Developer's Guide to nlcd.c and nlcd.h

Contents (page number):

Introduction

The files **nlcd.c** and **nlcd.h** compose the driver for a Newhaven Display International Character Liquid Crystal Display Module model number NHD-0116GZ-FSW-FBW (referred to as the "LCD" from here on). This LCD allows for one line of sixteen 14.54mm by 6.00mm dot matrix alphanumeric characters. The LCD is controlled by a SUNPLUS SPLC780D controller chip ("controller chip" from here on), which makes for an unusual approach when writing the driver code. The controller chip was intended for two lines of eight characters, while the LCD is only a single line of sixteen characters. Thus a few of the features of the controller chip (shifting in particular) cannot be used as designed, as will be explained later.

The drivers were built to be used with an ATmega168 8-bit AVR microcontroller, though theoretically any AVR microcontroller could be used with minor adjustments. The project was initiated out of a NerdKits project, so the driver was also built using the **delay.h** file created by the NerdKits designers.

This is a 4-bit initialization.

Rationale for Approach

This code was written with future development in mind. The abundance of functions serves this purpose and makes the code highly readable. The function and variable names avoid ambiguous abbreviations and acronyms as much as possible (with the exception of lowest-level microcontroller port and pin names assigned by the AVR library). Care was made to comment or document any tricky or creative sections of the code so future developers can quickly understand how to implement further functions. The rationale for the way scrolling has been implemented is explained in the scrolling section.

Description

#includes
avr/pgmspace.h - For reading from the microcontroller's program memory (constant strings).

nlcd.h - Header file containing a number of used macros.
delay.h - For delaying by a set number of micro- or milliseconds with the 14.7456 MHz clock.

.h file #defines
The commands to the LCD were determined from page 6 of the LCD driver datasheet.

State Variables

static int next_pos
Keeps track of the next display position to be written to. For example, when it equals 0, the 0th (of 0 - 15) display position is the next position to be written to. This variable is utilized to determine when the end of the screen has been reached (when scrolling is not enabled), when the split in DDRAM been reached (indicating the need to jump either one direction or another over the gap), and when the display has nothing written to it (indicating that further backspacing should do nothing).

Incremented: when a character has been successfully written to the screen.

Decremented: when a backspace has been successfully written to the screen.
Reset: when nlcd_wipe() is called, clearing the screen and the buffer.

Notes: next_pos increments to 16. Thus next_pos == 16 (macro'd as LAST_POS + 1) serves as an extra state. This is not indicating that the next displayable position is 16 (there are only 0…15 = total of 16 character positions displayable at a time), but rather as an "if scrolling is enabled, then we will need to scroll here" determiner. When scrolling is not enabled, then it simply means that the display is full. With a backspace, the variable returns to next_pos = 15, and scrolling stops until it reaches it reaches the value 16 again.

static int db_next_pos
Keeps track of the next buffer array position to be written to, modulo 16. The counter actually increments past the buffer's size of sixteen (when necessary), with the modulo 16 operation saving the code from having to wrap the counter from position 15 to 0 on an increment or 0 to 15 on a decrement.

Incremented: with each new character typed in (regardless of whether or not scrolling is enabled).

Decremented: with any backspace character not typed with the screen empty (a null operation occurs in that case).

Reset: when nlcd_wipe() is called, clearing the screen and the buffer.

Notes: this variable also serves as the first position of the current buffered string, or the first character to be printed on a call to nlcd_bstring(). The idea here is that the first character you type in is the last character pushed out, so this position is simultaneously where the first character to be printed is located as well as where the next inputted character will be located.

static unsigned char scrolling_enabled

Allows the screen to scroll (shift) when the screen is filled. Note that this does not implement "scrolling back": a backspace shifting the screen to the right. Scrolling is forward-only.

Activated (1): when nlcd_enable_scrolling() is called.

Deactivated (0): when nlcd_disable_scrolling() is called.

static unsigned char disp_buffer[]

Holds the last sixteen characters received. This buffer can be classified as a circular or ring buffer: its contents initially start at index zero and occupy sequential array positions for sixteen full characters. The current contents' start position then moves around the buffer incrementing (modulo 16) with each character (except the backspace). A modulo operation (%16) on the index in both printing and storing allows a small sixteen-character array to be recycled continuously since scrolling is not bi-directional (it is forward-only as previously mentioned) and does not need to keep track of characters seen in the non-displayable past. Note that this buffer is only used for printing after sixteen characters are already displayed on the LCD.

Scrolling

One of the features of this LCD driver is that it allows for screen to scroll on character input. For example, the user might type the following

```
|----------------|
|MAKE A U-TURN   |
|----------------|
```

This string utilizes only thirteen of the display positions, leaving three empty. Now suppose the user wants to append “`AT D ST`” to the end of this string, to make his directive clearer. Without scrolling enabled, the following will be displayed after typing the appended string:

```
|----------------|
|MAKE A U-TURN AT|
|----------------|
```

Only sixteen characters can be displayed at a time, so the appended text’s meaning has been lost entirely. Scrolling serves to solve problems like these where a few more letters of text are necessary to make the meaning clear or to complete the word. With scrolling enabled, appending that string would have yielded:

```
|----------------|
|A U-TURN AT D ST|
|----------------|
```

With scrolling the meaning has been saved.

How Scrolling Actually Works

A type of hybrid system is used, where, for the first sixteen characters entered and whenever a backspace is entered, the driver functions the same whether or not scrolling is enabled. Typing “`HELLO`

WORLD" prints exactly that on the screen. Then typing "123" appends those three numbers to the end. Typing a backspace clears the last character entered. In other words, as long as the user's input stays within the sixteen displayable positions on the LCD, the output on the screen will be consistent with anything the user has seen with a word processor or terminal.

With scrolling enabled and the screen already with of sixteen characters, the following occurs:

1. The character is a backspace: the screen backspaces as normal, removing the sixteenth character from the screen.
2. The character is not a backspace:
    a. The character is placed into a buffer (so it can be used in future scrolling).
    b. The screen is cleared (not wiped: this would reset the buffer).
    c. The entire buffered string is printed to the screen, one character at a time. Thus the screen gives the appearance of scrolling even though it is not. There is a noticeable flicker, but this flicker is not much worse than what would occur if the controller chip scrolled properly (this is known from running scrolling on only the first eight characters.)

This procedure allows for the seamless transition from scrolling mode back to non-scrolling mode if it is either disabled or the backspace key is hit, thus reducing the number of characters to be re-displayed on the screen for any keystroke.

Note: no matter which way scrolling was implemented, whether done by the driver or the controller, the nature of the screen necessitates a re-write of the entire string. Our implementation is likely slower and more noticeably flicker-prone than code that would do this at a lower level (controller-level).

The seamless transition from non-scrolling mode back to scrolling mode (not to be confused with scrolling enabled or disabled, which is set by a function call) is done to:

1. Reduce the burden on the LCD driver to print and reprint strings and instead rely on the controller to do the work (probably in a more efficient manner).
2. Reduce waste by drawing less power (see 1).
3. Avoid the flicker effect for a large portion of LCD usage, reducing the strain on the user's vision and making the LCD seem more refined.

Also, it really wasn't that difficult to implement since the next_pos variable was already created for printing without scrolling.

Rationale for this Implementation of Scrolling

Also called "shifting" by the LCD driver and controller, scrolling is supposed to be implemented by a simple set of commands to the LCD that would enable, disable, and control the direction of the scrolling. However, as mentioned before, there is a problem with the compatibility between the controller and the LCD hardware: the controller assumes two eight-character lines whereas the LCD displays a single sixteen-character line. Thus, the display data RAM within the controller looks like the following (values are in hexadecimal):

```
|-----------------------------------------------------------------------------|
| 00 | 01 | ... | 07 | 08 | ... | 27 |xxx| 40 | 41 | ... | 47 | 48 | ... | 67 |
|-----------------------------------------------------------------------------|
{                    }                /  {                    }
 \                  /                /    \                  /
 Initially displayed                /      Initially displayed
                                   /
           No DDRAM between 0x27 and 0x40.
```

The controller chip assumes two lines of 40 bytes. However, the LCD display hardware simply places these two groups of RAM into a single line. This is what will cause the problems that necessitate unusual fixes for not only scrolling, but also for simply writing to all sixteen characters onto the screen.

Writing the first eight characters to be displayed is no problem. Writing a character to the LCD moves the LCD's internal character address position, so writing a second character writes to the second position on the screen, as expected. However, once the first eight characters have been written, the first (internal) line of characters has been effectively filled. But the display shows another eight character spaces left after this first eight entered. So typing a ninth would seem to be no problem. Except that the ninth DDRAM position is not actually inside the displayable memory array (0x08 above): we have to jump to position 0x40. To get to the next displayable position requires sending a command to the LCD to move the address pointer to that position, and then typing can resume.

So what happens when scrolling/shifting is enabled? Because the controller assumes two lines, it shifts both sets of DDRAM. This effectively turns the displayed portion into:

```
|-----------------------------------------------------------------------------|
| 00 | 01 | ... | 07 | 08 | ... | 27 |xxx| 40 | 41 | ... | 47 | 48 | ... | 67 |
|-----------------------------------------------------------------------------|
     {                    }           /      {                    }
      \                  /           /        \                  /
      Displayed after shift         /        Displayed after shift
                                   /
            No DDRAM between 0x27 and 0x40.
```

Given the way we have written characters to the LCD, this poses a major problem: primarily that we will essentially "lose" the character (i.e. it disappears from the LCD) in 0x00. After a shift, there will be seven characters followed by one blank (or whatever vestige character is left in position 0x08), followed by seven characters and the latest character typed.

There were a number of attempted or proposed solutions to this problem, as follows:

1. Make DDRAM 0x08 - 0x27 and 0x40 - 0x60 be identical sets of characters. This would effectively reduce the maximum input to 8 + 32 + 8 = 48 characters, but would allow scrolling to work seamlessly inside that range. This entails writing to position 0x08 and 0x40 simultaneously, 0x09 and 0x41 simultaneously, and so on. Obviously, simultaneous writes are not possible, so one will have to be written before the other. The options are to:
    a. Write the lower position first, then the upper position.
    b. Write the upper position first, then the lower position.

    Each of these approaches was attempted, but neither with success. While it is possible that the code was in error, and that one of these approaches is in fact feasible, we were unable to implement it without running into a number of problems, all derivatives of the following. Writing one character and then jumping back or forward to the other position caused the LCD's internal address pointer to become incorrect. Mainly what would occur is that while the first character would work, the second would not due to the address pointer setting to 0x40 regardless of the command sent.

    We suspect that this approach is possible, but after multiple failed attempts at this implementation timeliness of the delivery of this system necessitated the approach to be abandoned in favor of one that could be done quickly.

2. Hold all the received characters in a large character array inside of **nlcd.c** and print from the array as necessary. A similar approach was actually chosen to be implemented. The problem with this approach is the large amount of RAM that is consumed by a large array that is mostly unused. This wouldn't normally be a major obstacle with the abundance of RAM on most microcontrollers today, but our simple microcontroller had a relatively small amount of RAM and our keyboard driver already used a large portion of it for scancode tables.

3. Have the calling code (**comaidsystem.c** or one of its derivatives) keep its own buffer of characters received from the keyboard input and then print a string of the sixteen characters to be displayed, on its own terms. This would be easy to implement, but does not make nlcd.c nearly as useful a driver for future applications. This was not attempted and was considered only as a last resort.

Functions

Both sets of functions below are described in the same order that they are listed in **nlcd.c**.

Driver API

int nlcd_init (void)

This initialization function is based on page 12 of the LCD datasheet. First, the driver directions (write) for the microcontroller are set for the operational enable signal and register select signal pins. The macros DDRX and PORTX_E and _RS are used to allow for the simple transition of these pins to different pins on the microcontroller. If the pins were to be used on different ports instead of on the same one, adjustments could be made with a similar macro. The driver directions (write) are also set for the four data pins, again aided by a macro for transition to different pins.

The next group of lines, delays and commands, are the portion based on the LCD datasheet. The screen initialization is also done with macros.

Once the initialization is complete, the screen prints the project name and version (currently "Comaidsystem 1.0") for approximately 1.5 seconds, and then proceeds to clear the screen, indicating it is ready for use.

Note that scrolling is disabled by default: scrolling must be enabled by the API user.

void nlcd_char (unsigned char)

The most utilized function, it prints a single character to the screen. If the character is a backspace, it needs some special handling (see the notes about scrolling below and the function nlcd_backspace()). Otherwise, the character is placed into a buffer and, depending on the mode, does one of three things:

1. Scrolling has been enabled and the screen is already full: calls nlcd_bstring() to print the buffered display string.
2. Scrolling has not been enabled and the screen is already full: does nothing. The screen remains the same.
3. The screen is not yet full: merely print the character and increment the next_pos pointer to the next display position.

void nlcd_string (const char*)

Constant strings are located in program memory (i.e. flash memory), requiring the use of pgm_read_byte() for proper access. Merely passes the characters to nlcd_char() for handling. The screen is wiped due to nlcd_string() being used primarily for a macro of some sort ("Turn Left" for example), which is more sensible when used with no preceding garbage on the display.

void nlcd_vstring (unsigned char*)

A variable string can be printed (not one from program memory). Merely passes the characters to nlcd_char() for handling. The screen is wiped due to nlcd_vstring() being used primarily for a macro of some sort ("Turn Left" for example), which is more sensible when used with no preceding garbage on the display. This function is needed for writing strings from EEPROM and RAM.

void nlcd_flash (int)

Enables the user to flash the screen for a number of seconds. Useful for alerting the driver (or other LCD user) that there is new text on the screen. The option of having the screen flash until notified to stop, i.e. by character input, was not implemented here, but would be simple to do with the use of a single flash function that was turned on and off using the while loop that runs the system.

void nlcd_wipe (void)

Clears the LCD screen as well as resetting the buffer. If the buffer is not to be reset, nlcd_clear() should be used instead.

void nlcd_enable_scrolling (void)

Exactly as it sounds. The only caveat here is that if scrolling was previously not enabled and the LCD is currently full, a write is done to start up the scrolling.

void nlcd_disable_scrolling (void)

Deactivates the variable determining scrolling.

Internal Functions

void nlcd_command (unsigned char)

Sends a command to the LCD, first by changing to command setting and then sending the character.

void nlcd_inc_pos (void)

For tracking the next_pos variable as well as moving the DDRAM address across the split in RAM when necessary.

void nlcd_dec_pos (void)

For tracking the next_pos variable as well as moving the DDRAM address back across the split in RAM when necessary.

void nlcd_nibble (void)

Sends a rising edge along the operation enable pin, pauses, then sends a falling edge, creating a square wave indicating data is ready to be taken.

void nlcd_place_data (unsigned char)

Places the four bits of data into the data port lines.

void nlcd_put_data (unsigned char)

This encapsulates the entire transmission of the eight bits of data to be sent to the LCD, whether they make a command or a character. Sends four bits at a time and clocks the bits with nlcd_place() and nlcd_nibble().

void nlcd_set_command (void)

Sets the RS bit low to indicate the next byte is a command.

void nlcd_set_data (void)

Sets the RS bit high to indicate the next byte is data for the LCD.

void nlcd_bstring (void)

An internal string printing function for use with the way scrolling is implemented. Clears the screen, then prints the current buffer character by character, filling the screen. Jumps across the gap in DDRAM after half the string has been printed.

void nlcd_clear (void)

Also called by nlcd_wipe(), this function simply clears the screen and the current DDRAM but does not affect the internal buffer.

void nlcd_backspace (void)

The backspace function that takes into consideration the current position in the display. If trying to backspace when there is nothing on the screen, nothing will be done. Otherwise, the backspacing handles the gap in DDRAM with a couple of next_pos variable checks.

The backspacing itself is a different process. There is no direct way that the LCD controller allows the user to perform a "backspace"; that is, to remove the most recent character typed in. Thus we implement the backspacing function first by moving the cursor left a space (to the most recent character typed), then by writing a blank to this position. However, this causes the cursor to move forward once more since we have written a character. This is handled by simply sending a command to move the cursor back left.

An alternate solution considered was to disable the cursor's movement on the input of a character. However, the controller chip does not seem to have a command that allows for that.

The display buffer is also updated with the blank space, and the position counter decremented as expected when moving one position left on the screen.

Etymology

The name "nlcd" is short for "new lcd". The ComAidSystem project was started using a different LCD and its driver was named "lcd.c". Then, both LCDs were used concurrently for debugging purposes while making the new LCD functional, so the new LCD's code had to be of a different name. The name was never reverted intentionally, for the designers chose to retroactively rationalize the name as being short for "Newhaven lcd".

Last Updated: 19 May 2010