1 Project 2

Due: Oct 11, before midnight.

This document first provides the aims of this project, followed by a discussion of its background. It then lists the requirements as explicitly as possible. It then hints at how these requirements can be met. Finally, it describes how it can be submitted.

1.1 Aims

The aims of this project are as follows:

- To give you more exposure to bit twiddling.
- To give you some familiarity with writing code which uses pointers.
- To force you to write *portable code* which automatically adjusts itself to different size integers.
- To expose you to *conditional compilation* using the C preprocessor.
- To introduce you to *Test-Driven Development TDD*.

1.2 Background

The primary requirement for this project is to implement an integer representation called Binary Coded Decimal. While solving this problem you will get exposure to conditional compilation and test-driven development. This section provides some background material for these topics.

1.2.1 Binary Coded Decimal

The unsigned and 2's complement representations covered in class are only some of the many ways integers can be represented using bits. Another alternative is *binary-coded-decimal BCD*.

BCD representation is straightforward: each decimal digit is represented using a single *nybble* (4-bits) with the binary-encoding of the digit being the straightforward bit-encoding of that digit. Successive digits are encoded using successive nybble's.

Examples of encodings: (we show the BCD encoding in both hex and decimal; for interest, we also show a binary encoding).

```
-------------------------+-----------------+---------------
  Decimal Representation |  BCD Encoding   | Binary Encoding
-------------------------+-----------------+---------------
       798               |   0x798 (1944)  |        0x31e
       123               |   0x123 (291)   |         0x7b
     98765               | 0x98765 (624485)|      0x181cd
-------------------------+-----------------+---------------
```

1.2.2 Conditional Compilation

Recall that the C preprocessor is a text-to-text transformer which runs before compiler proper. It has no relationship to the compiler proper other than providing the input to the compiler proper. The preprocessor has various features, one of which is *conditional compilation*.

The preprocessor allows use of `#if` and `#endif` directives to determine whether code should be included in the compilation. Here is a simple example of using these directives to comment out code:

```
#if 0  //preprocessor directives must be on separate lines
  const char *str = "some code which is commented out";
#endif
```

Another example is to include a `main()` function used to test the code within a file `module.c`:

```
#if TEST
/** code for running an interactive test */
int main() { ... }
#endif
```

In this case, `TEST` is a preprocessor macro. These are usually defined within the source code using `#define` directives, but can also be defined at the compiler command-line using `-DTEST=0` or `-DTEST=1` command-line options. For example, a `Makefile` for `module.c` may include targets like the following:

```
OBJ_FILES = ... module.o ...

#link to produce main executable without test code
my-executable:   $(OBJ_FILES)
                 $(CC) $? $(LDFLAGS) -o $@

#module.o built using a normal implicit compilation command like
```

```
#
#$(CC) $(CFLAGS) -c module.c;
#
#since TEST is not defined, it defaults to 0
#and no test code is compiled in.
module.o:       module.c module.h

#this builds a module-test executable.
#since TEST defined as 1, test code is compiled in.
module-test:    module.c module.h
                $(CC) $(CFLAGS) -DTEST=1 $< $(LDFLAGS) -o $@
```

With the above `Makefile`, when `module.o` is linked into `my-executable`, `TEST` is not defined and hence the testing `main()` is not included within `my-executable`. However, if `module-test` is built using a command like `make module-test`, then the compiler command-line option `-DTEST=1` sets `TEST` to 1 and the testing `main()` will be included within the compilation.

This project uses conditional compilation to define different C integer types for `Bcd`:

```
/** Representation used for a BCD number: define based on BCD_BASE */
#ifndef BCD_BASE
  #define BCD_BASE 2
#endif

//Depending on BCD_BASE, conditionally define:
//   Bcd:                    C type used to represent a BCD
//   BCD_FORMAT_MODIFIER: modifier used to printf() a BCD
//   SCANF_MODIFIER:         modifier used to scanf() a BCD
#if BCD_BASE == 0
  typedef unsigned char Bcd;
  #define BCD_FORMAT_MODIFIER ""
  #define SCANF_MODIFIER "hh"
#elif BCD_BASE == 1
  typedef unsigned short Bcd;
  #define BCD_FORMAT_MODIFIER ""
  #define SCANF_MODIFIER "h"
#elif BCD_BASE == 2
  typedef unsigned Bcd;
  #define BCD_FORMAT_MODIFIER ""
  #define SCANF_MODIFIER BCD_FORMAT_MODIFIER
#elif BCD_BASE == 3
  typedef unsigned long Bcd;
  #define BCD_FORMAT_MODIFIER "l"
```

```
  #define SCANF_MODIFIER BCD_FORMAT_MODIFIER
#elif BCD_BASE == 4
  typedef unsigned long long Bcd;
  #define BCD_FORMAT_MODIFIER "ll"
  #define SCANF_MODIFIER BCD_FORMAT_MODIFIER
#else
  #error "invalid BCD_BASE value"
  //avoid subsequent compiler errors by providing definitions
  typedef unsigned long long Bcd;
  #define BCD_FORMAT_MODIFIER "ll"
  #define SCANF_MODIFIER BCD_FORMAT_MODIFIER
#endif
```

The above code **typedef**'s `Bcd` to different C integer types as well as defining the appropriate modifiers which can be used for printing or reading a `Bcd`. The `Bcd` type chosen depends on the value of `BCD_BASE` which can be specified on the compiler command-line using `-DBCD_BASE=...`; if `BCD_BASE` is not provided on the compiler command-line, then the initial `#ifdef` section will default it to 2 (with `Bcd` becoming an **unsigned**).

1.2.3 Test-Driven Development

Automated testing is an integral part of modern software development practices. In **test-driven development**, tests are written before the code which is to be tested. In this project, you are being provided with a set of tests and you will write code to make the tests pass; i.e. your development will be driven by the tests.

There are multiple testing frameworks available for different programming languages. This project uses the check testing framework for C.

1.3 Requirements

Update the `master` branch of your github repository with a directory `submit`¬ `/prj2-sol` such that that typing `make` within that directory will build executables `bcd-0`, `bcd-1`, `bcd-2`, `bcd-3` and `bcd-4` which correspond respectively to using **unsigned char**, **unsignedshort**, **unsigned int**, **unsigned long** and **unsigned long long** to represent a BCD. Additionally, `bcd` should be symlinked as `bcd-2`.

When any of the above programs is invoked, it must read lines from standard input.

- If the line contains a single non-negative decimal integer, then the program must print a line containing the string representation of the BCD

representation of that integer, followed by the decimal value of that BCD representation in parentheses.

- If the line contains 2 non-negative decimal integers separated by a `+` character and the sum of the two integers is `sum`, then the program must print a line containing the string representation of the BCD representation of `sum`, followed by the decimal value of that BCD representation in parentheses.
- If the line contains 2 non-negative decimal integers separated by a `*` character and the product of the two integers is `prod`, then the program must print a line containing the string representation of the BCD representation of `prod`, followed by the decimal value of that BCD representation in parentheses.
- If the contents of the line does not match any of the above, then the program should signal an appropriate error and continue processing its standard input.

Errors which must be signalled include:

- **Bad BCD Digit**: A nybble within a BCD is greater than 9.
- **Overflow**: A value is too large to be represented within a BCD type.

The above program **must** be implemented using a `bcd` module which meets the specifications provided in bcd.h.

You may assume that the character encoding for the digit-characters is such that the value of the digit represented by digit-character `c` is `c - '0'`.

The operation of the program is illustrated in run.LOG. Testing behavior is illustrated in check.LOG.

1.4 Provided Files

The prj2-sol directory contains the following files:

bcd.c The main file which you will be modifying. It contains `@TODO` comments which you should replace with your code.

bcd.h The specification for the BCD functions you will be implementing. You should not modify this file.

bcd-test.c A file which contains test code.

check-extra.h Supplements `check` with testing macros for **unsigned long** and **unsigned long long**.

main.c The command-line interface. You should not need to modify this file.

Makefile A non-trivial Makefile. Simply typing `make` runs the default targets which builds `bcd` as well as `bcd-0`, ..., `bcd-4`.

Running `make check` runs all tests for all of `bcd-0`, ..., `bcd-4`. You can selectively run tests for different functions by typing `make check TEST=`*test* for *test* in `binary_to_bcd_test`, `bcd_to_binary_test`, `str_to_bcd_test`, `bcd_to_str_test`, `bcd_add_test`, `bcd_multiply_test` and `bcd_multiop_test` all of which run tests only for the corresponding BCD function except `bcd_multiop_test` which runs multiple BCD functions.

If a test is failing, you can look at the code in bcd-test.c to understand why it may be failing. However, you can also add `BCD_TEST_TRACE=1` to the `make check` command which shows the actual and expected output for each test which is run.

README A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project. If your project is not complete, document what is not working in the `README`.

.gitignore A file which lists paths which should be ignored by git. You should not need to modify this file.

The extras directory contains the following files:

run.LOG A log of the command-line behavior.

check.LOG A log of running tests.

`bcd`, `bcd-0`, ..., `bcd-4` Executables which you can use to validate your understanding of the project requirements.

1.5 Hints

Since you are required to have your code work with different underlying C types mapped to the `Bcd` type, it is imperative that your code not make any assumptions on the size of the `Bcd` type. Instead your code should use the constants `BCD_BITS`, `MAX_BCD_DIGITS` and `BCD_BUF_SIZE` defined in bcd.h.

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

1. Make sure you understand BCD representation. It may also be a good idea to review elementary school methods for addition and long multiplication.
2. Start your project in a manner similar to how you start a lab.
   (a) Ensure that your local copy of the `cs220` course repository is up-to-date:

```
$ cd ~/cs220
```

```
$ git pull
```

(b) Create a branch for this project in your working copy of your github respository:

```
$ cd ~/i220?            #go to clone of github repo
$ git checkout master   #ensure in master branch
$ git pull              #ensure master up-to-date
$ git checkout -b prj2  #make new branch
```

(c) Copy over the provided files and commit them to github:

```
$ cd ~/i220?/submit
$ cp -pr ~/cs220/projects/prj2/prj2-sol .
$ cd prj2-sol
$ git add .
$ git commit -m 'started prj2'
$ git push -u origin prj2  #push prj2 branch to github
```

3. Fill in your details in the `README` template. Commit and push your changes.

4. Verify that you can build the executables using the provided `Makefile` by simply typing `make`.

5. Verify that you can run tests by typing `make check`. At this stage, most of the tests will fail (some pass purely coincidentally).

6. Open up your copy of `bcd.c` in a text editor. It may be a good idea to define utility routines like `get_bcd_digit()` and `set_bcd_digit()` with suitable parameters.

7. Define the `binary_to_bcd()` routine. Notice that you can iterate through the digits of a binary `value` by using `value%10` to extract the digit and `value /= 10` to "shift" the value over by one digit. Once you have the digit, it needs to be inserted into the correct position into a `Bcd` accumulator.

   Verify your `binary_to_bcd()` code by running:

   ```
   $ make check TEST=binary_to_bcd_test
   ```

   If you would like information about the tests being run, use:

   ```
   $ make check TEST=binary_to_bcd_test BCD_TEST_TRACE=1
   ```

8. Define the `bcd_to_binary()` routine. You can use your `get_bcd_digit¬ ()` function to extract a BCD digit. Once you have it, ensure that it is less than 10, otherwise signal a `BAD_VALUE_ERR` error. Accumulate it into a binary number, "shifting" it for each digit.

   Test using `make check TEST=bcd_to_binary_test`.

9. Define the `str_to_bcd()` routine. For each digit-character `d`, get its value using `d - '0'` and accumulate the value in a `Bcd`. If the number of digits turns out to be greater than `MAX_BCD_DIGITS` return an overflow error. Make sure that the pointer argument to the routine is updated to point to the first non-digit-character.

   Test using `make check TEST=str_to_bcd_test`.

10. Define the `bcd_to_str()` routine. You should first check whether the provided buffer size is at least `BCD_BUF_SIZE`, otherwise report an `OVERFLOW_ERR` error. After ensuring that all the BCD digits are valid (less than 10), the guts of the functionality can be achieved by simply using snprintf() with the `%x` format-specifier.

    Test using `make check TEST=bcd_to_str_test`.

11. Write the `bcd_add()` routine. Use the addition algorithm you learnt in elementary school: add digit by digit, propagating a carry if necessary. If the binary sum of 2 BCD digits and current carry is `sum`, you can get the resulting BCD digit as `sum % 10` and the propagated carry as `sum / 10`. You will have an overflow if the carry is non-zero from the most significant BCD addition.

    Test using `make check TEST=bcd_add_test`.

12. In preparation for the multiplication, it may be a good idea to write a routine which multiplies a BCD representation by a single BCD digit (in [0, 9]):

```
/** Return Bcd representation of result of multiplying Bcd
 *  value multiplicand by BCD digit bcdDigit (in [0, 9]).
 *
 *  If error is not NULL, sets *error to BAD_VALUE_ERR if
 *  multiplicand contains a BCD digit which is greater than 9,
 *  OVERFLOW_ERR on overflow, otherwise *error is unchanged.
 */
static Bcd
bcd_multiply_digit(Bcd multiplicand, unsigned bcdDigit,
                   BcdError *error);
```

13. Implement `bcd_multiply()`. Use the long-multiplication algorithm you learnt in elementary school. Initialize an accumulating result to 0. For

each digit in the multiplier, multiply the multiplicand by it (using `bcd_multiply_digit()`) and then add the result (suitably shifted, using `bcd_add()`) to the accumulating result.

Test using `make check TEST=bcd_multiply_test`.

14. Iterate the previous steps until you meet all the project requirements and all tests pass.

1.6 Submission

Submit using a procedure similar to that used in your labs:

```
$ cd ~/i220X
$ git branch -l        #list all branches;
                        # current branch has *, should be prj2.
$ git checkout master  #goto master branch
$ git pull             # pull changes (if any)
$ git checkout prj2    #back to prj2 branch
$ git merge -m 'merged master' master  # may not do anything
$ git status -s        #should show any non-committed changes
$ git commit -a -m 'completing prj2'
$ git push             #push prj2 branch to github
$ git checkout master  #switch to master branch
$ git merge prj2 -m 'merged prj2'  #merge in prj2 branch
$ git push             #submit project
```