CMSC 313 — Spring 2017

Project 2 — Escape Sequence Handling with Subroutines and Tables

Assigned	Thursday, March 2nd
Program Due	Wednesday, March 8th by 11:59pm
Updates:	None yet.

Objectives

The objectives of the programming assignment are 1) to gain experience writing larger assembly language programs, 2) to gain further experience with branching operations, 3) to learn how to make simple subroutine calls, and 4) to practice some indexed addressing modes

Background

String constants in C/C++ are allowed to contain control characters and other hard-to-type characters. The most familiar of these is "\n" for a newline or linefeed character (ASCII code 10). The "\n" is called an escape sequence. For this project, we will consider the following escape sequences:

Sequence	**Name**	**ASCII code**
\a	alert(bell)	07
\b	backspace	08
\t	horizontal tab	09
\n	newline	10
\v	vertical tab	11
\f	formfeed	12
\r	carriage return	13
\\	backslash	92

In addition, strings can have octal escape sequences. An octal escape sequence is a '\' followed by one, two or three octal digits. For example, "\7" (or "\07", or "\007") would be another way to embed an alert bell (ASCII code 7) into a string (we could also have used "\a"); similarly, "\134" is equivalent to "\\" (since $134_8 = 92_{10}$). You can even encode "ABC" as "\101\102\103", although that would be a silly way to do it. Note that in this scheme, the null character can be represented as "\0". The octal escape sequence ends at either the third octal digit, the end of the string, or before the first non-octal digit, whichever comes first. For example "abc\439xyz" is equivalent to "abc#9xyz" because 9 is not an octal digit, so the octal escape sequence would only include "\43", and 43_8 is the ASCII code for '#'.

Assignment

For this project, you will write a program in assembly language which takes a string input by the user, convert the escape sequences in the string as described above and print out the converted string. So, if the user entered "Hello\nbye\a\n", it would print out:

```
Hello
bye
```

as well as sounding a beep. In addition, your program should be robust enough to handle user input that might include malformed escape sequences. Examples of malformed escape sequences include: a '\' followed by an invalid character, a '\' as the last character of the string, and a '\' followed an octal number that exceeds 255_{10}.

All the invalid escape sequences should be reported to the user (i.e., your program should not just quit after detecting the first invalid escape sequence). When the user input has malformed escape sequences, your program should still continue and convert the rest of the string (which might contain additional valid escape sequences). In the case of a malformed escape sequence, a '\' should be printed at the location. For example, if the user types in "abc \A def \43 ghi \411" your program should have output:

```
Error: unknown escape sequence \A
Error: octal value overflow!
Original: abc \A def \43 ghi \411
Convert: abc \ def # ghi \
```

Additional Requirements

There are several important additional requirements for the project:

1. You must manually null-terminate the input string that is returned by the read syscall: recall that the read syscall does not do that for you, but the `handle_ESC` subroutine (described next) requires it. This is quite simple to do: the read syscall does tell you how many characters it read, so you just need to do the assembly language equivalent of "buf[rlen] = 0". Note that this requires that you have room in your buffer for one additional byte, so make sure your read request is at least one less than the actual size of your buffer.

2. You must put the code for handling escape sequences into a separate subroutine, labelled `handle_ESC`. This subroutine will be called every time the main program loop hits a '\' in the input string. `handle_ESC` takes one parameter as input: it expects the register ESI to contain a pointer to (i.e., hold the address of) the first byte of the escape sequence *after* the '\'. The source string is guaranteed to be null-terminated (note the first requirement above), so you do not need to be passed the string length as an additional parameter. This subroutine must process the pointed-to part of the source string to parse the complete escape sequence, which might be more than one character for octal numbers. It must then return the character the escape sequence maps to, stored in the register EAX. If it finds a malformed escape sequence as defined earlier, it should print out the appropriate error message, and then return '\' in EAX. It must also have updated ESI to now point to the first character after the complete escape sequence.

3. You must call `handle_ESC` from your main source string-handling loop whenever you detect a '\' in the source string. ESI must be pointing to the character *after* the '\'. (See "Hints" below for a comment about using ESI here.) The main loop should then take the replacement character that `handle_ESC` returns in EAX and copy that to the output string. For all non-escape-sequence input characters, the main loop itself should simply copy it to the output string.

   Note that `handle_ESC` might end up processing more than one character in the case of octal codes, resulting in the pointer in ESI having moved forward multiple positions.

4. The subroutine `handle_ESC` must use a lookup table to map the alphabetic escape characters (a, b, t, n, v, f, and r) to their resulting translated characters. (Octal sequences, as well as "\\" are handled separately, not using the table.) The table must have no more than 26 entries. So, if you are looking for the translation for "\a", handle_ESC would be called with ESI pointing to the 'a'. It would figure out that 'a' is a lowercase alphabetic character, and index into the table (array) at position 0 (since it is the first alphabetic character), where it should find the value 7 (since "\a" should map to ASCII code 7).

5. When accessing this lookup table, you ***must*** use one of the indexed addressing modes, one that uses at least two of the base, index, and displacement fields in the operand specification, to directly index into the appropriate location.

6. Your program must exit cleanly using the exit system call.

Sample Run

Your program input/output should look exactly like the following (user input is underlined):

```
% ./escapeseqs.out
Enter string: Yowza\nThis is exciting\a\7\007!
Original: Yowza\nThis is exciting\a\7\007!
Convert:  Yowza
This is exciting!
%
% ./escapeseqs.out
Enter string: This is weird...\rYOWZA!!!! THAT\12
Original: This is weird...\rYOWZA!!!! THAT\12
YOWZA!!!! THAT is weird...
%
```

In the first example, you would hear 3 beeps (possibly blurred into one longer beep). In the second, notice that the "Convert: " label is gone from the output--why?

Examples of errors were given earlier.

Implementation Notes/Hints

- As for Project 1, you should start from the source for toupper.asm. However, you will have to restructure the main loop to handle the more complex character conversion involved here, and to also call the `handle_ESC` subroutine when needed. (Look back at the Project 1 description if you don't remember how to download a copy of toupper.asm.)

- Because the subroutine `handle_ESC` might process more than one character, it becomes very difficult to use a simple loop counter in the main routine as toupper does. Luckily, `toupper` also maintains a pointer to the character it is currently converting. You will need to modify the loop structure to, instead of using the counter to decide when to stop, instead stop when it reaches a null character.

- Your program will have numerous branches. You should think about the layout of your program and how to make it more readable. Avoid spaghetti code. Related parts of your program should be placed near each other. You will be graded on clarity and structure.

- Note that if there were any escape sequences, the output string will be shorter than the input string, so do not just use the read length (rlen) as the write length.

- ASCII is actually only a 7-bit code (it only uses the values 0-127), so if you use 8-bit signed numbers in your lookup table, you can put in negative values as special flags; we recommended that you create a table with -1's in all the cells that are not legal escaped characters, making it easy to detect illegal alphabetic escape sequences (e.g., "\q").

- Note that the lookup table only has 26 entries, so you cannot directly index into the table with the character's actual ASCII code. For example, 'a's ASCII code is 97, which is outside the table; you must convert 'a' to index 0, 'b' to 1, ..., 'z' to 25.

- The following is a suggested pseudocode structure for your `handle_ESC` subroutine. You should flesh it out into something more like real C code before embarking on writing your assembly language version.

```
handle_ESC() {
    c = *ESI;   // Get a temporary copy of the next SINGLE character
    ++ESI;

    if (c == SOME OCTAL DIGIT) {
        // NB: max value of 3-digit octal might not fit in a byte
        total_value = (CONVERT ASCII CHAR c TO INTEGER VALUE);
        for (i = 0; i < 2; i++) { // process up to 2 more octal chars
            c = *ESI;   // peek ahead at next character
            if (c != OCTAL_DIGIT)
                break;
            ++ESI;
            digit_value = (CONVERT ASCII CHAR c TO INTEGER VALUE);
              // You can do following with MUL, or something easier
            total_value = (total_value * 8) + digit_value;

        }
        if (total_value TOO BIG) {
            OUTPUT ERROR MSG;  // "octal value overflow!"
            value = '\';  // for error!
        }
        else {
            value = total_value;
        }
    }
```

```
        else if (c IS LOWERCASE LETTER) {
            // Project requirement: do not use a loop to search the table--
            value = LOOK UP c IN TABLE USING AN INDEXED ADDRESSING MODE;
            if (value == -1) {
                OUTPUT ERROR MSG; // "Unknown escape sequence \X"
                value = '\';  // for error!
            }
        }
        else if (c == '\') {
            value = '\';  // but this is not an error :-)
        }
        else {
            OUTPUT ERROR MSG;  // "Unknown escape sequence \X"
            value = '\';  // for error!
        }

        RETURN value;
    }
```

What to Submit

Before you submit your program, record some sample runs of your program using the UNIX script command. You should select sample runs that demonstrate the features supported by your program. Picking good test cases is your responsibility.

Use the UNIX `submit` command on the GL system to turn in your project. You should submit two files: 1) your assembly language program, named as `escapeseqs.asm`, and 2) the typescript file of your sample runs. The UNIX command to do this should look something like:

```
submit cs313_park proj2 escapeseqs.asm typescript
```