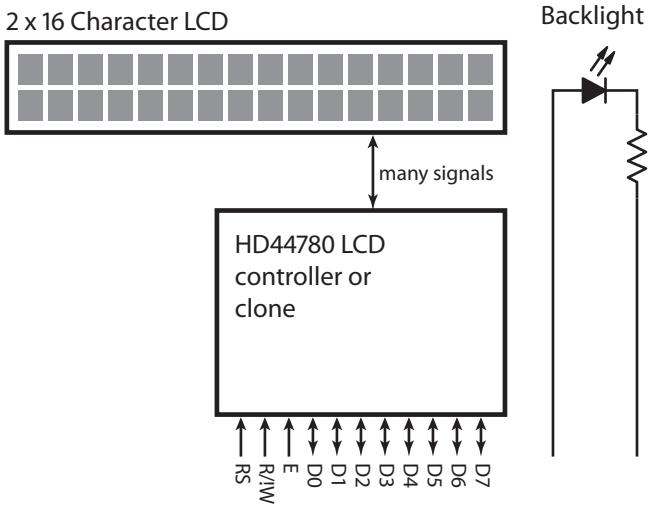
Exercise in Embedded Computing: Driving an Alphanumeric LCD

Sivan Toledo, Tel-Aviv University

November 7, 2010

The purpose of this exercise is to drive an alphanumeric LCD display. These displays are inexpensive and very common. These displays are usually controlled by a dedicated LCD controller chip, an HD44780 LCD or a clone. The LPC2148 Education Board contains a clone, KS0070B; the data sheet fo the HD44780 is clearer, so you should probaly use it rather than the data sheet of the KS0070B when you develop your code. The LCD unit shown on sheet 4/4 of the schematics of the Education Board represents not just the display, but a printed-circuit board that contains the dipslay, the controller, and the LED backlight.

The LCD controller contains 80 bytes of RAM that are mapped to the display. Each byte is mapped to one character on the diplay. In a 2-line display, like the one we have, the top line starts at address `0x40`. There is also a little bit of additional RAM to define custom characters. The controller has a bus interface that the microcontroller uses to write to this RAM, to read from it, and to send commands (instructions) to the controller. The LCD controller has an internal address pointer that that MCU can set using a bus command. When the MCU writes or reads a byte, this pointer is incremented automatically. The following diagram shows the structure of the diplay and names the bus signals.



The RS and R/!W signals determine the nature of each bus transaction, according

to the following table:

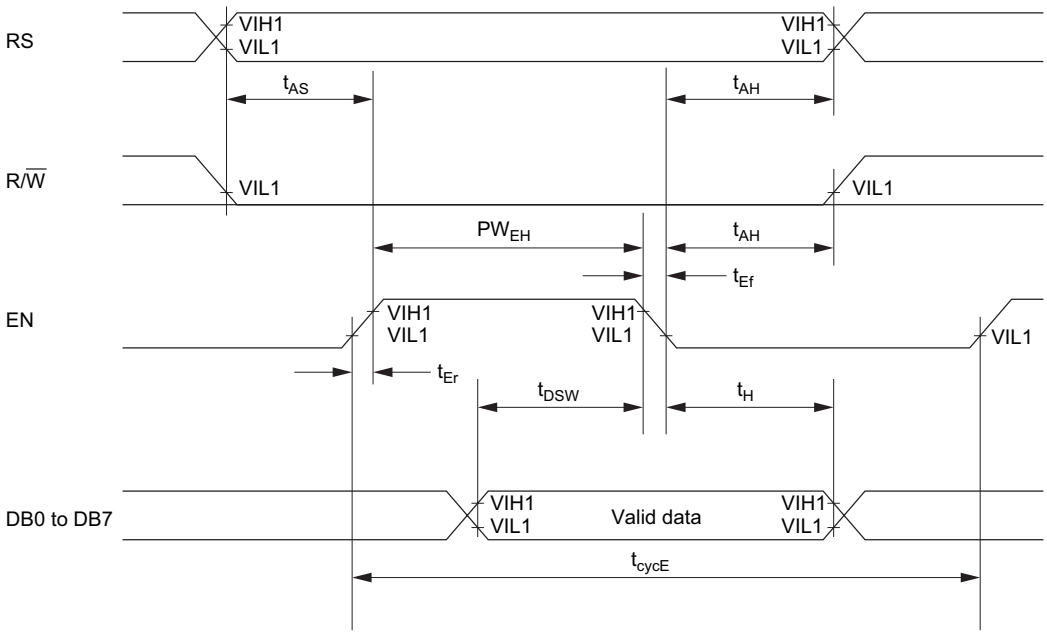
RS	R/!W	
1	0	the MCU writes to the controller's RAM
1	1	the MCU reads from the controller's RAM
0	0	the MCU sends an instruction to the controller
0	1	the MCU reads a busy flag (bit 7) and the address pointer (bits 0–6)

When the MCU sends data (or an instruction) to the LCD controller, the E signal tells the controller when to sample the data on the bus. From the timing diagram below we can infer that the data is sampled on the falling edge of the E signal. This means that a E signal must go down only after the signals on the data bus are stable, and then should remain stable for some time after the falling edge of E (the data sheet tells you for now long).

The bus can be used as an 8-bit bus or as a 4-bit bus. The 4-bit interface is slower, which is not important, and uses fewer MCU pins, which is often important. When we use the 4-bit interface, communication between the MCU and the LCD controller is still byte-based, but each byte is sent in two bus cycles, with the most-significant bits (MSB) sent first. The LCD controller always starts in the 8-bit mode. To tell it to switch to the 4-bit mode, the MCU sends the "function-set" command, whose 4 most-significant bits set the bus width. This command is sent in one bus cycle, because the LCD controller is is 8-bit mode. If the MCU only drives 4 bits, the lower 4 bits of the "function-set" command that the LCD controller gets can be anything, so the MCU must send it again, this time in two bus cycles.

Other instructions clear the display, set the internal address pointer, choose fonts (not much choice), cause the display to blink, show or hide a cursor, and shift the display.

To drive the LCD controller, you need to understand the behavior of the bus. Here is the timing diagram for writing to the LCD controller.



Your code needs to drive these signals and to obey the timing constraints that the data sheet specify. This mainly means the program must not drive the LCD controller too fast.

In addition to timing constraints on bus transactions, certain operations take a long time (more than a millisecond) to complete. The program must either wait more than the maximum time the operation takes, or it can periodically poll the controller's busy flag to find out when it finishes performing an instruction.

The driver that you will write will have a minimum of three functions: `lcdInit()`, `lcdPrintChar(uint8_t c)`, and `lcdGoto()`, which changes the internal address pointer. You can add additional functions to turn the display on and off, to clear it, and so on.

The same alphanumeric LCD can be used with almost any microcontroller, so we will keep the driver completely independent of the MCU. To achieve this independence, implement the driver in a separate C file `char-lcd.c` that you will `#include` into the main program. Do not use separate compilation. The driver will control the bus by calling macros or functions that the main program will define before the inclusion of `char-lcd.c`. The test program should look like this:

```
...
#define lcd_backlight_on() ...
#define lcd_en_rs_out()    ... /* set the direction to output */
#define lcd_rw_out()       ...
#define lcd_data8_out()    ...
#define lcd_data8_in()     ... /* set the direction to input  */
#define lcd_data4_out()    ...
#define lcd_data4_in()     ...
#define lcd_data8_set(c)   ...
#define lcd_data4_set(c)   ...
#define lcd_rs_set()       ( IO1SET = BIT24 )
#define lcd_rs_clear()     ( IO1CLR = BIT24 )
#define lcd_en_set()       ...
#define lcd_en_clear()     ...
#define lcd_rw_set()       ...
#define lcd_rw_clear()     ...
#include "char-lcd.c"

int main() { ... }
```

1. Start with a partial implementation of `lcdInit` that only turns on the backlight. Without a working backlight, you won't be able to see text on the display.

2. Your LCD driver needs a way to make sure that it satisfies lower-bounds on bus timing. Implement in the main program a function `busywait(uint32_t microseconds)`that will wait a given number of microseconds or longer. We will calibrate it in a future exercise, but right now just make sure it waits at least as long as the argument tells it to. You can achieve this by assuming that the processor runs at 60 MHz or less and that each instruction takes at least one clock cycle to complete. Don't worry if the wait is significantly longer than the argument.

3. Read the datasheet of the LCD controller and implement the rest of the driver using the 8-bit bus interface. Make sure your code waits for the LCD to reset; this is easy to overlook, because when you download code repeatedly to the board to test it, the LCD remains on all the time, so you won't notice if your code tries to communicate with the LCD controller too soon after power up.

4. Modify the driver to use the 4-bit bus interface. You should not use any of the `data8` macros or functions; you should assume that there are no MCU pins connected to D0 to D3.

5. Additional ideas:

   (a) Implement driver functions to support blinking and a cursor.

   (b) Implement a driver function that accepts a string and displays it; use `'\n'` to separate the two lines (so that one string argument can span the two lines).

   (c) Enhance the function from part (b) to display strings that are longer than 16 characters, by slowly shifting the display to the left after an initial delay, to reveal the part of the text beyond 16 characters.

   (d) Implement a function that reads back data from the LCD controller. Because the controller has only 80 bytes, it's RAM is not terribly useful, but reading from it is a good exercise in controlling a bi-directional bus. Consult the timing diagram of read transactions in the controller's data sheet.

   (e) Implement custom graphic characters.