1 Project 1 Bit Twiddling in C

Due: Sep 12 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

This document first provides the aims of this project. It then provides some background information. It then lists the requirements as explicitly as possible. It lists the files with which you have been provided and describes how those files are organized into modules. It then provides some hints as to how those requirements can be met. Finally, it describes how to submit the project for grading.

1.1 Aims

The aims of this project are as follows:

- To expose you to a typical Unix-based C development environment.
- To introduce the implementation of modules in C.
- To force you to understand the use of bitwise operations in C.

1.2 Background

Binary data is often transmitted serially, i.e. sequentially, bit-by-bit. Examples of serial communication protocols include most popular network protocols and I/O devices like USB.

When data is transmitted serially, reading it often involves segmenting the bit-stream into higher order entities like "words". This segmenting depends on the following parameters:

Number of bits within a word This parameter is used to find word boundaries within a bit stream.

Order of bytes within a word This parameter specifies the order in which the bytes which constitute a word are transmitted within a bit-stream. Common byte orders are most-significant byte to least-significant byte referred to as **big-endian**, and least-significant to most-significant, referred to as **little-endian**.

Order of bits within a byte This parameter specifies the order in which the bits which constitute a byte are transmitted within a bit-stream. Common bit orders are most-significant bit to least-significant bit referred to again

as **big-endian**, and least-significant bit to most-significant bit, referred to again as **little-endian**.

Disregarding second-order considerations, the values of these parameters can pretty much be chosen arbitrarily. However, in order to achieve correct communication, it is **imperative** that both sender and receiver agree on the values of these parameters.

This project uses a fixed big-endian order for the byte order of a word, and a fixed little-endian order for the bits within a byte, but allows the number of bits within a word to be specified at runtime.

1.3 Requirements

Submit a `submit/prj1-sol` folder in your `i220`*X* repository in github such that typing `make` within that folder produces a `bits-to-ints` executable within that directory with usage:

```
./bits-to-ints N_BITS [FILE...]
```

`N_BITS` should specify the number of bits in a word which should be a power-of-2 between 8 and 64 inclusive and `[FILE...]` specifies 0-or-more filenames.

Each specified file should contain the characters `'0'`, `'1'` or whitespace, where a whitespace character is any character `c` such that `isspace(c)` from `<ctype.h>` returns non-zero. The number of non-whitespace characters within each file should be an integral multiple of `N_BITS`; i.e., words cannot span multiple files.

The program should segment the bit stream contained in each file into an unsigned word of `N_BITS`, assuming that the bytes within a word are in big-endian order and the bits within a byte are in little-endian order. Whitespace characters should be ignored. It should print out the resulting value of each word in hexadecimal on standard output, one word per line, where each value is preceeded by the requisite number of `0` hexets needed to pad out the value to `N_BITS`.

The program should read from standard input if zero file names are specified on the command-line.

The program should print out error messages on standard error for the following error conditions:

- An invalid character which is not one of `'0'`, `'1'` or whitespace is encountered in one of the `[FILE...]` arguments.
- End-of-file is encountered within a word.

You are being provided with much of the necessary code. What you really need to provide is an implementation for this *specification header file*.

The behavior of the program is illustrated in the provided aux-files/test1.LOG log file.

1.4 Provided Files

The provided prj1-sol directory includes the following files:

bits-to-ints.c A skeleton file which you will need to modify.

bits-to-ints.h A specification header file which provides the specification you need to implement. You should not modify this file.

errors.c and errors.h Specification and implementation of a trivial error reporting module. It provides two functions `error()` and `fatal()` where each function takes `printf()`-style arguments and prints the message specified by its arguments on standard error; the difference between the two is that `fatal()` additionally terminates the program. You should not need to modify these files.

main.c This provides the necessary command-line behavior for the program. You should not need to modify this file.

Makefile A `Makefile` with default target which builds the entire program. It also provides a `clean` target for cleaning out object and executable files and emacs backup files. You should not need to modify this file.

test1.txt A file containing test data; one byte per line.

1.5 Modules

Modules are available in many programming languages to provide a unit of encapsulation larger than a function. The encapsulation serves to export some useful functionality to the rest of the program, while keeping the details of the implementation of that functionality hidden.

Unfortunately, C does not provide any language support for modules. Fortunately,\ is possible to implement modules in C by following some simple rules:

1. Provide a specification file usually named with a `.h` extension, which declares the functions and types exported by the module.
2. Provide an implementation file usually named with a `.c` extension, which implements the specifications. All implementation details should be hidden within this implementation file by being declared using a `static` modifier, thus restricting the visibilitity to within that file.
3. `#include` the `.h` specification file into the `.c` implementation file. Doing so guarantees that any syntactic incompatibility between the specification and the implementation is caught by the compiler.
4. `#include` the `.h` specification file into any client files which require the functionality exported by the module.
5. Link the implementation file into the executable.

The provided files provide an error-reporting module with specification file errors.h and implementation file errors.c, Similarly, the `bits-to-ints` module uses the bits-to-ints.h specification file and a bits-to-ints.c implementation file.

Note that it is possible to extend the above ideas for implementing modules in C to handle object encapsulation.

1.6 Hints

This project will require some C knowledge which has not yet been explicitly covered in class:

- One of the parameters to the `bits_to_ints()` function is declared `bool *isEof`, which means `isEof` is a pointer to a `bool`. We have not yet covered pointers in class, but assigning a value through this pointer is as simple as `*isEof = true;` or `*isEof = false;`.
- You will need to read characters from the input stream `inFile`. You can do so using the `<stdio.h>` function `fgetc(inFile)` which will return an `int` representing the ASCII code of the next character read from stream `inFile`. The return value is set to the special value `EOF` (also defined in `<stdio.h>`) on end-of-file.
- You can check if an integer `c` represents the code for a whitespace character using the `<ctype.h> isspace(c)` function.

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Review all relevant material, specifically the out_bits() function in the slides and the sections 2.1 *Information Storage* and 2.2 *Integer Representations* of the text.
2. Identify suitable abstractions for your problem:
   - You need to read a bit at a time from the `inFile` stream.
   - You need to assemble bits into bytes.
   - You need to assemble bytes into words.

   It is reasonable to create different functions for each of these abstractions.
3. Identify a strategy to assemble bits into a byte. Note that since the order of bits within a byte is little-endian, you will need to ensure that the bits are inserted into the byte LSB to MSB.
4. Identify a strategy to assemble bytes into a word. Note that since the order of bytes within a word is big-endian, this strategy turns out to be quite simple.
5. Come up with a way of handling terminating conditions:

- An invalid character within the bit-stream.
- An end-of-file is encountered after an integral number (possibly 0) of words have been read.
- An end-of-file is encountered while reading a byte.
- An end-of-file is encountered on a byte boundary but within a word.

You should try to do this without using global variables.

6. Get started on your project by copying over the provided prj1-sol directory to your `work` directory. Assuming that you have set up your course account as instructed, the following should work:

```
$ cd ~/i220?/work
$ cp -r ~/cs220/projects/prj1/prj1-sol .
$ cd prj1-sol
```

You can compile the project by simply typing `make`. You can even run it but it will not perform any useful functionality.

7. Implement the above strategies by making suitable changes to your copy of the `bits-to-ints.c` skeleton file. The amount of code involved should be minimal; around 100 lines or so.

8. Test and iterate the above steps until your project meets all the requirements. While working on the project be sure to `commit` and `push` your changes to git frequently in order to avoid loosing your work accidentally.

1.7 Submission

When you are happy with your project, move it over from your work directory to your submit directory:

```
$ cd ~/cs220X #X is either a or b
$ git mv work/prj1-sol submit
$ git commit -a -m 'suitable comment'
$ git push
```

This should submit your project as `submit/prj1-sol` via github. Your submission should not include any object files or executables; this will be prevented by the provided *.gitignore* file.

If submitting late, please drop an email to the TA for your section:

Section A yli241@binghamton.edu

Section B rrausha1@binghamton.edu