Introduction

This first assignment is intended to let you refresh some basic C skills and also gain familiarity with logical bit operations. The project is due on Wednesday, September 4 at 9:00 am Mountain time and should be submitted on Canvas. Your submission will consist of :

- A single private GitHub repo URL which will be accessed by SAs to review your code and documentation. For this assignment, this repo should have at least one .c file, along with a README.md file which (at minimum) documents how to build your code. See details for setting up your GitHub repo later in this document.

You may consult with other students, the SAs, and the instructor in reviewing concepts and crafting solutions to problems (and may wish to credit them in documentation).

For this assignment, you may NOT use standard C library functions, including the printf family, in your production code. However, you may use such functions in any of the test code.

Assignment Detail

For this assignment, you will create a C program with seven functions:

```
uint8_t rotate_right(uint8_t num, uint8_t nbits)
```

Rotate the bits of *num* to right by *nbits* places. The rotation uses logical shifts and not arithmetic shifts. A rotation (or circular shift) is an operation similar to shift except that the bits that shifted out on the right are put back at the most significant position i.e. left. There is no restriction in the values that *num* or *nbits* can take other than they are confined to be a uint8_t type.

Returns the new value in uint8_t type with the bits rotated.

Examples:

num	num in binary	nbits	Return value
1	0b1	1	0b10000000
1	0b1	2	0b01000000
2	0b10	4	0b00100000
1	0b1	8	0b00000001

`uint32_t binstr_to_uint(const char *str)`

Returns an unsigned integer represented by the binary string input *str* which is null terminated. The binary string must be of the form '0bcccccc' where each 'c' must be a '0' or '1'. For a valid input, the number (or count) of binary digits 'c' must be between 1 and 32. For any errors, the return value is 0xFFFFFFFF(which basically renders the maximum uint32_t value as illegal).

The 'b' in the representation must be lower case.

Examples:

str	**Return value**
NULL	0xFFFFFFFF (illegal str)
"0b00"	0(valid 0)
"0b05100"	0xFFFFFFFF (illegal str with the char 5)
0b01	1
0b110	6

`int32_t int_to_binstr(char *str, size_t size, int32_t num, uint8_t nbits)`

Returns the binary representation of a signed integer, as a null-terminated string. On input, *str* is a pointer to a char array of at least *size* bytes, *num* is the value to be converted to a binary string, and *nbits* is the number of bits for representing the number *num*. The *nbits* representation does not include the "0b" prefix nor shall it include the null termination. See the examples below.

Basically, the parameter *nbits* can be any integer between 0 and 255 and you must check for illegal values of *nbits* and return error. *nbits* must be large enough to provide a correct representation of the number *num* in 2's complement.

If the operation was successful, the function returns the number of characters written to *str,* not including the terminal \0. Therefore, the return value will be *nbits* + 2(for 0b) if the conversion succeeded.

In the case of an error, the function returns a 0xFFFFFFFF, and *str* is set to the empty string "".

Examples:

num	**nbits**	**Str**	**Return value**
18	8	0b00010010	10
-1	4	0b1111	6
-3	8	0b11111101	10
-18	4	""	0xFFFFFFFF

```
uint32_t hexstr_to_uint(const char *str)
```

Returns an unsigned integer represented by the hex string input *str* which is null terminated. The hex string will be of the form '0xcccccc' where 'c' must be between '0' and 'F'. The casing of the characters in the string should not matter and you should be able to handle '0x0F' same as '0X0f'.

For a valid input, the number (or count) of hex digits 'c' must be between 1 and 8(why?). For any errors, the return value is 0xFFFFFFFF (which basically casts the maximum uint32 value as illegal). Be careful in catching the illegal characters in the string and missing '0x' at the beginning.

In the following representation, both lower case and upper case 'x' are allowed.

Examples:

str	**Return value**
0x12	18
0x0012	18
0XfF78	65400
0x0136	310

```
typedef enum {
	CLEAR,
	SET,
	TOGGLE
} operation_t;
```

```
uint32_t twiggle_except_bit(uint32_t input, int bit, operation_t operation)
```

Changes all bits of the *input* value except the given *bit*. Upon invocation, *bit* is in the range 0 to 31, inclusive. Returns 0xFFFFFFFF in the case of an error.

Examples:

input	**bit**	**operation**	**Return value(convert to uint from hex)**
0	0	SET	0xFFFFFFFE
0	3	SET	0xFFFFFFF7
0x7337	5	TOGGLE	0xFFFF8CE8
0xFFFF	31	CLEAR	0

`uint32_t grab_four_bits(uint32_t input, int start_bit)`

Returns four bits from the *input* value, shifted down. This function's output is best shown graphically. If *start_bit* is 20 and the 32-bit *input* is represented in binary as follows:

```
........ XXXX.... ........ ........
```

Then the output would be the four bits represented by the Xs, also in binary:

```
00000000 00000000 00000000 0000XXXX
```

Note that *start_bit* represents the least-significant of the four bits of interest.

Returns 0xFFFFFFFF on error.

Examples:

input	start_bit	Return value
0x7337	6	12
0x7337	7	6

`char *hexdump(char *str, size_t size, const void *loc, size_t nbytes)`

Returns a string representing a "dump" of the *nbytes* of memory starting at *loc.* Bytes are printed up to 8 bytes per line, separated by newlines, using the following format:

```
0x0000  53 70 61 63 65 3A 20 74
0x0008  20 66 72 6F 6E 74 69 65
0x0010  20 61 72 65 20 74 68 65
0x0018  20 6F 66 20 74 68 65 20
0x0020  20 45 6E 74 65 72 70 72
0x0028  20 66 69 76 65 2D 79 65
0x0030  6F 6E 3A 20 74 6F 20 65
0x0038  74 72 61 6E 67 65 20 6E
0x0040  73 2E 20 54 6F 20 73 65
0x0048  65 77 20 6C 69 66 65 20
0x0050  63 69 76 69 6C 69 7A 61
0x0058  6F 20 62 6F 6C 64 6C 79
0x0060  65 20 6E 6F 20 6D 61 6E
0x0068  65 20 62 65 66 6F 72
```

The first column shows the offset in bytes from *loc*, in hex. Next, up to 8 two-digit hex values are printed per line, representing the corresponding bytes of memory. After 8 values, a newline (\n) is inserted. *nbytes* must fit in 2 bytes i.e. is <= 65535 otherwise return an error.

The function returns the pointer *str*, which facilitates daisy-chaining this function into other string-manipulation functions such as puts. In the case of an error (for instance, because *str* is not large enough to hold the requested hex dump), *str* will be set to the empty string.

Implementation note: This is a good function to have in your toolbox—often, when debugging, you wish to examine some memory but do not have access to a debugger or printf. You should generate the entire string above through direct manipulation of the ASCII characters.

As an example, the following code:

```
const char *buf= \
  "To achieve great things, two things are needed:\n" \
  "a plan, and not quite enough time.";

char str[1024];
puts(hexdump(str, sizeof(str), buf, strlen(buf)+1));
```

creates this output:

```
0x0000  54 6F 20 61 63 68 69 65
0x0008  76 65 20 67 72 65 61 74
0x0010  20 74 68 69 6E 67 73 2C
0x0018  20 74 77 6F 20 74 68 69
0x0020  6E 67 73 20 61 72 65 20
0x0028  6E 65 65 64 65 64 3A 0A
0x0030  61 20 70 6C 61 6E 2C 20
0x0038  61 6E 64 20 6E 6F 74 20
0x0040  71 75 69 74 65 20 65 6E
0x0048  6F 75 67 68 20 74 69 6D
0x0050  65 2E 00
```

All functions should check for invalid inputs and return appropriate errors. All functions should start with a block comment describing the purpose of the function, the parameters, the return value, and anything special the caller should be aware of.

You should not use the division or modulus operators (`/` or `%`) in your production code. Instead, consider how you might achieve the same ends by using the bit-wise operators (`<<`, `>>`, `&` and `|`).

Each function should have a corresponding unit test function named by prepending "test_" to the name of the function being tested. (For example, the unit test for uint_to_binstr() would be called test_uint_to_binstr()). The unit test should thoroughly exercise the function being tested, passing a range of **valid and invalid** inputs, and validating that the return results are correct. Test functions should return 1 if all tests succeeded, and 0 otherwise.

In your unit tests, be devious: Try to ensure you have covered all the corner cases.

You may print any diagnostic output you find useful from the test functions, but your output should be sensible to another engineer. At a minimum, you should print diagnostic information about failed tests.

Your main() should call all the test functions.

Development Environment

We have set up a Linux cloud-based server named CUPES for you to use as a development environment for this assignment. (See the document "Using CUPES", available on Canvas, for details about how to access CUPES.)

For this project, may follow either of two development methodologies:

1. You may develop your code entirely on CUPES, connecting to it using SSH as described in the "Using CUPES" document and editing files on CUPES.

2. If you prefer, you may also develop your code to the nearly complete level using your own compiler and your own development machine. I recommend using GCC as your complier[1] to minimize problems later. Once you are nearly complete with the assignment, you MUST compile and test your submission on CUPES prior to committing the final version to GitHub.

To aid in your efforts, I have made an autograder available. Its use is completely optional, but it will help you understand how we will evaluate your project's quality while you still have time to improve it. From the CUPES command prompt, you can run the autograder by typing the command **`autograder1`** and specifying all the .c and .h files (if any) in your project. Here is a simple example, if all your code is in a single .c file:

```
autograder1 assignment1.c
```

Or, if your project uses three files:

```
autograder1 main.c functions.c functions.h
```

Your code should follow the ESE C Style Guide (posted on Canvas) as closely as possible.

When compiling with GCC, use the -Wall and -Werror compiler flags at a minimum. Your code must have no compilation errors or warnings.

I encourage you to use a Makefile to build your code, if you are familiar enough with make to do so. (However, to be clear, a Makefile is not required for this assignment and no points will be deducted if you do not use one. We will be covering Makefiles soon.)

---

[1] For those of you on Macs, note that Apple has their own compiler, Clang, which is a very good C compiler—but it isn't GCC. Confusingly, if you type **`gcc`** on a Mac, you will get Clang instead, which probably isn't what you want. You can tell what compiler you are actually running by typing **`gcc --version`**.

GitHub

Follow these steps to create your GitHub repository:

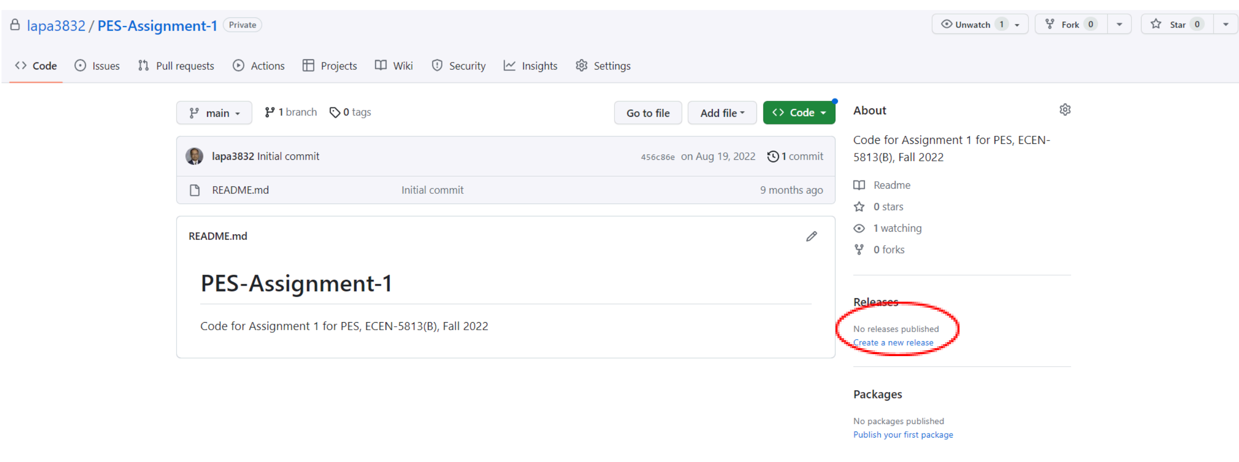
If you do not already have one, create a GitHub account at http://www.github.com.

Your are given a GitHub classroom link in the Canvas assignment. Once you click on the link and accept, it will setup your GitHub repository.

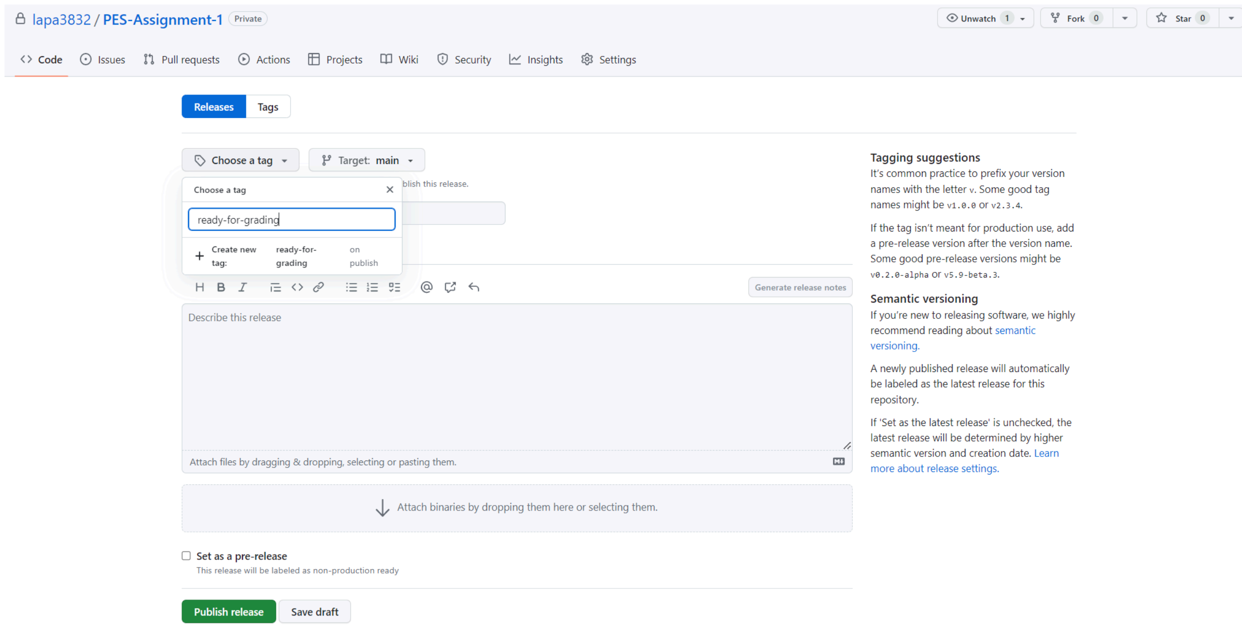
A video tutorial of using the GitHub is available on the Canvas course page. We may also setup SA sessions to help you with GitHub if needed. For now, you can use the GUI to add your .c file(s) and edit the README that GitHub created for you.

When you are ready to submit the assignment, you will need to create a tag named "**ready-for-grading**" as follows BEFORE THE ASSIGNMENT DUE DATE. SAs will be checking the date of this tag and code associated with this tag for grading. Please use the exact name of the tag with letter casing and dash in the middle.

To create a tag "ready-for-grading", you will need to create a new release. Press the "Create a new release" as illustrated in the following screen:



Press Choose a tag, enter the name of the tag "ready-for-grading", press "Create new tag" and then press "Publish release".



Grading

Points will be awarded as follows:

Points	Item
1	Did you submit the GitHub url on Canvas and create “ready-for-grading” tag? (All or nothing)
5	Does your code compile without warnings, and do you have all 7 required functions and all 7 required test functions? (Points are all-or-nothing here.)
70	Correctness of the result, based on running our automated tests. [10 points per function.]
5	Test coverage basics: Are your tests automated? (2) Do they output whether they passed or failed? (2) Is there other relevant diagnostic output? (1)
5	Test coverage: Do your unit tests do a good job of testing all the interesting combinations?
5	Bitwise operators: Do you use the bitwise operators instead of division and modulus? Did you implement it without using printfs or any string or other function library. (All or nothing).
5	Code legibility: Is your code indented reasonably and are variable names well selected? (3) Does your code follow the ESE style guide? (2)

4	Function documentation: Are there well-written comments documenting each function, including inputs and outputs?

Some advice: If you do a good job on your test cases, you will almost certainly get the code entirely correct, as well. Good luck!