Parameters:
* none
*
* Returns:
* none
*/
static void update_head_tail();


/*
* Helper Function to reset tail incase of over flow
*
* Parameters:
* none
*
* Returns:
* none
*/
static void reset_tail();

/*
* Helper Function to enque data per byte
*
* Parameters:
```

```
* buf Pointer to the data
* nbyte Max number of bytes to enqueue
*
* Returns:
* none
*/
void helper_cbenque(void *buf, size_t nbyte);

#endif // _CBFIFO_H_
```

LLFIFO.c

```
/******************************************************************************
*Copyright (C) 2020 by Arpit Savarkar
*Redistribution, modification or use of this software insource or binary
*forms is permitted as long as the files maintain this copyright. Users are
*permitted to modify this and use it to learn about the field of embedded
*software. Arpit Savarkar and the University of Colorado are not liable for
*any misuse of this material.
*
*******************************************************************************/
/**
* @file llfifo.c
* @brief An abstraction to maintain and instantiate Linked List Based
* Queue (FIFO)
*
* This file provides functions and abstractions for handling and
* manipulating Circular Buffer
*
* @author Arpit Savarkar
* @date September 10 2020
* @version 1.0
*
*
Sources of Reference :
Online Links : https://github.com/geekfactory/FIFO/blob/master/FIFO.h
Textbooks : Embedded Systems Fundamentals with Arm Cortex-M based MicroControllers
I would like to thank the SA's of the course Rakesh Kumar, Saket Penurkar and Howdy Pierece
for their
support to debug the Linkedlist FIFO Implementation
*/

#ifndef _LLFIFO_C_
#define _LLFIFO_C_

#include "llfifo.h"
```

```
// Definition of the Node of the Linked List
typedef struct Node {
void* key;
struct Node* next;
} node;

// Structure definition for LLFIFO
typedef struct llfifo_s {
node *front, *rear, *start;
size_t allocatednodes;
size_t storednodes;
int del_nodes;
bool created;
bool destroy;
void* val;
} llfifo_t;


// Helper Function to setup the nodes
node* newNode(void* ele, size_t capacity, llfifo_t* fifo)
{
if(!fifo->created) {
fifo->start = (node*)malloc( capacity * sizeof(node));
fifo->created = true;
}
int loc = llfifo_length(fifo);
if(loc <= fifo->allocatednodes - 1) {
node* T = fifo->start;
T[loc].key = ele;
T[loc].next = NULL;
node* temp = &T[loc];
return temp;
}
else {
node* temp2 = (node*)malloc(sizeof(node));
temp2->key = ele;
temp2->next = NULL;
return temp2;
}
}

/*
* Initializes the FIFO
*
* Parameters:
* capacity the initial size of the fifo, in number of elements
```

```
*
* Returns:
* A pointer to an llfifo_t, or NULL in case of an error.
*/
llfifo_t *llfifo_create(int capacity) {
if(capacity >= 0) {
// Dynamic Allocation of the Base
llfifo_t* fifo = (llfifo_t*)malloc(sizeof(llfifo_t));
// Initially No Nodes Stored
fifo->storednodes = 0;
fifo->val = NULL;
fifo->allocatednodes = capacity;
fifo->front = fifo->rear = NULL;
fifo->created = false;
fifo->destroy = false;
fifo->del_nodes = 0;
return fifo;
}
else
// Error Handling
return NULL;
}

/*
* Returns the number of elements currently on the FIFO.
*
* Parameters:
* fifo The fifo in question
*
* Returns:
* The number of elements currently on the FIFO
*/
int llfifo_length(llfifo_t *fifo) {
return (fifo->storednodes - fifo->del_nodes);
}

/*
* Returns the FIFO's current capacity
*
* Parameters:
* fifo The fifo in question
*
* Returns:
* The current capacity, in number of elements, for the FIFO
*/
int llfifo_capacity(llfifo_t *fifo) {
if( llfifo_length(fifo) > fifo->allocatednodes ) {
```

```
return llfifo_length(fifo);
}
else
return fifo->allocatednodes;
}

/*
* Enqueues an element onto the FIFO, growing the FIFO by adding
* additional elements, if necessary
*
* Parameters:
* fifo The fifo in question
* element The element to enqueue
*
* Returns:
* The new length of the FIFO on success, -1 on failure
*/
int llfifo_enqueue(llfifo_t *fifo, void *element) {
if(fifo) {
// Create a new LL node
node* temp = newNode(element, fifo->allocatednodes, fifo);

// If queue is empty, then new node is front and rear both
if (fifo->rear == NULL) {
fifo->front = fifo->rear = temp;
fifo->storednodes++;
return llfifo_length(fifo);
}
// Add the new node at the end of queue and change rear
fifo->rear->next = temp;
fifo->rear = temp;
fifo->storednodes++;
return llfifo_length(fifo);
}
else {
return -1;
}
}

/*
* Removes ("dequeues") an element from the FIFO, and returns it
*
* Parameters:
* fifo The fifo in question
*
* Returns:
* The dequeued element, or NULL if the FIFO was empty
```

```
*/
void *llfifo_dequeue(llfifo_t *fifo) {

if(!fifo->destroy) {
fifo->destroy = true;
}

if(fifo->del_nodes == fifo->storednodes - 1) {
node* temp = fifo->front;
fifo->val = temp->key;
fifo->del_nodes++;
return fifo->val;
}

// If queue is empty, return NULL.
// if (fifo->storednodes <= 0) {
if (llfifo_length(fifo) <= 0) {
fifo->front = NULL;
fifo->rear = NULL;
return NULL;
}
// Store previous front and move front one node ahead
node* temp = fifo->front;
fifo->front = fifo->front->next;
// If front becomes NULL, then change rear also as NULL
if (fifo->front == NULL ) {
fifo->rear = NULL;
return NULL;
}

fifo->val = temp->key;
if(llfifo_capacity(fifo) > fifo->allocatednodes) {
free(temp);
}
fifo->del_nodes++;
return fifo->val;
}

/*
* Teardown function. The llfifo will free all dynamically allocated
* memory. After calling this function, the fifo should not be used
* again!
*
* Parameters:
* fifo The fifo in question
*
* Returns:
```

```
* none
*/
void llfifo_destroy(llfifo_t *fifo) {
if (fifo) {
while( llfifo_length(fifo) > llfifo_capacity(fifo) ) {
llfifo_dequeue(fifo);
}
fifo->del_nodes = fifo->storednodes;
free(fifo->start);
free(fifo);
}
else {
free(fifo);
return;
}
}

#endif // _LLFIFO_C_
```

LLFIFO.h

```
/*
* llfifo.h - a dynamically-growing FIFO
*
* Author: Howdy Pierce, howdy.pierce@colorado.edu
* Modeified : Arpit Savarakara, arpit.savarkar@colorado.edu
*/

#ifndef _LLFIFO_H_
#define _LLFIFO_H_

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

/*
* The llfifo's main data structure.
*
* Defined here as an incomplete type, in order to hide the
* implementation from the user. You will need to define this struct
* in your .c file.
*/
typedef struct llfifo_s llfifo_t;
```

```
/*
* Initializes the FIFO
*
* Parameters:
* capacity the initial size of the fifo, in number of elements
*
* Returns:
* A pointer to an llfifo_t, or NULL in case of an error.
*/
llfifo_t *llfifo_create(int capacity);


/*
* Enqueues an element onto the FIFO, growing the FIFO by adding
* additional elements, if necessary
*
* Parameters:
* fifo The fifo in question
* element The element to enqueue
*
* Returns:
* The new length of the FIFO on success, -1 on failure
*/
int llfifo_enqueue(llfifo_t *fifo, void *element);


/*
* Removes ("dequeues") an element from the FIFO, and returns it
*
* Parameters:
* fifo The fifo in question
*
* Returns:
* The dequeued element, or NULL if the FIFO was empty
*/
void *llfifo_dequeue(llfifo_t *fifo);


/*
* Returns the number of elements currently on the FIFO.
*
* Parameters:
* fifo The fifo in question
*
* Returns:
```

```
* The number of elements currently on the FIFO
*/
int llfifo_length(llfifo_t *fifo);


/*
* Returns the FIFO's current capacity
*
* Parameters:
* fifo The fifo in question
*
* Returns:
* The current capacity, in number of elements, for the FIFO
*/
int llfifo_capacity(llfifo_t *fifo);


/*
* Teardown function. The llfifo will free all dynamically allocated
* memory. After calling this function, the fifo should not be used
* again!
*
* Parameters:
* fifo The fifo in question
*
* Returns:
* none
*/
void llfifo_destroy(llfifo_t *fifo);

#endif // _LLFIFO_H_
```

MAIN.c

```
#include "test_llfifo.c"
#include "test_cbfifo.c"

int main() {
int success = 1;

success &= llfifo_main();
success &= cbfifo_main();
if (success)
printf("All tests succeeded\n");
else
printf("NOTE: FAILURES OCCURRED\n");
}
```

MAKEFILE

```
# -*- MakeFile -*-

main: main.c
gcc main.c -o main
```

TEST_CBFIFO.c

```
/*******************************************************************************
*Copyright (C) 2020 by Arpit Savarkar
*Redistribution, modification or use of this software insource or binary
*forms is permitted as long as the files maintain this copyright. Users are
*permitted to modify this and use it to learn about the field of embedded
*software. Arpit Savarkar and the University of Colorado are not liable for
*any misuse of this material.
*
*******************************************************************************/
/**
* @file test_cbfifo.c
* @brief An abstraction to maintain and instantiate Ciruclar Buffer
* instantiated globally in cbfifo.h
*
* This file provides functions and abstractions for to test and
* manipulate Circular Buffer in cbfifo.c
*
* @author Arpit Savarkar
* @date September 10 2020
* @version 1.0
*
*
Sources of Reference :
Online Links : https://embeddedartistry.com/blog/2017/05/17/creating-a-circular-buffer-in-c-and-c/
Textbooks : Embedded Systems Fundamentals with Arm Cortex-M based MicroControllers 
I would like to thank the SA's of the course Rakesh Kumar, Saket Penurkar and Howdy Pierece 
for their 
support to debug the Cirular Buffer Implementation
*/

#include "cbfifo.c"

int test_cbfifo_enqueue()
```

```
{
typedef struct {
char element;
int expected_res;
} test_matrix_t;
int act_ret;
char str[11] = "testString";
char ch = 'a';
test_matrix_t tests[] =
{
{str[0], 1},
{str[2], 2},
{ch, 3}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
int tests_passed = 0;
char *test_result;

for(int i=0; i<num_tests; i++) {
act_ret = cbfifo_enqueue( &tests[i].element, sizeof(tests[i].element));
if (act_ret == tests[i].expected_res ) {
test_result = "PASSED";
tests_passed++;
} else {
test_result = "FAILED";
}
printf("\n %s: cbfifo_enqueue(fifo, %d) returned %d expected %d ", test_result,
tests[i].element, act_ret, tests[i].expected_res);
}

printf("\n %s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
return (tests_passed == num_tests);
}


int test_cbfifo_capacity()
{
typedef struct {
int expected_res;
} test_matrix_t;
test_matrix_t tests[] =
{
{128}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
```

```
int tests_passed = 0;
char *test_result;
size_t act_ret;
for(int i=0; i<num_tests; i++) {
act_ret = cbfifo_capacity();
if (act_ret == tests[i].expected_res ) {
test_result = "PASSED";
tests_passed++;
} else {
test_result = "FAILED";
}
printf("\n %s: cbfifo_capacity (fifo) returned %ld expected %d ", test_result,
act_ret, tests[i].expected_res);
}

printf("\n %s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
return (tests_passed == num_tests);
}



int test_cbfifo_length()
{
typedef struct {
int expected_res;
} test_matrix_t;
test_matrix_t tests[] =
{
{3}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
int tests_passed = 0;
char *test_result;
size_t act_ret;
for(int i=0; i<num_tests; i++) {
act_ret = cbfifo_length();
if (act_ret == tests[i].expected_res ) {
test_result = "PASSED";
tests_passed++;
} else {
test_result = "FAILED";
}
printf("\n %s: cbfifo_capacity (fifo) returned %ld expected %d ", test_result,
act_ret, tests[i].expected_res);
}
```

```
printf("\n %s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
return (tests_passed == num_tests);
}



int test_cbfifo_dequeue()
{
typedef struct {
void* element;
int expected_res;
} test_matrix_t;
int act_ret;
char strDump[] = "zzzzzzzzzz";
// The 3 bytes Enqueued in teh cbfifo_enqueue Function and
// Dequesed bytes here
test_matrix_t tests[] =
{
{strDump, 2},
{strDump, 1},
{strDump, 0},
{strDump, 0}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
int tests_passed = 0;
char *test_result;

for(int i=0; i<num_tests; i++) {
act_ret = cbfifo_dequeue( strDump, 2 );
if (act_ret == tests[i].expected_res ) {
test_result = "PASSED";
tests_passed++;
} else {
test_result = "FAILED";
}
printf("\n %s: cbfifo_dequeue(strDump, %d) returned %d expected %d ", test_result,
*(int*)tests[i].element, act_ret, tests[i].expected_res);
}

printf("\n %s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
return (tests_passed == num_tests);
}


int cbfifo_main()
{
```

```c
int pass = 1;
pass &= test_cbfifo_enqueue();
pass &= test_cbfifo_capacity();
pass = test_cbfifo_length();
pass = test_cbfifo_dequeue();
return pass;
}
```

TEST_LLFIFO.c

```c
/********************************************************************************
*Copyright (C) 2020 by Arpit Savarkar
*Redistribution, modification or use of this software insource or binary
*forms is permitted as long as the files maintain this copyright. Users are
*permitted to modify this and use it to learn about the field of embedded
*software. Arpit Savarkar and the University of Colorado are not liable for
*any misuse of this material.
*
********************************************************************************/
/**
* @file test_llfifo.c
* @brief An abstraction to test the functionalities of Linked List Based 
* Queue (FIFO) in llfifo.c 
*
* This file provides functions and abstractions for handling and
* manipulating Circular Buffer
*
* @author Arpit Savarkar
* @date September 10 2020
* @version 1.0
*
*
Sources of Reference :
Online Links : https://github.com/geekfactory/FIFO/blob/master/FIFO.h
Textbooks : Embedded Systems Fundamentals with Arm Cortex-M based MicroControllers 
I would like to thank the SA's of the course Rakesh Kumar, Saket Penurkar and Howdy Pierece 
for their 
support to debug the Linkedlist FIFO Implementation
*/

#include "llfifo.c"

int test_llfifo_create()
{
typedef struct {
int capacity;
llfifo_t *expected_val;
} test_matrix_t;

llfifo_t* fifo =llfifo_create(0);
test_matrix_t tests[] =
{
{-1, NULL},
{0, NULL},
{1, fifo},
{2, fifo}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
int tests_passed = 0;
char *test_result;

for(int i=0; i<num_tests; i++) {
fifo = llfifo_create(tests[i].capacity);

if ((tests[i].capacity >=0) && (fifo !=NULL)) {
test_result = "PASSED";
tests_passed++;
printf(" \n %s: llfifo_create(%d) returned %ld", test_result, tests[i].capacity, fifo-
>allocatednodes);
}
else if ( ( (fifo == NULL) && tests[i].capacity <0)) {
test_result = "PASSED";
tests_passed++;
printf(" \n %s: llfifo_create(%d) returned NULL", test_result, tests[i].capacity);
}
else {
printf(" \n %s: llfifo_create(%d) returned Illegal size", test_result, tests[i].capacity);
test_result = "FAILED";
}

}

printf("\n%s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
llfifo_destroy(fifo);
return (tests_passed == num_tests);
}

int test_llfifo_enqueue()
{
llfifo_t* fifo;
llfifo_t* gigo;
fifo = llfifo_create(6);
```

```
gigo = llfifo_create(-1);
typedef struct {
void* element;
int expected_res;
} test_matrix_t;
int act_ret;
size_t temp_len = 10;
act_ret = llfifo_enqueue(gigo, &temp_len);
llfifo_destroy(gigo);

if (act_ret == -1 ) {
printf("\n PASSED: Enquining to Uninitialized FIFO returned null ");
} else {
printf("\n Uninitialized test failed ");
}

typedef struct test_struct {
int x;
char y;
} test_st;
test_st object;
object.x = 1;
object.y = 'a';
test_matrix_t tests[] =
{
{(void*)INT8_MAX, 1},
{(void*)INT16_MAX, 2},
{(void*)INT32_MAX, 3},
{&object, 4},
{NULL, 5}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
int tests_passed = 0;
char *test_result;

for(int i=0; i<num_tests; i++) {
act_ret = llfifo_enqueue(fifo, tests[i].element);

if (act_ret == tests[i].expected_res ) {
test_result = "PASSED";
tests_passed++;
} else {
test_result = "FAILED";
}
printf("\n %s: llfifo_enqueue(fifo, %p) returned %d expected %d ", test_result,
tests[i].element, act_ret, tests[i].expected_res);
```

```
}

printf("\n %s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
llfifo_destroy(fifo);
return (tests_passed == num_tests);
}

int test_llfifo_dequeue()
{
llfifo_t* fifo;
fifo = llfifo_create(6);
int a = INT8_MAX;
int b = INT16_MAX;
int c = INT32_MAX;
char str[] = "papiha";

size_t len = llfifo_enqueue(fifo, &a);
len = llfifo_enqueue(fifo, &b);
len = llfifo_enqueue(fifo, &c);
len = llfifo_enqueue(fifo, str);
len = llfifo_enqueue(fifo, &a);
len = llfifo_enqueue(fifo, &b);
len = llfifo_enqueue(fifo, &c);
if(len==0) {
len = 0;
return 0;
}
typedef struct {
void* expected_res;
} test_matrix_t;
void* act_ret;
test_matrix_t tests[] =
{
{&a},
{&b},
{&c},
{str},
{&a},
{&b},
{&c}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
int tests_passed = 0;
char *test_result;

for(int i=0; i<num_tests; i++) {
```

```
act_ret = llfifo_dequeue(fifo);
if ( *(int*)act_ret == *(int*) tests[i].expected_res) {
test_result = "PASSED";
tests_passed++;
} else {
test_result = "FAILED";
}
printf("\n %s: llfifo_dequeue() returned %d expected %d ", test_result,
*(int*)act_ret, *(int*) tests[i].expected_res);
}

printf("\n %s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
llfifo_destroy(fifo);
return (tests_passed == num_tests);
}

int test_llfifo_capacity() {

llfifo_t* fifo;

typedef struct {
int expected_res;
} test_matrix_t;
int act_ret;
test_matrix_t tests[] =
{
{1},
{2},
{3}
};

const int num_tests = sizeof(tests) / sizeof(test_matrix_t);
int tests_passed = 0;
char *test_result;
for(int i=0; i<num_tests; i++) {
fifo = llfifo_create(tests[i].expected_res);
act_ret = fifo->allocatednodes;
if ( act_ret == tests[i].expected_res) {
test_result = "PASSED";
tests_passed++;
} else {
test_result = "FAILED";
}
printf("\n %s: llfifo_length() returned %d expected %d ", test_result,
act_ret, tests[i].expected_res);
}
```

```
printf("\n %s: PASSED %d/%d\n", __FUNCTION__, tests_passed, num_tests);
llfifo_destroy(fifo);
return (tests_passed == num_tests);
}

int llfifo_main() {
int pass = 1;
pass &= test_llfifo_create();
pass &= test_llfifo_enqueue();
pass &= test_llfifo_dequeue();
pass &= test_llfifo_capacity();
return pass;
}
```

README.md

```
# PES-Assignment-1
Author: Arpit Savarkar

## Repository Comments
_Contains_
Code for Assignment 2 for PES, ECEN-5813, Fall 2020

Repository for PES-Assignment 1

- <b>llfifo.h - Header file which contains the function prototypes and enumerators needed for llfifo.c</b>
- <b>llfifo.c - The main script for instantiating and testing a linkedlist based Queue</b>
- <b>cbfifo.h - Header file which contains the function prototypes and enumerators needed for cbfifo.c</b>
- <b>cbfifo.c - The main script for instantiating and testing a Circular Buffer based Queue</b>

## Circular Buffer based Queue
cbfifo.h Involves Four Functions and Unit Tests and helper functions for the following
1) cbfifo_enqueue(void *buf, size_t nbyte
- Returns the number of bytes requested to be enqueued on a linkedlist based implementation of a linkedlist. Enqueues data onto the FIFO, up to the limit of the available FIFO capacity The number of bytes actually enqueued, which could be 0. In case of an error, returns -1.

2) cbfifo_dequeue(void *buf, size_t nbyte)
- Attempts to remove ("dequeue") up to nbyte bytes of data from the FIFO. Removed data will be copied into the buffer pointed to by buf. Returns The number of bytes actually copied, which will be between 0 and nbyte. In case of an error, returns -1.
```

3) cbfifo_length()
- Returns the Number of bytes currently available to be dequeued from the FIFO

4) cbfifo_capacity()
- Returns the capacity, in bytes, for the FIFO

==
===
## Linked List Based Queue
1) llfifo_create(int capacity)
- Initializes the FIFO and A pointer to an llfifo_t, or NULL in case of an error.

2) llfifo_enqueue(llfifo_t *fifo, void *element)
- Enqueues an element onto the FIFO, growing the FIFO by adding additional elements, if necessary. The new length of the FIFO on success, -1 on failure

3) llfifo_length(llfifo_t *fifo)
- Returns the number of elements currently on the FIFO.

4) llfifo_dequeue(llfifo_t *fifo)
- Removes ("dequeues") an element from the FIFO, and returns it

5) llfifo_capacity(llfifo_t *fifo)
- Returns The current capacity, in number of elements, for the FIFO

4) llfifo_destroy(llfifo_t *fifo)
- Teardown function. The llfifo will free all dynamically allocated memory. After calling this function, the fifo should not be used again!

## Assignment Comments
This assignment demonstrates C Programming from scratch for data representation conversion and FIFO Based implementation using both LinkedList and Ciruclar Buffer, it also demonstrates a code for testing the specified data structures.

## Execution
- To run the Program (Linux) :
1) make
2) ./main