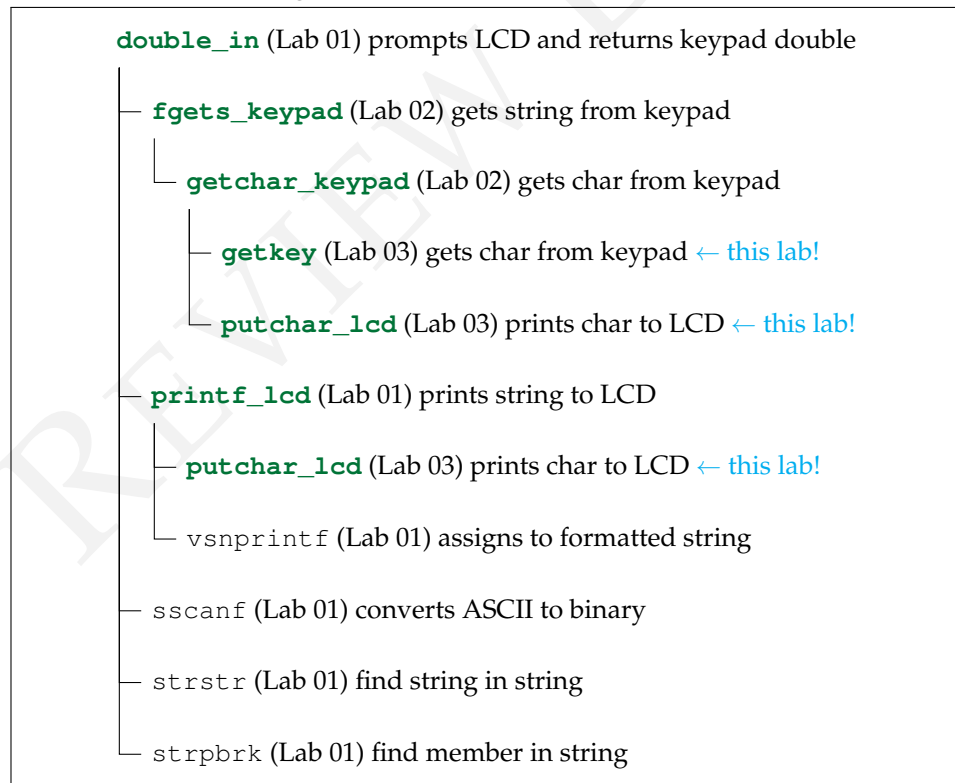## Lab Exercise 03 Low-level character io

Lab 03.1   Objectives

In this exercise you will gain experience with:

1. The keypad and LCD display.
2. Code requirements for character I/O of a custom embedded computing application.
3. On-line debugging techniques.

Lab 03.2   Introduction

In this lab you will write the lowest-level routines for character I/O for our keypad and LCD display. They are the putchar_lcd function and the getkey function called from getchar_keypad in Lab Exercise 02, as shown in the following function structure.

double_in (Lab 01) prompts LCD and returns keypad double

    fgets_keypad (Lab 02) gets string from keypad

        getchar_keypad (Lab 02) gets char from keypad

            getkey (Lab 03) gets char from keypad ← this lab!

            putchar_lcd (Lab 03) prints char to LCD ← this lab!

    printf_lcd (Lab 01) prints string to LCD

        putchar_lcd (Lab 03) prints char to LCD ← this lab!

        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

Lab 03.3   Pre-laboratory preparation

Two functions, in addition to `main`, must be written in the exercise.

Lab 03.3.1   Part #1: character output: writing `putchar_lcd`

The function `putchar_lcd` puts a single character on the LCD display. The character may be any in the ASCII code or any of the escape sequences described in Lab Exercise 01 (`\f`, `\v`, `\n`, `\b`). The prototype of the `putchar_lcd` function is

```
int putchar_lcd(int value);
```

where the argument (`value`) is the character to be sent to the display. If the input value is in the range $[0, 255]$ then the returned value is also equal to the input value. If the input value is outside that range then an error is indicated by returning `EOF`.

 Your version of `putchar_lcd` will replace that in the `me477` library. Calls to `putchar_lcd` might be

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

Serial data is sent to the LCD display through a Universal Asynchronous Receiver/Transmitter (UART). Write the `putchar_lcd` to perform four functions:

1. Initialize the UART the *first* time that `putchar_lcd` is called.
2. Send a character to the display or send a decimal code to the display to implement an escape sequence.
3. Check for the success of the UART write.
4. Return the `EOF` error code, if appropriate. Otherwise, return the character to the calling program.

 The UART must be initialized *once* before any data is passed to the display. It is initialized through the `Uart_Open` function that sets appropriate myRIO control registers to define the operation of the UART. The initialization may be accomplished as shown in Listing 03.1, where `uart` (type: `static MyRio_Uart`) is a port information structure, and the returned value is assigned to `status` (type: `NiFpga_Status`). The macros `Uart_StopBits1_0` and `Uart_ParityNone` are defined in `UART.h`. You must `#include` `UART.h` in your code.

```
uart.name = "ASRL2::INSTR"; // UART on Connector B
uart.defaultRM = 0;         // def. resource manager
uart.session = 0;           // session reference
status = Uart_Open( &uart,  // port information
                    19200,  // baud rate
                    8,      // no. of data bits
                    Uart_StopBits1_0, // 1 stop bit
                    Uart_ParityNone); // No parity
```

Listing 03.1: initializing the UART.

```
status = Uart_Write( &uart,  // port information
                     writeS, // data array
                     nData); // no. of data codes
```

Listing 03.2: writing to the UART.

Perform this UART initialization just once, and immediately return EOF from putchar_lcd if status is less than the VI_SUCCESS macro.

Escape sequences, received as the argument of putchar_lcd, control the cursor position and the function of the LCD display. They are implemented by sending the escape sequences of Table 02.2.

Arguments of putchar_lcd, in the range of 0 to 127, are sent to the display where they are interpreted as the corresponding ASCII characters. Other arguments, in the range 128 to 255 are used for special control functions of this display.

Both escape sequences and ASCII characters are sent to the display using the Uart_Write function. A typical call would be as shown in Listing 03.2, where uart is the port information structure defined during the initialization, writeS (type: **uint8_t**) is an array containing the data to be written, and nData (type: **size_t**) indicates the number of elements in writeS. Again, return EOF if status is less than the VI_SUCCESS. Under normal operation (no errors), return the input character to the calling program.

See Algorithm 5 for putchar_lcd pseudocode.

Lab 03.3.2   Part #2: keypad input: writing getkey

You will write the getkey function, which waits for a key to be depressed on the keypad, and returns the character code corresponding to that key.

Algorithm 5 buffered `putchar_lcd` pseudocode

---

function PUTCHAR_LCD(c)   ▷ *c is ASCII character code*
    initialize variables   ▷ *include* **`static int`** `iFirst=1`
    if `iFirst==1` **then**   ▷ *first call!*
        initialize UART (Listing 03.1)   ▷ `status ← Uart_open(...)`
        if `status < VI_SUCCESS` **then**
            return EOF
        end if
    end if
    $n \leftarrow 1$ ▷ *assume n (data points) is 1*
    if `c == '\f'` **then**   ▷ *clear display, backlight on*
        `S[0] ← 17`   ▷ *S is* **`uint8_t`** *array*
        `S[1] ← 12`
        $n \leftarrow 2$ ▷ *n actually 2 in this case*
    else if `c == '\b'` **then**   ▷ *cursor backspace*
        `S[0] ← 8`
    else if `c == '\v'` **then**   ▷ *cursor line-0*
        `S[0] ← 128`
    else if `c == '\1'` **then**   ▷ *cursor line-1*
        `S[0] ← 148`
    else if `c == '\2'` **then**   ▷ *cursor line-2*
        `S[0] ← 168`
    else if `c == '\3'` **then**   ▷ *cursor line-3*
        `S[0] ← 188`
    else if `c == '\n'` **then**   ▷ *cursor to next line*
        `S[0] ← 13`
    else if `c > 255` **then**   ▷ *outside range*
        return EOF
    else   ▷ *send ascii code*
        `S[0] ←` c cast as **`uint8_t`**   ▷ *cast syntax* (**`uint8_t`**) c
    end if
    write S to UART (Listing 03.2)   ▷ `status ← Uart_Write(...)`
    if `status < VI_SUCCESS` **then**
        return EOF
    else
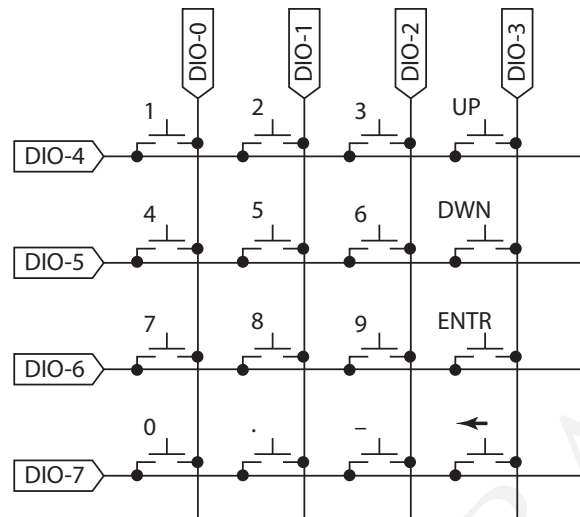   **return** c
    end if
end function

Figure 03.4: keypad circuit.

The prototype of the `getkey` function is

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

Your version of `getkey` will replace that in the C library. A call to `getkey` might be:

```
key = getkey();
```

The keypad is a matrix of switches. When pressed, each switch uniquely connects a row conductor to a column conductor. The row and column conductors are connected to eight digital I/O channels of connector-B (`DIO-0`–`DIO-7`) of the myRio as shown in Figure 03.4.

Each channel may be programmed to operate as either a digital input or an output. As an output, the channel operates with low output impedance as it asserts either a high or a low voltage at its terminal. Programmed as an input, the channel has high input impedance ("Hi-Z mode") as it detects either a high or a low voltage.

How will we detect if a key is depressed? Briefly, this is accomplished by driving (as output) one column to low voltage (digital false), with the other columns channels in Hi-Z mode. Then, all of the rows are scanned (detected). If a row is found to be low, the key connecting that row to

the driven column must be depressed. This procedure is repeated for each column. The entire process is repeated until a key is found.

pull-up resistor　　Essential to this scheme is that a *pull-up resistor* is connected between each channel and the high voltage.[3] So, unless a row is connected (through a key) to a low-impedance, low-voltage column, it will always read high.

Strategy　　A strategy for `getkey` is shown in the pseudocode Algorithm 6.

Algorithm 6 `getkey` pseudocode

---

function GETKEY
　　initialize the 8 digital channels
　　while a low bit not detected **do**
　　　　for each column **do**
　　　　　　for each column **do**
　　　　　　　　set column to Hi-Z
　　　　　　end for
　　　　　　set one column low
　　　　　　for each row **do**
　　　　　　　　read bit
　　　　　　　　if bit is low **then**
　　　　　　　　　　break row loop
　　　　　　　　end if
　　　　　　end for
　　　　　　if bit is low **then**
　　　　　　　　break out of column loop
　　　　　　end if
　　　　end for
　　　　wait for some msec
　　end while
　　while row is still down **do**
　　　　wait for some msec
　　end while
　　identify key from row, column in table
　　return key
end function

---

[3]The *NI myRIO-1900 User Guide and Specifications* describes the DIO as having built-in 40 KΩ pull-up resistors to 3.3 V (Instruments, 2013, p. 11).

Channel initialization The `MyRio_Dio` structure, defined in `DIO.h`, identifies the control registers and the bit to read or write for a channel.

```c
typedef struct { uint32_t dir;   // direction register
                 uint32_t out;   // output value register
                 uint32_t in;    // input value register
                 uint8_t bit;    // Bit to modify
} MyRio_Dio;
```

Declare an array of `MyRio_Dio` structures, one element for each of the 8 necessary channels. In a loop initialize the channels as follows.

```c
MyRio_Dio Ch[8];
for (i=0; i<8; i++) {
    Ch[i].dir = DIOB_70DIR;
    Ch[i].out = DIOB_70OUT;
    Ch[i].in = DIOB_70IN;
    Ch[i].bit = i;
}
```

Again, the symbols shown are defined in `DIO.h`.

Channel I/O

Input—Digital channel read function prototype:

```c
NiFpga_Bool Dio_ReadBit(MyRio_Dio* channel);
```

For example, a typical call might be:

```c
bit = Dio_ReadBit(&Ch[row+4]);
```

Note: In addition to reading the bit, `Dio_ReadBit` sets the channel to Hi-Z mode.

Output—Digital channel write function prototype:

```c
void Dio_WriteBit(MyRio_Dio* channel, NiFpga_Bool value);
```

For example, a typical call might be:

```c
Dio_WriteBit(&Ch[col], NiFpga_False);
```

The data type `NiFpga_Bool` may take values of either `NiFpga_True` (high), or `NiFpga_False` (low).

Key code   The key code returned by `getkey` is determined by the indices of a key code table. The key code table can be stored in a statically declared 4×4 array of characters.

```
char table[4][4] = { {'1','2','3', UP},
                     {'4','5','6', DN},
                     {'7','8','9',ENT},
                     {'0','.','-',DEL}  };
```

For example, if the detected row was `1`, and the column was `2`, then the value of `table[1][2]` is the character `'6'`.

The symbols `UP, DN, ENT, DEL` are defined in `me477.h`.

Wait   The `x` ms time delay will be determined by executing a delay-interval routine. The "wait" function below is suggested. It executes in a small fraction of a second. In next week's lab we will calculate and measure its precise duration.

```
/*----------------------------------------
Function wait
     Purpose:      waits for x ms.
     Parameters:   none
     Returns:      none
*----------------------------------------*/
void wait(void) {
  uint32_t i;

  i = 417000;
  while(i>0){
    i--;
  }
  return;
}
```

Lab 03.3.3   Writing the `main` function

Write a `main` function that tests your versions of `putchar_lcd` and `getkey`. It should:

1. Make at least one individual call to each of `putchar_lcd` and `getkey`. Be sure to test the value-out-of-range error returned by `putchar_lcd`.

2. Collect an entire string using `fgets_keypad` (which automatically calls `getkey`).
3. Write an entire string using `printf_lcd` (which automatically calls `putchar_lcd`). Be sure to test all four escape sequences.

Lab 03.4   Laboratory Procedure

Test and debug your program.