ASSIGNMENT 1.2 – NEW BUFFER SPECIFICATION

General View

Due Date: prior or on **Oct 2nd 2021 (midnight)**

- **2nd Due date** (until 9th Oct) - 50% off.

Earnings: 5% of your course grade

Purpose: Programming and Using Dynamic Structures (buffers) with C

❖ This is the SECOND new task in Compilers: You need to create your buffer to be used in your new language. Follow the example of **SOFIA** implementation.

❖ This is a review of and an exercise in:
  - C coding style
  - Programming techniques
  - Data types and structures
  - Memory management, and
  - File input/output.

❖ It will give you a better understanding of the type of internal data structures used by a simple compiler you will be building this semester. This assignment will be also an exercise in "*excessively defensive programming*".

❖ You are to write / complete functions that should be "overly" protected and should not abruptly terminate or "**crash**" at run-time due to invalid function parameters, erroneous internal calculations, or memory violations. To complete the assignment, you should fulfill the following two tasks:

❖ The current version of code requires *Camel Code style* (ex: "bufferCreate"). Use it appropriately.

Buffer (before start coding)

Here are some tips (not exactly in a "logical" sequence of steps), with some ideas to help you during the development of A12 (**Buffer**):

- Please read the Assignment **Submission Standard and Marking Guide** (at "Assignments > Standards" section).
- To do this activity:
  - You will see that you need to:
    - Download the files (see the **ZIP** file at A12 Assignment).
    - In your Project, import the .C and .H files in (including **Compilers.c, Compilers.h, Buffer.h, Buffer.c** and **MainBuffer.c**.

Note 1: Progressive Development

During this course additional code will be provided. The suggestion is that you can include in your project (already configured to use ANSI C).

Buffer implementation

- You will see a partial code developed to SOFIA ("**Buffer.h**" and "**Buffer.c**"), check all **TODO** comments.

1.1. COMPILER – Header file

- In this file (**Compiler.h**), you need to define the main elements used in all assignments and basic definitions are used for Buffer, Scanner and Parser.
- The most important part is to use typedefs to define your own language datatypes.

ACTION

Ex: If your language is called "Rio", then, your datatypes are supposed to be something like "rio_int" (or similar) etc.

- Before start, remember that the buffer is using datatypes related to the language. It is done by typedef definitions linking ANSI C datatypes with the language (ex: **SOFIA**) datatypes. Check the *header* about these definitions (that you can continue):

```
typedef short              sofia_int;    /* Adjust for your language */
typedef char               sofia_chr;    /* Adjust for your language */
typedef unsigned char      sofia_flg;    /* Adjust for your language */
typedef char               sofia_bol;    /* Adjust for your language */
```

1.2. COMPILER – Source code

- ❖ In **Compiler.c**, you will see the main method of the project. Essentially, according to inputs, specific functions will be called: (Ex: "1" will invoke "mainBuffer()", etc.).
  - ➢ **TIP**: You just need to comment the code to invoke the appropriate function.

1.3. BUFFER STRUCTURES – Header file

Note 2: About Buffers

***Buffers** are often used when developing compilers because of their efficiency (see page 111 of your textbook).*

The buffer implementation is based on data structures: The Buffer and the Position. The buffer will be created "on demand" at run time, that is, they are to be allocated dynamically.

*The buffer is used to control data the main structure (the **content** of the buffer – remember that in C, we have to use pointer to chars) – that contains all the necessary information about the array of characters.*

We need also to consider some elements: a pointer to the beginning of the character array location in memory, the current size, the next character entry position, the increment factor, the operational mode and some additional parameters.

ACTION

Use your datatypes (defined in Compilers.h) to define all variables using your language prefix.

- In the code, you have a generic definition (see the enum BUFFERMODES and the BUFFER_ERROR), that should not be changed.
- However, several constants are included and must be adjusted according to your specification. So: Use your language name and, if required, change the values (this is optional).
  - For instance, when you see "BUFFER_DEFAULT_SIZE" and other constants, you can change them or not.

/* Adjust for your language */

- ACTION: *Define the alias to your language – obviously, you need to adapt your original datatype to an ANSI C Datatype*.

The following structure declaration must be used to implement the Buffer:

```
/* Buffer structure and pointer */
typedef struct buffer {
        sofia_chr* content;        /* pointer to the beginning of character array (character buffer) */
        sofia_int size;           /* current dynamic memory size (in bytes) allocated to character buffer */
        sofia_int increment;   /* character array increment factor */
        sofia_chr mode;         /* operational mode indicator*/
        sofia_flg flags;          /* contains character array reallocation flag and end-of-buffer flag */
        Position position;
} Buffer, * BufferPointer;
```

And:

```
/* Offset information */
typedef struct position {
        sofia_int writePos;     /* the offset (in chars) to the add-character location */
        sofia_int readPos;      /* the offset (in chars) to the get-character location */
        sofia_int markPos;      /* the offset (in chars) to the mark location */
} Position;
```

- ACTION: Define the Buffer and Position using your language datatypes.

PART I - BUFFER STRUCTURE EXPLANATION

Where:

- ***content*** is the pointer that indicates the beginning of useful information (loaded from a source file).
- ***size*** is the current total size (measured in bytes) of the memory allocated for the character array by ***malloc()/realloc()*** functions. In the text below it is referred also as current size. It is whatever value you have used in the call to ***malloc()/realloc()*** that allocates the storage pointed to by ***string***.
- ***increment*** is a buffer increment factor. It is used in the calculations of a new buffer **size** when the buffer needs to grow. The buffer needs to grow when it is full but still another character needs to be added to the buffer. The buffer is full when ***addC offset*** measured in bytes is equal to ***size*** and thus all the allocated memory has been used. The ***increment*** is only used when the buffer operates in one of the "self-incrementing" modes and it must respect the integer definition of your language.
  - In "**additive** self-incrementing" mode the integer data increment is used in ADDITION to the current size.
  - In "**multiplicative** self-incrementing" the integer data increment is used to MULTIPLY the current value.
  - In "**fixed**" mode, once defined the size, the buffer size cannot increase.
- ***mode*** is an operational mode indicator. It can be set to three different values that must be defined in your buffer header: given by the **enum MODES**).

- ***flags*** is a field containing different flags and indicators.
    - Each flag or indicator uses one or more bits of the ***flags*** field.
        - The flags usually indicate that something happened during a routine operation (end of file, end of buffer, integer arithmetic sign overflow, and so on).
        - Multiple-bit indicators can indicate, for example, the mode of the buffer (three different combinations – therefore 2-bits are needed). In this implementation the ***flags*** field has the following structure:
    - In SOFIA, we have:

Bit 7 (MSB)							Bit 0 (LSB)
-	**-**	**-**	**-**	**REL**	**EOB**	**FUL**	**EMP**

- Note that the sequence of 1111.0000 (1 byte) correspond to the value **F0** hexadecimal. In short:
    - It means that the DEFAULT (initially created) is F0.

Note 2: Remembering Flags

In cases when storage space must be as small as possible, the common approach is to pack several data items into single variable; one common use is a set of single-bit or multiple-bit flags or indicators in applications like compiler buffers, file buffers, and database fields.

The flags are usually manipulated through different bitwise operations using a set of "masks." Alternative technique is to use bit-fields.

Using bit-fields allows individual fields to be manipulated in the same way as structure members are manipulated.

Since almost everything about bit-fields is implementation dependent, this approach should be avoided if the portability is a concern. In this implementation, you are to use bitwise operations and masks.

- The LSB (***EMP***) bit is used to identify that Buffer is empty (ex: when it is created), and **when set to 1, it indicates that buffer content is empty**. You need to adjust it appropriately when creating and updating the buffer. Use also check bit comparison in the function isEmpty().
- The 2nd LSB (***FUL***) bit is used to identify that Buffer is full (ex: when the position given by writePos has achieved the MAX_SIZE), and **when set to 1, it indicates that buffer content is not possible to include chars** (unless the size is increased by one of the modes: **additive** or **multiplicative**). You need to adjust it appropriately when creating and updating the buffer. Use also check bit comparison in the function isFull().
- The 3rd LSB (***EOB***) bit is by default **0**, and **when set to 1, it indicates that the end of the buffer content has been reached** during the buffer read operation

(***getChar()*** function). If ***EOB*** is set to **1**, the function ***getChar()*** should not be called before the *readPos* is reset by another operation.

  - When you have read the buffer, once you achieve the end of the buffer, the last bit (EOB) must be SET.

- The 4th LSB **(*RLB*)** is a single-bit reallocation flag. It is by default **0**, and **when set to 1, it indicates that the location of the buffer character array in memory has been changed due to memory reallocation**. This could happen when the buffer needs to expand or shrink. The flag can be used to avoid dangling pointers when pointers instead of offsets (positions) are used to access the information in the character buffer.
  - When the reallocation of the buffer is called, if it is created a new memory position, the RLB bit must be SET.
- The remaining bits can be reserved for further use and **must be set by default to 1**. When setting or they must not be changed by the bitwise operation manipulating bit 0 and 1.

- ACTION: Define the values for bit-wise operations using the flags in the buffer structure to accommodate a ***NEW definition:***

```
/* Add your bit-masks constant definitions here */
/* TO_DO: BIT 0: EMP: Empty */
/* TO_DO: BIT 1: FUL = Full */
/* TO_DO: BIT 2: EOB = EndOfBuffer */
/* TO_DO: BIT 3: REL = Relocation */
```

- **Attention**: the operations over bits (SET, RST and CHK) will be explained during lectures and are also described in the **Lecture Notes**.

PART II – POSITION STRUCTURE EXPLANATION

About the offsets, we have:

❖ ***writePos*** is the distance (measured in chars) from the beginning of the character array (***content***) to the location where the next character is to be added to the existing buffer content.

  - ***writePos*** must never be larger than ***size***, or else you are overrunning the buffer in memory and your program may crash at run-time or destroy data.

❖ ***readPos*** is the distance (measured in chars) from the beginning of the character array (***string***) to the location of the character which will be returned if the function ***getChar()*** is called.

  - The value ***readPos*** must never be larger than ***writePos***, or else you are overrunning the buffer in memory and your program may get wrong data or crash at run-time. If the value of ***readPos*** is equal to the value of ***writePos***, the buffer has reached the **end** of its current content.

- ***markPos*** is the distance (measured in chars) from the beginning of the character array (***string***) to the location of a ***mark***.
    - A *mark* is a location in the buffer, which indicates the position of a specific character (for example, the beginning of a word or a phrase). It will be used in the Scanner, but the implementation must be done in buffer.
- ACTION: Check all **TO_DO** labels related to activities:
    - All constant definitions, data type and function declarations (prototypes) must be located in the buffer header file (remember to rename according to your language name).
        - **TIP_01**: You need to read the following section (about functional specification) to adjust the values appropriately.
    - In the end of the header, you can see all function declarations that will be used in buffer.

Buffer test

NOTE
When you download the files, you will see that the compilation will succeed. So, when you update your files, you need to have a "perfect" compilation – no errors / warnings.

2.1. GENERAL VIEW

To test your program, you are to use the test harness program ***MainBuffer.c*** and test it with input files.

- **TIP_04**: In your main buffer file, you must have just ONE reference to your header:

    ```
    #include "Buffer.h"
    ```

- For this reason, you have some constants in your buffer header file that are considered "language neutral". For instance: BUFFER_ERROR.

- ***Important Note:*** Because each project is implementing a specific language, you need to **create input files** (using your EXTENSION language name) that can be read by your language. For instance, you will see inputs such as:
    - An empty file (**A12A_Empty.sof**);
    - A Hello World file (**A12A_Hello.sof**);
    - A basic file about the Sofia City (**A12C_Basic.sof**);
    - A big file for test (**A12C_Big.sof**);

- However, remember that your buffer must be able to work with **ANY text file**.

❖ **About input files**:

- In the case of SOFIA, using the extension, you will find **".sof"** files.
    - Use an **empty file** – see the corresponding SOFIA empty file **a12_empty.sof** *note that this is really an empty text file).*
    - Use normal files (*you need to create according to your language*) for testing the buffer.
    - And a big file (it is not required to be "real" code) for testing limits of your buffer. For instance, look the way to create long strings (*note that this is really an enormous string*).
        - Ex: the well-known randomized words "Loren Ipsum" (see: https://www.lipsum.com/)
        - You can test buffer modes and parameters with a "Big file" – it means that your input must have about 32.000 chars (less than your "integer" definition).

❖ For standard tests, inputs can be tested using:

- By default, the Fixed mode ("**f**") mode.
- But you need also to test using the modes "**a**" (Additive) and "**m**" ("Multiplicative") modes.

❖ The corresponding output files (remember to redirect the outputs) must be also provided:

- Remember that the way you can generate must be similar to this:

```
<PROGRAM.EXE> <BUFFER_OPTION="0"> <INPUT_FILE>  > <OUTPUT_FILE> 2> <ERROR_FILE>
```

- **Example:**
    - Suppose that your project is "Rio".
    - The value "0" is related to the option to execute buffer, as it will be shown in the next section.

```
Rio.exe 1 hello.rio > hello.out 2> hello.err
```

❖ **About outputs**:

- ACTION: So, for each input, provide the *default output* and the *default error* file:
    - You will find similar outputs for standard output (with **".out"** extension)
    - And also, for standard error (with **".err"** extension).

❖ ***Important Note:*** All files are supposed to have only plain ASCII characters.

NOTE

When you submit, your input (using your extension language name), and output and error files must be included. It is expected that all the errors files must be empty.

2.2. ABOUT EXECUTION - Example

Here is a brief description of the program that is provided for you on Brightspace (BS).

- You will find a unique main file (**Compilers.c**) with the corresponding header file (**Compilers.h**).
  - This file is already prepared to execute all activities in this course (A12 - buffer, A22 - scanner and A32 - parser). So, you do not need to change it.
  - The execution for buffer is given by this command (suppose that "**Rio**" is the name of your city - is your project name):

```
Rio.exe 1 hello.rio > hello.out 2> hello.err
```

    - When "1" (argv[1]) is passed, mainBuffer (function in the **MainBuffer.c** file) is invoked – the other options will be available in future assignments ("2" = Scanner and "3" = Parser).
- The main buffer program (***Compilers.c***) takes to parameters from the command line and for buffer execution, it is required:
  - An input file name and a character (**F** – ***MODE_FIXED***, **A** – ***MODE_ADDIT***, or **M** – ***MODE_MULTI***) specifying the buffer operational mode.
  - For testing additional files, include the size and the increment.
- ***NOTE:*** Your program must not overflow any buffers in any operational mode, no matter how long the input file is.
- **TIP**: **In this assignment, the provided main program will not test all your functions.** You are strongly encouraged to test all your buffer functions with your own test files and modified main function.

2.3. ABOUT BATCH

- Besides the code, you need to submit a batch file with the complete execution of all scenarios (empty file, hello, basic file – with different datatypes – and big file).
- For this reason, we are including one specific example in Sofia Language:

```
:: SCRIPT A12 - CST8152 - Fall 2021
::
:: COMPILERS COURSE - SCRIPT -------------------------------------------

SET COMPILER=Buffer.exe
SET FILE1=A12A_Empty
SET FILE2=A12B_Hello
SET FILE3=A12C_Basic
SET FILE4=A12D_Big

REM ---------------------------------------------------------------------
REM - Begin of Tests (A12 - F21) ----------------------------------------
REM ---------------------------------------------------------------------

ren *.exe %COMPILER%

::
:: BASIC TESTS  ----------------------------------------------------------
::
REM - Basic Tests (A12 - F21) - - - - - - - - - - - - - - - - - - - - - -

%COMPILER% 1 %FILE1%.%EXTENSION%          > %FILE1%.%OUTPUT%        2> %FILE1%.%ERROR%
%COMPILER% 1 %FILE2%.%EXTENSION%          > %FILE2%.%OUTPUT%        2> %FILE2%.%ERROR%
%COMPILER% 1 %FILE3%.%EXTENSION%          > %FILE3%.%OUTPUT%        2> %FILE3%.%ERROR%

::
:: ADVANCED TESTS  -------------------------------------------------------
::
REM - Advanced Tests (A12 - F21) - - - - - - - - -- - - - - - - - - - - -

%COMPILER% 1 %FILE4%.%EXTENSION%          f 100 10   > %FILE4%-f-100-10.%OUTPUT%        2> %FILE4%-f-100-10.%ERROR%
%COMPILER% 1 %FILE4%.%EXTENSION%          a 100 10   > %FILE4%-a-100-10.%OUTPUT%        2> %FILE4%-a-100-10.%ERROR%
%COMPILER% 1 %FILE4%.%EXTENSION%          m 100 10   > %FILE4%-m-100-10.%OUTPUT%        2> %FILE4%-m-100-10.%ERROR%
REM ---------------------------------------------------------------------
REM - End of Tests (A12 - F21) ------------------------------------------
REM ---------------------------------------------------------------------
```

- In “Submission Pattern”, it is mentioned that you must include:
  - The code with the adaptation for your language.
  - Inputs, standard outputs and error output files.
  - **Test Plan**: The way you can show / describe how to execute your program (ex: script / batch or simply command line arguments).
- Finally, remember to avoid problems with warnings, late submission or plagiarism.

Buffer evaluation

Marking Rubric

Maximum Deduction (%)	Deduction Event
CRICTICAL	**Severe Errors**
100	Late submission (after 1 week due date)
Up to 100	Plagiarism detection
up to 100	Does not compile in the environment (considering **Visual Studio 2019**)
up to 100	Compile but crashes after launch
PATTERN	**Non-compliance**
Up to 20	Language adaptation (missing elements in the buffer)
Up to 10	Missing files (input, output, error)
Up to 10	Missing script (test plan)
BASIC	**Execution**
Up to 30	Problems with small files (empty, hello, basic)
Up to 30	Big file errors (problems with modes – 'f', 'a', 'm')
ADDITIONAL	**Small problems**
1 per each	Warnings
Up to 20	Missing defensive programming
up to 10	Other minor errors
up to 10	Bonus: clean code, elegant code, enhancements, discretionary points.
Final Mark	**Formula: 5***((100- ∑ penalties + bonus)/100), max score 5%**.**

Appendix – Explaining Functionalities

- ***IMPORTANT NOTE:*** **Since the code has been provided for SOFIA,** just adjust the Buffer functionalities to execute in your language (see the "TO_DO" comments in code).

1.4. BUFFER FUNCTIONALITIES (C File)

You are to implement the following set of buffer utility functions (operations). All function definitions must be stored in a file named by ***Buffer.c***. Later they will be used by all other parts of the compiler when a temporary storage space is needed.

The **first implementation step** is adjust the inclusion of header file. All functions must be the validation (if possible and appropriate) of the function arguments. If an argument value is invalid, the function must return immediately an appropriate failure indicator.

Note 3: Programming Best Practices

In Compilers, you are invited to show your best practices. Some of they are already illustrated here:

- Standard for codification (here, we are using Camel syntax to make code better readable. Especially when we are not using OO paradigm, the construction of methods and variables should be understood by anyone.

- Boundary conditions: all codes should consider problem (normally during their initialization = parameter checking). See several "Error conditions" shown below.

*- Use **DEFENSIVE PROGRAMMING**: Test not only the initial conditions, but rules from specification carefully. For instance, evaluate all important parameters that you are receiving in one specific function, before perform one action.*

Now, we will explain each function in buffer (the code has already been provided for you):

- ***create(size, increment, mode);***
  - This function creates a new buffer in memory (on the program heap), trying to allocate memory for one ***buffer*** structure using ***calloc()***;
  - It tries to allocates memory for one dynamic character buffer (***content***) calling ***malloc()*** with the given initial capacity ***(size)***;
  - The buffer is supposed to use the parameters passed to initialize the fields size, increment and mode, but some adjustments must be done:
    - If there is no value for ***size***, default values (for size and increment) are used.

    - If there is no value for ***increment***, the ***mode*** must be fixed.
    - If mode is different from valid options (Fixed, Additive or Multiplicative), no buffer must be created.
    - The ***flags*** must be initialized by default values defined in the header.
  - ***Return value***: Buffer pointer.
  - ***Error Condition:*** If run-time error occurs, the function must return immediately after the error is discovered. Check for all possible errors which can occur at run time. Do not allow **"memory leaks", "dangling"** pointers, or "**bad**" parameters.

- ***addChar(pBuffer, ch)***

  - This function is responsible to include a char in the buffer. Because of the limit (given by ***size***), several actions should be done before simply include it in the end of the buffer.
    - Because there is a possibility that the buffer can be already full, while the maximum size is not achieved (for instance, SOFIA_MAX_VALUE), if the mode is not fixed, it is required to reallocate additional space (using C function to realloc).
  - If the buffer can include a char (it is not full), just include it in the current position given by ***writePos*** offset and increment this position.
  - Otherwise, if the buffer is full, some actions must be done:
    - Start using a bitwise operation the function resets the RLB bit on ***flags***.
    - If mode is MODE_FIXED, it is not possible to allocate additional space and function ends.
    - If the mode is MODE_ADDIT, it is required to increase the size by using a linear formula:

      newSize = pBuffer->size + pBuffer->increment;

    - If mode is multiplicative (MODE_MULTI) the formula is given by:

      newSize = pBuffer->size * pBuffer->increment;

    - In both cases, it is required to check if the result is valid (not negative and lower than MAX_VALUE) and in case of error, no inclusion can be done.
    - Otherwise, if it is detected that the memory address has been changed, the **RLC** bit must be set (use comparison between addresses to check this).
    - If everything is ok, you can use ***realloc***() to increase the size of the ***content***.

        - However, remember to use *defensive programming*: call realloc using a temporary variable and avoid to destroy the original buffer.
    - Finally, add the character ***ch*** to the character array of the given ***string*** pointed by ***pBuffer***.
  - ***Return value***: Buffer pointer.
  - ***Error Condition:*** The function must return NULL on any error.
    - Some of the possible errors are indicated above but you must check for all possible errors that can occur at run-time.
    - Do not allow "**memory leaks**". Avoid creating "**dangling pointers**" and using "**bad**" parameters.
    - **TIP_02**: The function ***must not destroy*** the buffer or the contents of the buffer even when an error occurs – it must simply return NULL leaving the existing buffer content intact.

- ***clear(pBuffer)***
  - The function retains the memory space currently allocated to the buffer.
  - It reinitializes all appropriate data members of the given ***buffer*** structure so that the buffer will appear as just created.
  - In short, offsets and flags must be reset.
  - ***Return value***: Boolean value defined by your language.

- ***destroy(pBuffer)***
  - The function de-allocates (frees) the memory occupied by the character buffer and the ***buffer*** structure.
  - ***Return value***: Boolean value defined by your language.

- ***isFull(pBuffer)***
  - The function checks if the character buffer is full.
    - Basically, it is required to evaluate the bit-wise operation for checking the value of the bit flag FUL from flags.
  - ***Return value***: Boolean value defined by your language.

- ***getWritePos(pBuffer)***

- The function returns the current ***size*** (given by the ***writePos*** offset)
- ***Return value***: Integer value defined by your language.

❖ ***getMode(pBuffer)***

- The function returns the current ***mode*** (fixed / additive / multiplicative).
- ***Return value***: Value defined by your language.

❖ ***getMarkPos (pBuffer)***

- The function adjusts the ***mark*** offset.
- ***Return value***: Boolean value for this operation.

❖ ***setMark(pBuffer, mark)***

- The function sets a new value to ***mark*** offset.
- ***Return value***: Boolean value defined by your language.

❖ ***print(pBuffer)***

- This function is intended to print the content of buffer (in ***content*** field).
- Using the ***printf()*** library function the function prints character by character the contents of the character buffer to the standard output (***stdout***).
- In a loop the function prints the content of the buffer calling ***getChar ()*** and **checking the flags** in order to detect the **EOB - End Of the Buffer** content (using other means to detect the end of buffer content will be considered a significant specification violation).
- ***Return value***: The number of characters printed.

❖ ***int load(pBuffer, fi)***

- The function loads (reads) an open input file specified by ***fi*** into a buffer specified by ***pBuffer***.
  - **Note:** The file is supposed to be a plain file (for instance, a source code).
- **TIP_03**: The function must use the standard function ***fgetc(fi)*** to read one character at a time and the function ***addChar()*** to add the character to the buffer.

    - The operation is repeated until the standard macro ***feof(fi)*** detects end-of-file on the input file. The end-of-file character must not be added to the content of the buffer.
    - The standard macro ***feof(fi)*** can be used to detect end-of-file on the input file.
  - ***Return value***: The function returns the number of characters read from the file stream.
  - ***Error Condition:*** If the current character cannot be added to the buffer (***addChar()*** returns NULL. It is also necessary to use the ANSI C function ***ungetc()*** when if the function gets an error.
    - ***Note:*** This "***ungetc***()" operation is required because this latest char could not be included into the buffer.

- ***isEmpty(pBuffer)***
  - This function checks if the buffer is empty.
    - Basically, it is required to evaluate the bit-wise operation for checking the value of the bit flag EMP from flags.
  - ***Return value***: Boolean value defined by your language.

- ***getChar (pBuffer)***
  - This function is used to read the buffer. The function performs the following steps:
    - To get the current char during the read process, the ***readPos*** offset is used (and incremented).
    - If ***readPos*** offset and ***writePos*** offset are equal, using a bitwise operation it sets the ***flags*** field ***EOB*** bit flag to **1** and returns number the end of string (**'\0'**); otherwise, using a bitwise operation it resets ***EOB*** to **0.**
  - ***Return value***: The character located at ***readPos*** position.

- ***recover (pBuffer)***
  - The function sets both ***readPos*** and ***markPos*** offset to 0, so that the buffer can be reread again.
  - ***Return value***: Boolean value defined by your language.

- ***retract* (pBuffer)**
  - The function simulates the operation to “unread” one char from the buffer.
    - It is a logical function that decrements ***readPos*** offset by **1**.
    - It means that it is required to be sure that ***readPos*** is positive.
  - ***Return value***: Boolean value defined by your language.

- ***restore* (pBuffer)**
  - The function sets ***readPos*** offset to the value of the current ***markPos*** offset.
    - It will be used when it is necessary to “undo” several reading operations (and, for this reason, ***markPos*** offset is used).
  - ***Return value***: Boolean value defined by your language.

- ***getReadPos* (pBuffer)**
  - The function returns ***readPos*** offset to the calling function.
  - ***Return value***: Integer that represents the ***readPos*** value.

- ***getIncrement* (pBuffer)**
  - The function returns the value of ***increment*** to the calling function.
  - ***Return value***: Integer that represents the ***increment*** value.

- ***getContent* (pBuffer, pos)**
  - The function returns a pointer to the location of the character buffer indicated by ***pos*** that is the distance (measured in chars) from the beginning of the character array (***content***).
    - Because you can define the position, it is required to test it: the ***pos*** value must be between 0 and ***writePos*** offset.
  - ***Return value***: A pointer to char (string in C) that starts with the position ***pos*** after the ***string*** reference in buffer.

- ***getFlags* (pBuffer)**
  - The function returns the ***flags*** field from buffer.

- Note that the entire flag must be returned (not only one specific bit).

- ***Return value***: The value of ***flags*** in the appropriate datatype defined by your language.

File update: Sep 19th 2021.

Good luck with A12!