01.L Lab Exercise: Introduction to myRIO C programming and high−level io drivers

Objectives

In this exercise you will gain experience with:

1. C programming for myRIO.
2. The beginning of a device driver for the keypad/LCD.
3. On-line debugging techniques.

Introduction

In Lab Exercises 01, 02 and 03, we will write several functions that will allow a user to interact with the program through the keypad and LCD screen. Below is an outline of the functional dependencies and corresponding Lab Exercises. Functions provided by the `me477` library, core C, or the standard C library will be overwritten by those we write, which are shown in **`green`**.

- **`double_in`** (Lab 01) prompts LCD and returns keypad double ← this lab!
  - **`fgets_keypad`** (Lab 02) gets string from keypad
    - **`getchar_keypad`** (Lab 02) gets char from keypad
      - **`getkey`** (Lab 03) gets char from keypad
      - **`putchar_lcd`** (Lab 03) prints char to LCD
  - **`printf_lcd`** (Lab 01) prints string to LCD ← this lab!
    - **`putchar_lcd`** (Lab 03) prints char to LCD
    - `vsnprintf` (Lab 01) assigns to formatted string
  - `sscanf` (Lab 01) converts ASCII to binary
  - `strstr` (Lab 01) find string in string
  - `strpbrk` (Lab 01) find member in string

It is important to note that these functions are already available in `me477` library, so when we write our own version of a function, it supersedes the library version. This allows us to depend on the lower-level functions without writing them, first.

In this Lab Exercise, in addition to the `main` program, you will write `double_in` and `printf_lcd`. At this point, you are expected to have only an elementary knowledge of C, but you should become familiar with the procedures, such as debugging, that you will need in the future.

Pre-laboratory preparation

Complete the following and make sure your functions compile before running them while connected to the lab hardware.

Part #1 User input: writing the function *`double_in`*

Very often in an interaction between a computer and a user, a message or "prompt" is written on the LCD display and the user is expected to respond by entering an appropriate decimal number through the keypad. In this laboratory exercise you will write a C function, called `double_in`, to perform the complete keypad/LCD procedure.
This function will be used here, and in later exercises, to obtain numerical information through interaction with the terminal. It should execute the following steps each time it is called.

1. A user prompt (a string of ASCII characters) is written on Line-1 of the LCD display. A pointer to the string corresponding to this prompt is the only argument of the `double_in` function.
2. A floating point number is accepted from the keypad in response to the prompt. If an error occurs in the input string, the display is cleared, an error message is written on Line-2 of the display, and the prompt is issued again on the first line. The number is entered as a string of ASCII characters that may include the decimal digits 0 - 9, a decimal point, and a minus sign, and is terminated by ENTR.
3. The entered string is interpreted as a floating point number.
4. The floating point number (C data type `double`) is returned from `double_in` function to the calling program.

The prototype of the `double_in` function is

```
double  double_in(char *prompt);
```

For example, a call to `double_in` might be:

```
vel = double_in("Enter Velocity:  ");
```

The variable `vel` would be assigned the value entered.

The LCD interaction would look like:

```
Enter Velocity:  -50.75

```

Or, if an error occurs: (e.g. user enters: `-50..75`)

```
Enter Velocity: _
Bad Key.  Try Again.
```

Allow for four possible user errors:

Error Type	Error Message Displayed on Line-2
No digits are entered (e.g. ENTR only)	Short. Try Again.
↑ or ↓	Bad Key. Try Again.
"−" other than first character (e.g."−−")	Bad Key. Try Again.
".." double decimal point	Bad Key. Try Again.

Our goal here is that the user must enter a valid number before the `double_in` function can exit. Notice that the errors are detected in the string that the user enters.
Here is a possible strategy for `double_in`:
Begin by using the `printf_lcd` function (which we will also write in this exercise) to display the prompt on the LCD screen. Then,

1. Use `fgets_keypad` (get string) to obtain the string from the keypad. Its prototype is:

   ```
   char * fgets_keypad(char *buf, int buflen);
   ```

   When `fgets_keypad` is called, as in `fgets_keypad(string, 40)`, it assigns the characters from the keypad to the `string` variable, which should have been declared to be a character array, like `static char string[40]`. However, if ENTR is pressed, `fgets_keypad(string, 40)` returns `NULL` (instead of writing to `string`). So if you defined `flag = fgets_keypad(string, 40)`, if ENTR is pressed `flag == NULL` should be true.
2. Use `strpbrk` (string pointer break) to detect ↑ or ↓. Note: ↑ is returned by `fgets_keypad` as the ASCII character `[` and ↓ as `]`.
3. Use `strpbrk` to detect minus signs `-` beyond the first character.
4. Use `strstr` to detect double decimal points (i.e. `..`).
5. Use `sscanf` (scan formatted from string) to perform the ASCII-string-to-double conversion. Hint: because `sscanf` is converting to a variable of type `double`, you need to use the format `%lf` (long float).

Note: `printf_lcd` and `fgets_keypad` work like the standard C functions `printf` and `fgets`, and are linked to your program from `me477` library.
Write a main program that tests your `double_in` function by calling it twice from the `main` program, assigning each result to a different variable. Then, as a check, print the values of both variables on the console using `printf`. See Algorithm L.1 for `main` pseudocode and Algorithm L.2 for `double_in` pseudocode.

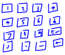
Part #2 Display on LCD: writing the function *printf_lcd*

Our second task is to write the `printf_lcd` function used by `double_in`. The C function

Algorithm L.1 `main` pseudocode

```
function main
    declare double variable vel for velocity
    open connection to myRIO and check for success
    call double_in and assign output to vel
    print vel to LCD with printf_lcd
    close myRIO connection and return its status
end function
```

x = double_in("enter value for x: ");



1..2 ✗
1↓2 ✗
1−2 ✗
1.2 ✓

x = 1.2

Algorithm L.2 `double_in` pseudocode

```
function double_in(p)    ▷ p is prompt pointer
    declare variables
    clear LCD display    ▷ use printf_lcd
    c ← 1    ▷ initialize stop check
    while c == 1 do
        print p to LCD    ▷ use printf_lcd
        f ← fgets_keypad(s,..)    ▷ get string s and out flag f
        if f == NULL then
            print "Short. Try again." to LCD    ▷ use printf_lcd
        else if s does not pass bad key checks then
            print "Bad key. Try again."    ▷ use printf_lcd
        else
            c ← 0    ▷ set stop condition
            sscanf(s,"%lf",&v)    ▷ convert s to double and assign to v
        end if
    end while
    return v
end function
```

`printf` prints to the standard output device, in our case the Console pane of the Eclipse IDE. We want `printf_lcd` to operate exactly as `printf`, except that it will print to the LCD screen. Refer to your C text. To do this, we want `printf_lcd` to accept a format string with a variable number of arguments. Therefore, the prototype for `printf_lcd` is

```
int printf_lcd(const char *format, ...);
```

where `format` is a string specifying how to interpret the data, and the ellipsis (. . .) represents the variable list of arguments specifying data to print. The return value is an `int` equal to the number of characters written if successful or a negative value if an error occurred.
For example,

a = 3.14159···

```
n = printf_lcd("\fa = %f, b = %f", a, b);
```

%f0.2 3.14

Here is a suggested strategy for `printf_lcd`:

- Use the C function `vsnprintf` to write the data to a C string.
- Then use the LCD driver function `putchar_lcd` to successively write each character in the string to the LCD display. Note: It is strongly suggested that you use an incremented pointer to access the string, rather than an array index. See Lec. 01.L for more guidance on `putchar_lcd`.

The C function `vsnprintf` writes formatted data from the variable argument list to a buffer (the string) of a specified size.
The tricky part is passing the variable argument list of `printf_lcd` to `vsnprintf`. Here is an example fragment of code. From your C text, study the data type `va_list`, and the C macros `va start` and `va end` to see how this works.

```
int printf_lcd(char *format, ...) {
    va_list args;
    va_start(args, format);
    n = vsnprintf(string, 80, format, args);
    va_end(args);
}
```

As usual, you must allocate storage for the C `string` of length 80.
The `main` program, the `double_in` function, and the `printf_lcd` function should all be in the same file: `main.c`. Be sure to `#include` the header files `me477.h`, `<stdio.h>`, `<stdarg.h>`, and `<string.h>` in the code.
Once you have defined `printf_lcd` within your `main.c`, your code will supersede the version in the `me477` library. See Algorithm L.3 for pseudocode for `printf_lcd`.

Algorithm L.3 `printf_lcd` pseudocode

```
function printf_lcd(f, v)    ▷ f is string format, v is variable data to print
    declare variables
    start parse args with va_list, va_start
    n ← vsnprintf(S,80,f,args)    ▷ S is the string char length 80
    finish parse args with va_end
    if n < 0 then    ▷ test for conversion error
        return n
    end if
    initialize s    ▷ s points to start of S
    while dereferenced s is not 0 do    ▷ check if S is done
        putchar_lcd(dereferenced s with postfix increment)
    end while
    return n
end function
```

Laboratory procedure

Debug and test your C program. As necessary, use breakpoints and single-stepping to find errors.

Guidance

This section provides guidance on several aspects of the Lab Exercise, above.

Background on *putchar_lcd*

The C function `putchar_lcd` places the single character corresponding to its argument on the LCD screen. 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. A character constant is an integer, written as one character within single quotes, such as `'x'`.
For example, calls to `putchar` might be:

```
ch = putchar('n');
putchar('\n');
```

To write both parts of your program you also need to know how the escape sequences used in the `putchar_lcd` function affect the LCD screen. This concerns the important matter of I/O (input/output), which we will consider in detail later. For now the following table explains the escape sequences:

Escape Sequence	Function
\f	Clear Display
\b	Move cursor left one space
\v	Move cursor to the start of line-1
\n	Move cursor to the start of the next line

\v\f a \n b

Dissecting a C program

This lab requires the use of several aspects of the C programming language. In this section, some of that is outlined, but a C textbook such as Kernighan and Ritchie (1988) is required for sufficient understanding.

We begin by writing a simple C program that sums loop indices and proceed unpack its meaning.

```
1   /* include libraries */
2   #include "stdio.h"
3
4   /* declare function prototypes */
5   int sum(int x); /* sum */
6
7   /* define external/global variables */
8   #define N 5 /* number of loops */
9
10  /* define functions */
11  int main(int argc, char *argv[]) {
12      static int x[10]; /* total */
13      static int i; /* index */
14      for (i=0; i < N; i++) {
15          x[i] = sum(i);
16          printf("%d",x[i]);
17          if (i < N-1) {
18              printf(",");
19          }
20          else {
21              printf(".");
22          }
23      }
24      return 0;
25  }
26
27  int sum(int x) {
28      static int y=0; /* initialize y */
29      y = y + x;
30      return y;
31  }
```

< >

// main

/*
some
text
coment
*/

int y;
y=0;

```
0,1,3,6,10.
```

C programs consist of variables and functions. Variables are defined via an assignment statement, the most common operator is `=` as when our program assigns the first value of the variable `i` in the expression `i = 0`, which could also be written `i=0`—spaces are added for clarity.

Our program has two functions: `main` and `sum`. Whenever a C program is executed, it begins with a function named `main`. Every program must have one. If we don't need to pass any arguments to our program, in its definition, the argument can be empty, as in:

```
int main() { /* statements */ }
```

If we need to pass arguments—say, from the command-line—there is a specific method described in detail by Kernighan and Ritchie (1988, p. 114) that would have as its definiton:

```
int main(int argc, char *argv[]) { /* statements */ }
```

For our program, we don't use the arguments, so either is valid.

The `int`s before the definitions of `main` and `sum` declare that these functions return data type `int` for integer. Although it is not strictly required in every instance, it is considered best practice to always precede a definition with its return data type.

Most C programs load external libraries with pre-compiled functions. The most popular libraries are from the C standard library. For `i/o` functions like the `printf` we use here to print to the console, the `stdio.h` header file must be `#include`d, as shown at the top of our program. We'll include the header file `me477.h`, which includes compiled versions of the functions we'll be writing over the next few Lab Exercises. Best practice is to declare prototypes for each function (we often skip `main`, which always has the same prototype), which, for our `sum`, looks like:

```
int sum(int x);
```

Here we're declaring that `sum` is a function with a single integer argument, which we'll call `x` inside the function, that returns an integer to the calling function. These declarations should be before `main`. The function definition can occur either before or after `main`, but we adopt the convention of defining functions after `main`.

External or global variables are those defined above `main`. These variables are defined once and can be accessed by every function that declares it with `extern`. A similar, but distinct object is the symbolic constant, defined by `#define`, as with `N` in our program. A difference is that symbolic constants need not be declared within a function. We conventionally capitalize symbolic constants.

Line 12 shows the declaration of variable `x` to be an array with `10` elements. This preallocates a block of memory for `x`. Most variables inside a function are automatic: they are not retained between function calls. However, often in embedded computing we will be using pointers to specific addresses in memory at which a variable can be found. The safest way to use pointers is to declare the variable to be `static`, as in Lines 12, 13, and 28. A very important consequence of this declaration is that the variable's value is retained between function calls. For instance, in `sum`, we initialize `y` to be a static integer (`0`), then add the argument `x` to it are return the sum, which has overwritten `y`. Each successive call, the old value of `y` is retained, so on the second call, third call, for which `x` is 2 and the old `y` is 1, the returned `y` is 3.

Line 14 is the beginning of a `for` loop. We highlight two syntactical nuances. First, there are the three flow control components in the statement `(initialize;condition;increment)`. The initialize statement is executed first and only once. The condition statement returns a boolean (actually just an `integer`) of `1` for true and `0` for false. If the condition is true, the statements between the following braces are executed. Afterwards, the increment statement is evaluated and the loop returns to evaluate the condition ….

The second syntactical consideration is that the braces `{}` should enclose the looped block. Although a single statement need not be enclosed, multiple statements must, and therefore we adopt the convention of always enclosing loop statements in braces.

The `if/else` execution control keywords are straightforward and are not expanded upon, here.

Finally, `main`, like any function, should return to the calling function (for `main`, calling program) some value, which, for most functions, can be of any data type, but for `main` is a status code as an `int`eger. The `return` keyword defines the return status, in our program, simply `0`. Conventionally, this signifies to the calling program that our program has run successfully. Nonzero `main` return values are used to signify different error codes, which should be documented for your program.

Execution control

As we saw in the example above, C has the usual execution control statements, which include `while`, `for`, `if`, `else`, and `else if`. This Lab Exercise should familiarize you with several of these.

C data types

C has only a few core data types:

- `char`s are single byte characters;
- `int`s are integers, the size of which is machine-dependent;
- `float`s are single-precision floating-point numbers, the size of which is machine-dependent; and
- `double`s are double-precision floating-point numbers, the size of which is machine-dependent.

Typically, a `float` is 32-bit and a `double` is 64-bit. There are also qualifiers such as `short` and `long`, which compilers typically take to mean "fewer" bytes for the specified representation or "greater," respectively.

Arrays are just lists of values. When declaring an array, one specifies the data type of each element and the number of elements, as in `double x[10];`, which is an array of ten `double`s. Accessing element `n` of an array `x` is done with the syntax `x[n]`. It is important to note that the first index of an array is `0` in C.

Pointers

Pointers are a key concept in C. A pointer is variable that is assigned not a value, but a memory address. To get some variable `x`'s value address, one uses the address operator `&`, like `&x`. In order to assign this to a pointer variable, the variable must be declared as a pointer to a specific data type. For `x` an integer, a pointer to it can be declared with `int *p;` and assigned with `p = &x`. To access the value to which a pointer `p` points, the dereferencing operator `*` can be used, as in `*p`.

Consider the following example.

```c
#include "stdio.h"

int main() {
    static int x = 1;
    static int *p = &x;
    printf("%d\n",x);  /* value */
    printf("%p\n",&x); /* address */
    printf("%p\n",p);  /* pointer */
    printf("%d\n",*p); /* deref'd pointer */
    return 0;
}
```

```
1
0x100693018
0x100693018
1
```

An array variable, say an integer array of length 10 declared by `int z[10];`, is just a pointer to the first value in the array. An array name is a constant pointer, so it cannot be reassigned (e.g. if `p_a` is an array, this is invalid: `p_b = p_a;`).

Cast operator

A cast operator on an expression to type type is `(type) expression`. It represents the expression in the new type in accordance with certain rules. It does not affect any definitions in the original expression; rather, it returns a new expression. Suppose you have the following:

```c
static int a; // 2-byte
static long int b; // 4-byte
b = 3;
a = (int) b; // cast and assign to a
```

The casting of a four-byte `long int` to a two-byte `int` means there is a potential for truncation because four bytes can represent more integers.

When casting to an `int` from a `float` or `double`, beware that truncation does not round in the usual sense: it simply drops the fractional part. It is preferable to use the function `round` provided by the standard library header file `math.h`.

Incrementing and decrementing

For `int x = 0`, instead of writing `x = x + 1` to increment `x`, we can write either `++x` or `x++`. The former is called a prefix operator and the latter postfix, both of which increment `x`, but they are interpreted differently in an expression:

- `++x` increments `x`, then uses it in the expression in which it appears (e.g. `n = ++x` assigns `1` to `x`, then `1` to `n`) and

- `x++` uses `x` in the expression in which it appears, then increments it (e.g. `n = x++` assigns `0` to `n`, then `1` to `x`).

The decrement operator `--` also has pre- and postfix versions, but subtracts one instead of adding.

The next example shows how pointers—not just **`int`**s—can be incremented. They can also be decremented. Incrementing a pointer moves it not to the next address, but to the next piece of data in memory, skipping the necessary number of bytes.

Operator precedence and associativity

See Lec. 02.2 for a table of operator precedence and associativity. The following example shows some interesting precedence and associativity interactions among operators `*` and `++` and parentheses `()`.

```c
#include "stdio.h"

int main() {
    static int x = 5;
    static int *p = &x;
    printf("(int) p   => %d\n",(int) p);
    printf("(int) p++ => %d\n",(int) p++);
    x = 5; p = &x;
    printf("(int) ++p => %d\n",(int) ++p);
    x = 5; p = &x;
    printf("++*p      => %d\n",++*p);
    x = 5; p = &x;
    printf("++(*p)    => %d\n",++(*p));
    x = 5; p = &x;
    printf("++*(p)    => %d\n",++*(p));
    x = 5; p = &x;
    printf("*p++      => %d\n",*p++);
    x = 5; p = &x;
    printf("(*p)++    => %d\n",(*p)++);
    x = 5; p = &x;
    printf("*(p)++    => %d\n",*(p)++);
    x = 5; p = &x;
    printf("*++p      => %d\n",*++p);
    x = 5; p = &x;
    printf("*(++p)    => %d\n",*(++p));
    return 0;
}
```

```
(int) p   => 81195032
(int) p++ => 81195032
(int) ++p => 81195036
++*p      => 6
++(*p)    => 6
++*(p)    => 6
*p++      => 5
(*p)++    => 5
*(p)++    => 5
*++p      => 0
*(++p)    => 0
```

Strings

Strings are arrays of **`char`**s, terminated by a `NULL` (which is a pointer that casts to `0`). For instance, the string `"HELLO"` could be represented in memory (with corresponding ASCII codes) as follows.

"H"
"E"
"L"
"L"
"O"
NULL

Function argument passing

All function arguments in C are passed "by value": the function receives its arguments through temporary local variables called automatic variables (see Lec. 01.L for more about automatic and global variables). When it's necessary to pass back an argument with a changed value, the caller can provide the function with the argument address via a pointer, and the function must access the value through the pointer. A potential alternative is a global **`extern`** variable.

Literal of a **`long`**

For the compiler to recognize a literal number as a **`long`**, it must have an `L` suffix. For instance, if

`val` is a **`long`** variable and you want to compare it to `32767`:

```c
if(val > 32767L) { /* validated! */ }
```

NULL detection

The following program gives some insight into detecting a returned `NULL`.

```c
#include "stdio.h"

int main() {
    printf("%p\n",NULL); /* print as pointer */
    printf("%d\n",(int) NULL); /* cast to int */
    if (NULL == 0) {
        printf("this works\n");
    }
    if (NULL == 0x0) {
        printf("this works, too!\n");
    }
    if (NULL == NULL) {
        printf("so does this!");
    }
    return 0;
}
```

```
0x0
0
this works
this works, too!
so does this!
```

Hex numbers—signed

In addition to the specifically C-related topics, above, the following is useful for the first assignment.

We can change the sign of a signed binary by taking the two's complement.

To put a negative hexadecimal number into a signed hexadecimal form, take the sixteen's complement. Steps:

- take fifteen's complement and
- add 1.

Convert 3A to -3A.

02

Exploring C and mid–level io