Introduction to Electrical and Computer Engineering Design ECEG 201

Session 14 Lecture Notes

Announcements

1. Contact me ASAP if you have trouble reading my email attachments or accessing the course web page

   Homework 8 is being sent to you as a fillable pdf form. You should be able to just fill in your answers and send it back to me…no printing or scanning required.

2. The Moodle gradebook has been updated with all graded work that I received before spring break. Let me know if you find any missing scores or if you have questions.

3. Reading:

   (a) Revised syllabus for remote instruction

   - Course content will be delivered asynchronously, either by email or through the course web page.
   - Collaboration **is not allowed** on any homework, laboratory exercise, project work, quizzes, or exams for the rest of the semester.
   - Your work will generally be submitted via email. Written work must be submitted as a pdf file (not a jpg or a word processor file). Code will be submitted as an email attachment…do **not** copy your code into another document or file type.
   - You must check in at least twice each week. The preferred method will be by email.
   - Check your Bucknell email **every day** for changes to course policies.

4. Homework 8 is due by *noon* on 2020-03-25

My Solutions For Homework 7

Many of you did much more work than was necessary to write the `build_message()` function. Remember that the data in the processor is **always** a binary value; there is never any need to convert a value to a string of 0 and 1 **characters** in order to work with its binary value. If you calculated the length of the second byte string and then converted that length integer into some kind of string then you need to think about how numbers exist inside the processor. There is no need to convert a value into a string or to print the value in order to make it exist.

The tricky part was separating the integer length value into two separate integers representing the top and bottom bytes of the length value. This is essentially identical to taking a 2-digit decimal number and separating it into its ten's digit and its one's digit. We can do that with two simple calculations in normal arithmetic:

$$TensDigit = \lfloor OriginalValue/10 \rfloor \qquad OnesDigit = OriginalValue \mod 10$$

Since we want to separate the original value into two **bytes** instead of two decimal digits, we need to divide the original value by 256 instead of 10. We want an integer result so we use Python's `//` operator. Remember that this operator discards the fractional part of the quotient, which is equivalent to the floor operation.

The modulo operation does the opposite of the integer divide: it performs the division but keeps only the integer remainder. In Python the percent sign is used for the modulo operation so we could calculate the value of the bottom byte with something like `low_byte = length % 256`.

However, division is expensive on a little microcontroller, and I've done this enough to know that performing the modulo operation with a value that is an integer power of 2 is equivalent to a **masking** operation. If you want to perform a $\mod 2^N$ operation you can do this easily by keeping just the bottom N bits. In Python we can do this masking operation by ANDing the original value with a constant that has its bottom N bits equal to 1 and all other bits equal to 0. So, $X \mod 256$ becomes `X & 0xFF`.

In fact, we can avoid division altogether if we want. Performing an integer division by an integer power of 2 is equivalent to a right-shift operation. To divide by 2^N you just need to shift right by N bits. So, we could say `high_byte = length >> 8` in Python. Note that the shift operators always shift by the specified number of **bit positions**…the processor doesn't know or care anything about decimal or hexadecimal.

Now that we have split the length value into two bytes, we need to convert those two Python **integer** values into two Python **byte** values. Note that we are not actually changing the value of anything, we are changing the **type**. Python uses the type of a variable to indicate what can be done with that variable. Since we want to construct and return a textbfbyte string we must convert the two 8-bit integers into something that Python recognizes as a **byte character**. The method I told you about in class was to use the `chr()` function to convert each integer to a single-character string, then use the `chr().encode('latin-1')` method to convert the Unicode characters to 8-bit characters.

There were many reasonable ways to write the function, but my version of `build_message()` looks like this:

```
def build_message(string1, string2):
    """Concatenate strings, insert length of string2 in middle."""
    high_length_byte = len(string2) // 256
    low_length_byte = len(string2) & 0xFF
    return (string1 + chr(high_length_byte).encode('latin-1')
            + chr(low_length_byte).encode('latin-1') + string2)
```

Note that I provided a **docstring** for my function…that's the comment that is the first statement in the function. You should always provide a docstring with your functions. See Pythons PEP 257 for docstring conventions.

I also used the lowercase_with_underscores naming style for my variables and functions, as you should. I chose descriptive names for the variables, which is much better than using cryptic names and adding comments.

I wrapped the last line of the function so it would not be more than 79 characters long. In Python, you can always wrap in the middle of a parenthetical expression. The `return` doesn't actually need its expression to be in parentheses, but adding them made it easier to wrap. Note also that when I wrapped the line I wrapped **before** the binary string concatenation operator. That's the style recommended in PEP 8.

For the `count_substrings()` function you needed to find a method that you could use on byte strings to look for matches. There was **no need** to convert the byte strings to conventional Unicode text strings; the byte string data type has the needed method. I used the `find()` method, which returns the index of the first match if the substring matches in the bigger string, or returns a -1 if there was no match. So, I kept looking for a match until I got the -1. After each match was found I started the next search just one character beyond where the previous match was found.

```
def count_substrings(string1, string2):
    """Count occurrences of string1 in string2, with overlap."""
    first_char = 0
    occurrences = 0
    while string2.find(string1, first_char) >= 0:
        occurrences = occurrences + 1
        # Start looking just beyond the match we just found
        first_char = string2.find(string1, first_char) + 1
    return occurrences
```

Homework 7 Grading

There were 70 points possible on Homework 7.

- 10 points each for the 6 tests

  There were 6 example invocations of the functions provided in the homework handout. If your code executed all of these examples and produced the same output that my code did then you received 60 points. If your function produced something, but not the correct response, then you received partial credit for that example.

- 5 points miscellaneous problems

  You could lose up to 5 points for miscellaneous issues, such as not submitting your code as requested.

- 5 points from pylint score

  I ran everyone's code through pylint, as you should do, and awarded points based on your pylint score. A score of 8.00 or higher earned all 5 points, 6.00 or higher earned 4 points, 4.00 or higher earned 3 points, 2.00 or higher earned 2 points,

and 0.00 or higher earned 1 point. You didn't get any points if your pylint score was negative or you didn't submit your code properly.

Homework/Lab Feedback

Although I did not grade the Lab 4 reports heavily on the basis of style, there were some common problems that I wanted to address.

- Capitalize proper nouns. (Feather, CircuitPython, Adafruit)

  This is very important. If you are making a pitch to Adafruit you had bloody well better capitalize the name of the company properly. Many of these words are registered trademarks and should be capitalized. Sometimes capitalization helps convey meaning…if you tell me you spent the day trying to program a feather I will wonder if you have been following chickens around the yard; tell me you are programming a Feather and I know what you mean.

- Don't capitalize for emphasis or at random.

  This is just annoying and makes your writing hard to read. Don't capitalize words just to give them emphasis, so don't write "The Maximum Average was…", just say "The maximum average…".

- Use proper format for variable names and mathematical operators.

  Remember that names of **quantities** must always be written in italic: V_{DD}, D_{AVG}, T_C. This is true whether you are writing an equation or just using the names in a sentence. Use the correct symbols for mathematical operators, particularly the multiplication ($\times$) operator. If a quantity name needs a subscript then make it a proper subscript and be consistent with the format of the name.

  If you are talking about the maximum voltage or the average current, call them V_{max} and I_{AVG}, not $MAXV$ and $AVGI$. The first character should indicate the general kind of quantity you are talking about, then use a subscript to refer to a specific quantity of that kind.

  However, the symbols for the SI units and prefixes are **never** written in italic; they must always be in an upright font. $V_{BAT} = 9\,\mathrm{mV}$

- "Digital voltage" and "analog integer" are oxymora.

  Voltage is analog and integers are digital. Always. You can say "the integer value provided by the ADC for a particular voltage" but remember that everything you manipulate with code is digital.

  For Lab 4 you were asked to observe the digital integer value coming directly from the ADC. If you converted this value to a voltage you introduced numerical error, and if you then converted that floating-point value back to an integer you accumulated even more error. It is important for you to understand that you can evaluate raw numerical values without relating them to a physical quantity in the real world. The ADC output doesn't become more "real" if you calculate the ideal corresponding voltage.

- Do not conflate V_{REF} and V_{IN} for ADCs.

  The reference voltage, V_{REF}, for an ADC or DAC is essentially fixed and determines the range of voltages that can be converted. For an ADC, V_{IN} is the input voltage, which may change frequently. For proper operation, $V_{IN} < V_{REF}$.