ME 477 Embedded Computing
Laboratory Experiment #2
A getchar_keypad for our application

Objectives

In this exercise you will gain experience with:

1. Code requirements for character I/O of a custom embedded computing application.
2. On-line debugging techniques.

Introduction

In Lab #1 you implemented a general-purpose function `double_in()` that prompts the user to enter a floating-point value on the keypad, and returns the result to the calling program. That function calls the C functions `printf_lcd()` and `fgets_keypad()`. These functions, in turn, call other lower-level C library functions according to the following hierarchy:

```
double_in()               -user double entry
  |_ printf_lcd()         -general display
  |      |_ putchar_lcd() -one char. to output
  |      |_ vsnprintf()   -print to string
  |
  |_ fgets_keypad()       -acquire string of chars
  |   |_getchar_keypad()  -one char. from input
  |      |_ putchar_lcd() -one char. to output
  |      |_ getkey()      -get 1 char from keypad
  |
  |_ sscanf()             -ascii-to-binary conversion
  |_ strstr()             -find string in string
  |_ strpbrk()            -find member in string
```

Continuing down the hierarchy, in this week's lab you will write the `getchar_keypad()` function. This function acquires a single character from the keypad. It must function identically to the standard C function `getchar()` that performs the same operations for the standard I/O device (the IDE console). You should review the `getchar()` functions in your C textbook.

Next week, you will write the lowest level `putchar_lcd()` and `getkey()` functions.

Pre-Laboratory Preparation

The prototype of the `getchar_keypad()` function is,

```
int getchar_keypad(void)
```

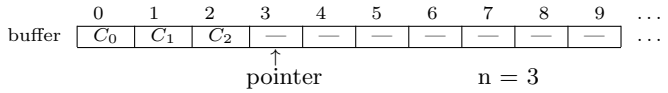
Each time `getchar_keypad()` is called it returns a single character from the keypad; and it returns EOF (defined in stdio.h) when `ENTR` is pressed. In the example below `getchar_keypad()` is used to obtain a string of characters until EOF is reached. The characters are stored sequentially in a buffer.

```
point = buffer;
while ( (ch=getchar_keypad()) != EOF ) {
    *point++ = ch;
}
```

There are two types of `getchar` functions that are useful in C. The first type, called an unbuffered `getchar`, simply returns the character to the calling program immediately after each keystroke. The second type, called a buffered `getchar`, collects the characters entered by the user in a temporary buffer. Pressing `ENTR` causes the block of characters to be made available to the calling program. You will write a buffered `getchar_keypad()` for the keypad.

The advantage of the buffered `getchar` is that the user can edit the characters in the buffer using the `Delete key` in the usual manner, before they are sent to the calling program. There is no possibility of editing with the unbuffered `getchar`.

You might wonder how a function that is designed to return only a single character could edit the whole buffer. This is accomplished by a simple and elegant means inside `getchar_keypad()`. The key idea is to use a statically declared character buffer. In that way, the characters remain in the buffer in between calls to `getchar_keypad()`. You will also need to statically declare a pointer to the buffer, and a variable (e.g. n) to keep count of the number of characters in the buffer.



Here's how the buffering scheme works:

Whenever `getchar_keypad()` is called there are two possible conditions of the buffer: either the buffer is empty, or the buffer contains one or more characters.

Initially, suppose that the buffer is empty when `getchar_keypad()` is called (n = 0). That is, the count is zero, and the pointer is at the beginning or the buffer. The function enters a loop, filling the buffer and displaying the characters, one keystroke at a time, until the `ENTR` key is pressed. Each time through the loop:

First, check if the buffer is full. If its not, then ...

1. The current character is entered into the buffer at the position determined by the pointer.
2. The pointer is incremented.
3. The character count is incremented, and
4. The character is displayed on the LCD.

After `ENTR` is pressed, the buffer pointer is set back to the beginning of the buffer, and the first character (alone) is returned to the calling program.

On subsequent calls to `getchar_keypad()` the buffer is not empty. For each call, the pointer is incremented, the count is decremented, and the character pointed to is

returned to the calling program. This continues until the last character in the buffer is returned, and the pointer is returned to the beginning of the buffer. Once the buffer is empty, the next call to `getchar_keypad()` begins the filling process again. Note: `getchar_keypad()` returns `EOF` in place of the `ENTR` key.

Putting these ideas together, the *pseudo code* (so far) for a buffered `getchar_keypad()` might look like this:

```
n       is  the number of characters in
        the buffer.
buffer  is a character array, of length buf_len + 2,
        used to hold the characters
pointer points to the location in the buffer
        where the next character will be put
        or taken.

if n is zero {
  -set the pointer to the start of the buffer
  while the character from keypad is not ENTR {
    if n < buf_len {
      -put character in buffer at pointer location
      -increment the pointer
      -increment n
      -put the character on the display
    }
  }
  -increment n
  -set the pointer to the start of the buffer
}

if n is greater than 1 {
  -decrement n
  -return the character pointed to &
           increment the pointer
}  else {
  -decrement n
  -return EOF
}
```

Now, suppose that the `Delete key` is pressed while characters are being entered. The deleted character is effectively "removed" from the buffer by decrementing both the buffer pointer and the variable containing the number of characters entered. The deleted character is removed from the display by moving the cursor left one space, sending a space, and moving the cursor left one space again. What should happen if `Delete key` is pressed before any characters have been entered (n = 0)? Modify the pseudo code above (and your program) to include the Delete function.

Main Function: Write a main function that tests your version of `getchar_keypad()`. It should collect at least two separate strings using `fgets_keypad()` (which calls `getchar_keypad()`).

Background

To accomplish its task `getchar_keypad()` must read characters from the keypad. The `getkey()` function returns a single key code for each keystroke. Prototype:

```
char  getkey(void);
```

A call to `getkey()` might be: `key = getkey();`

Corresponding to each of the 16 keys in the keypad, the key code is shown in the following table.

Keystroke	Decimal Code	Defined Symbol
← (delete)	8	DEL
ENTR	10	ENT
- (minus)	45	
. (decimal pt.)	46	
0 - 9 (digits)	48 - 57	
UP	91	UP
DWN	93	DN

(The symbols are defined in the header file `me477.h`)

In addition, `getchar_keypad()` must be able to write characters (decimal digits and - sign) to the LCD screen. The standard C language function `putchar_lcd()` should be used. Its prototype is

```
int  putchar_lcd(int c);
```

where both the input parameter and the returned value are the character to be sent to the display. Calls to `putchar_lcd()` might be:

```
ch = putchar_lcd('m');  or  putchar_lcd('\n');
```

It places the character corresponding to its argument on the LCD screen.

The `putchar_lcd()` function uses the same escape sequences that you used in `printf_lcd()`:

Escape Sequence	Function
\f	Clear Display
\b	Cursor left, 1 space
\v	Cursor to Start Line-1
\n	Cursor to Start Line-2

Again, next week, not this week, you will write your own versions of the `putchar_lcd()` and `getkey()`. This week they will automatically link to your program from the library when you call them from `getchar_keypad()`.

Laboratory Procedure

Test and debug your program.