CAB202 Topic 8 – LCD

Luis Mejias & Lawrence Buckingham, Queensland University of Technology.

Contents

Roadmap

Last week:

7. Teensy – Introduction to Microcontrollers; Digital Input/Output; Bitwise operations.

This week:

8. **LCD Display – sending digital signals to a device; directly controlling the LCD display.**

Still to come:

9. Debouncing, Timers and Interrupts – asynchronous programming.
10. Serial Communication – communicating with another computer.
11. Analogue to Digital Conversion; Pulse Width Modulation.
12. Assignment 2 Q&A.

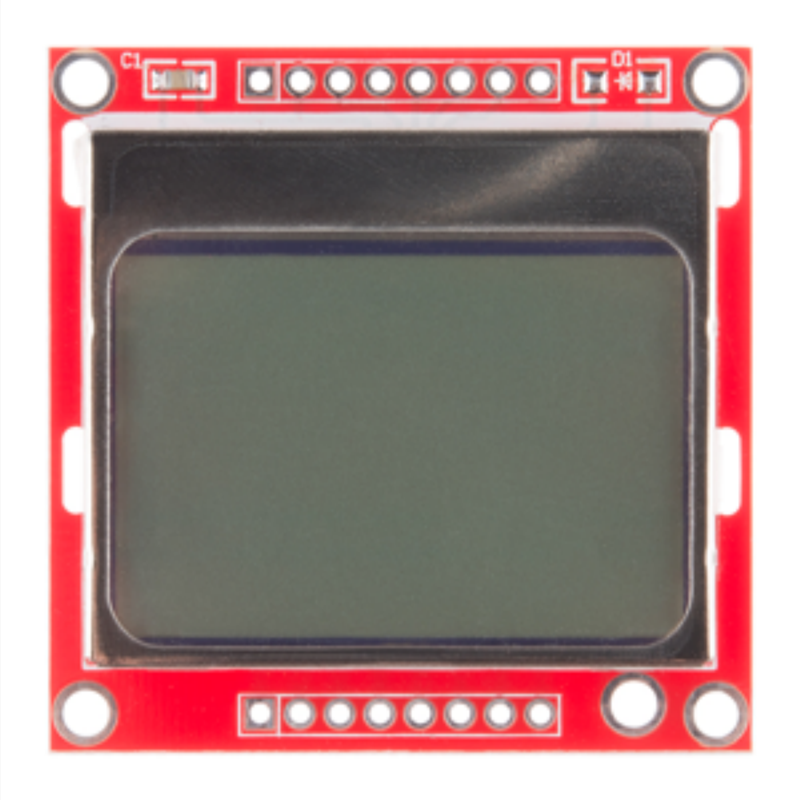
References

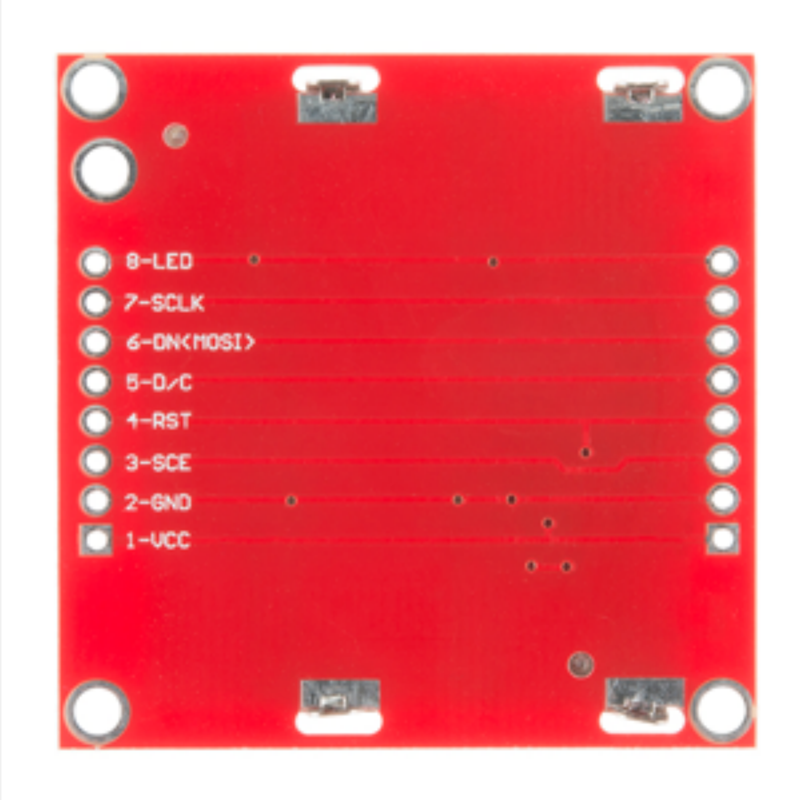
Recommended reading:

- Blackboard→Learning Resources→Microcontrollers→Nokia5110-LCD-Screen.pdf (PCD8544 Data Sheet).
- Blackboard→Learning Resources→Microcontrollers→TeensyPewPew Schematic.pdf

---

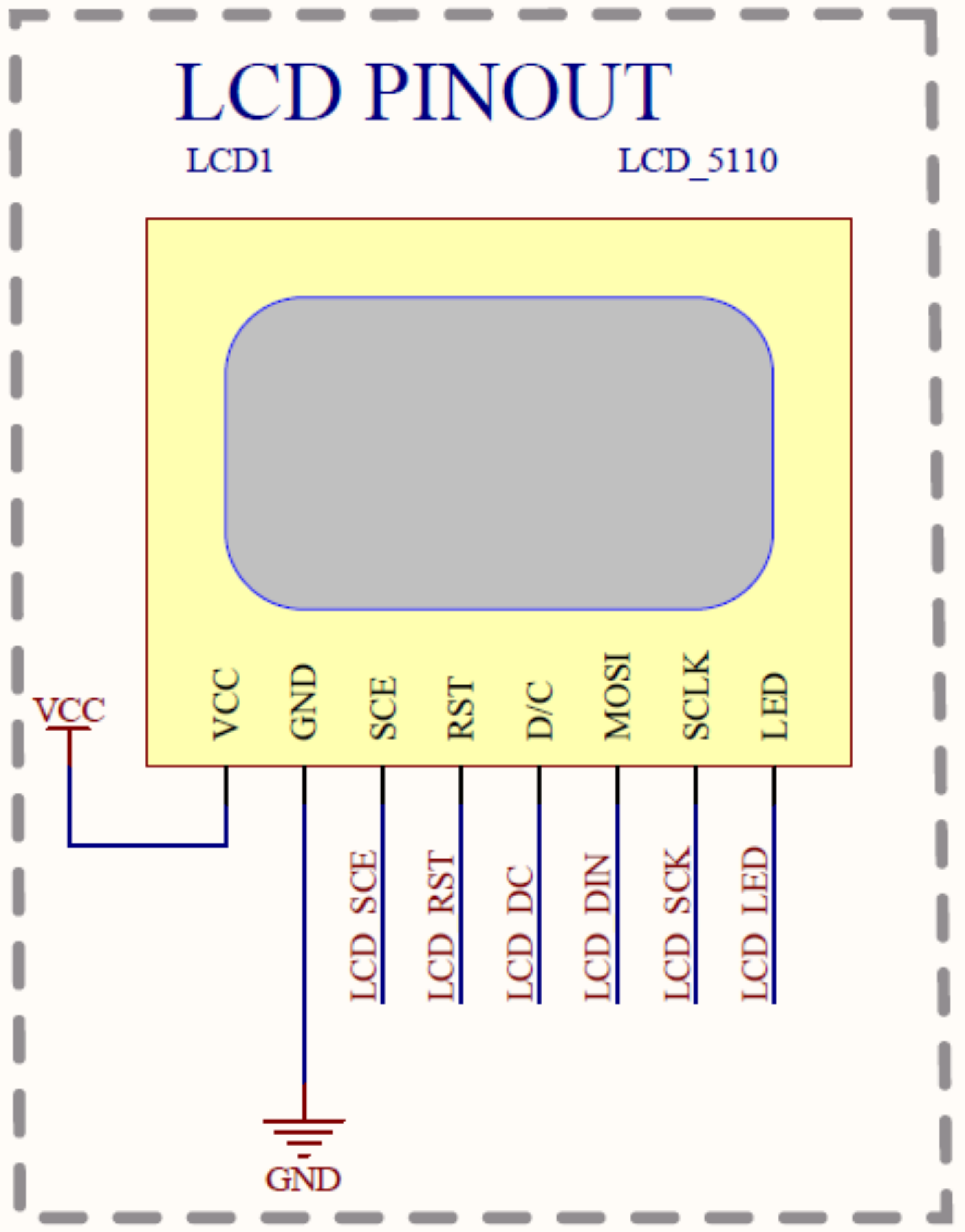
The PCD8544 LCD controller/driver





- *Refer: LCD data sheet, p3*.
- Low-power LCD controller.
- 48×84 pixel monochrome display.
- Build-in back-light.
- Interfaces to microcontrollers via serial bus interface.
    - Data flow is strictly unidirectional, from microcontroller to LCD.
- The controller has a small amount of RAM which holds the pixel data for display.

LCD Interface (From the LCD Point of View)



LCD has 8 externally accessible pins

- 6 of which are used to control the peripheral.
- the other two are the power connections: VCC and GND.
- *Refer: LCD data sheet, p5*.

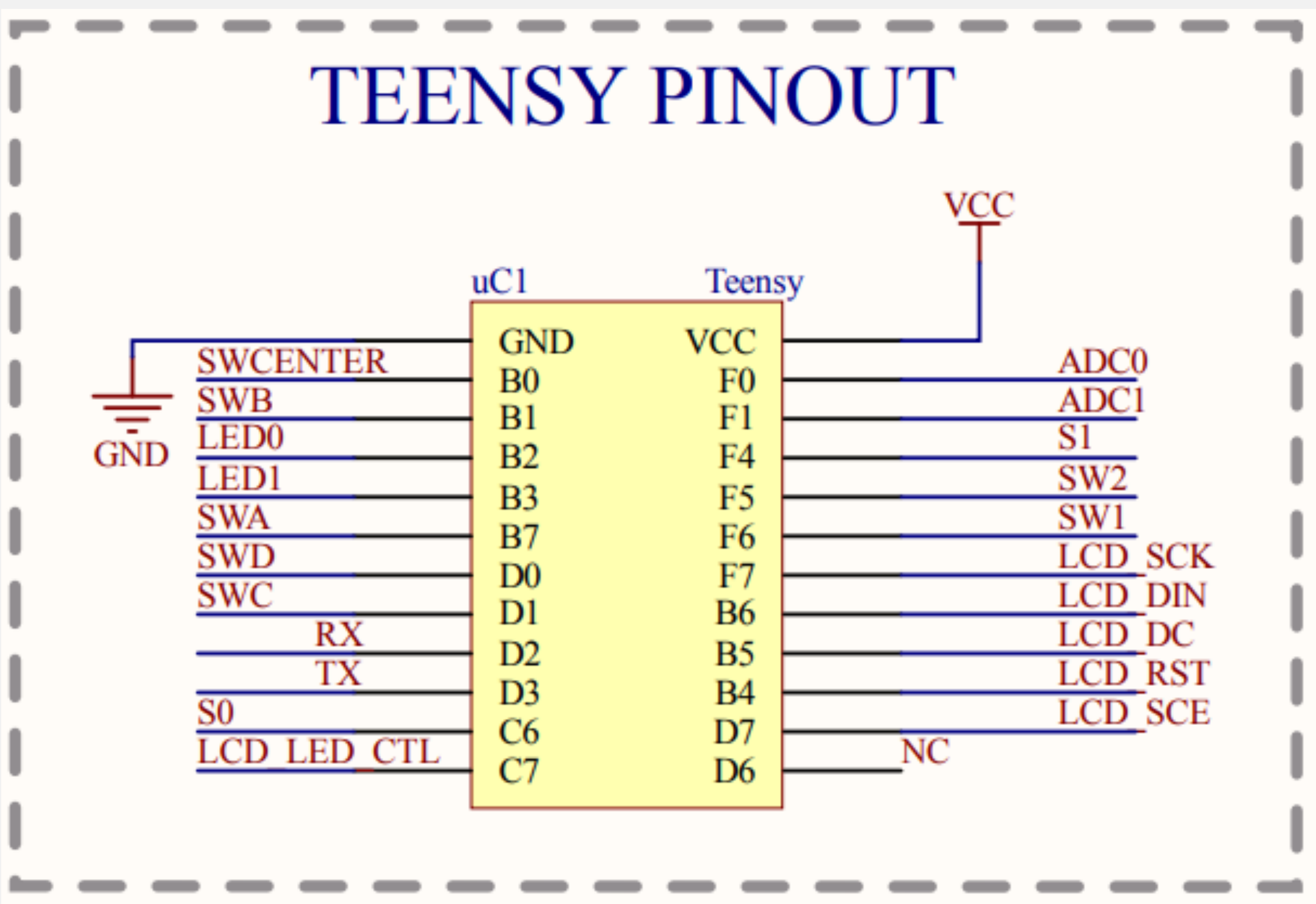
Pins we use:

1. **SCE** – Chip Select Pin

- Tells the LCD there is incoming data.
- Active when 0.

2. **RST** – Reset Pin
    - Resets the LCD to its default configuration when switched to 0.
    - To resume operation switch back to 1.
    - The old standby: “turn it off and on again”.
3. **D/C** – Data/Command Pin
    - 0 means incoming data must be interpreted as a command.
    - 1 means incoming data is pixel data to be displayed.
    - *Also called DC.*
4. **DIN** – Serial Data Input Pin
    - Data is transmitted to the LCD one bit at a time over this pin.
    - Data is interpreted as pixels if D/C == 1.
    - Data is interpreted as command if D/C == 0.
    - *Also called MOSI, SDIN.*
5. **SCK** – Serial Clock Pin
    - Toggled from 0 to 1 during data transfer to signal that a bit is available on DIN.
    - *Also called SCLK.*
6. **LED** – Back-light
    - When 1, turns on the back-light LED.

LCD Interface from the Teensy Point of View

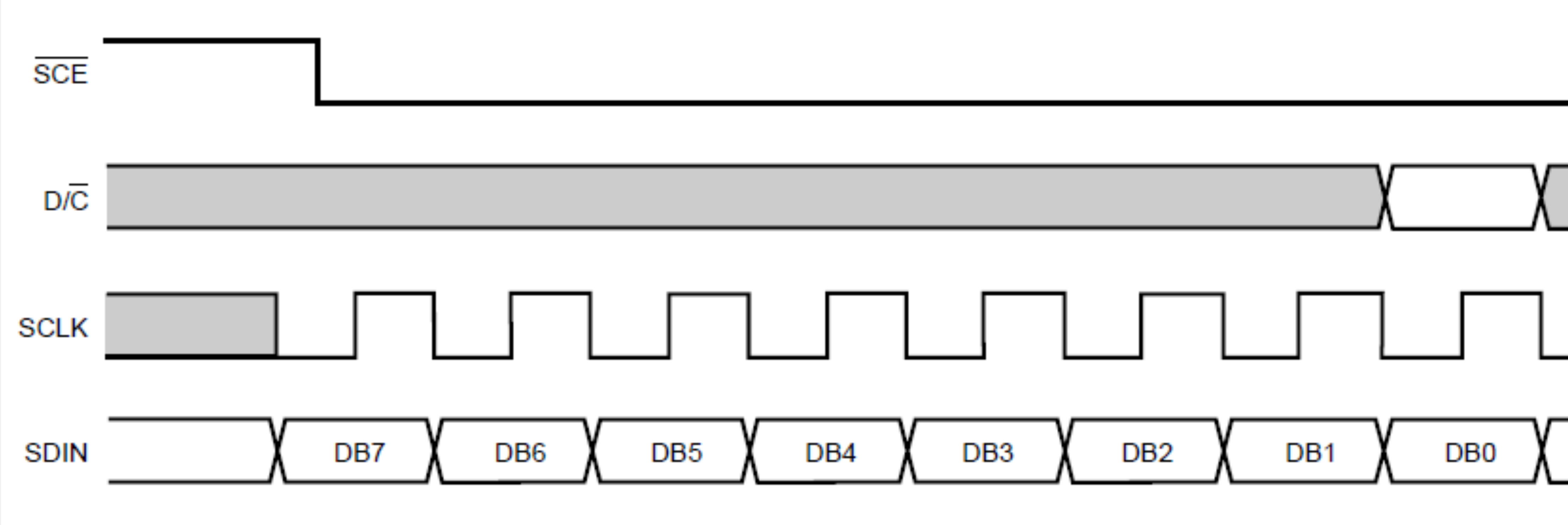


At the Teensy, the LCD pin mappings are:

- Port C, pin 7 → LCD backlight.
- Port F, pin 7 → LCD Serial Clock pin.
- Port B, pin 6 → LCD Serial Data Input pin.
- Port B, pin 5 → LCD Serial Data/Command pin.
- Port B, pin 4 → LCD Reset pin.
- Port D, pin 7 → LCD Chip Select pin.

Transmitting Data From Teensy to LCD

Transmitting a byte to the LCD involves four pins:



Refer: LCD data sheet, Fig 10, p12

Data transmission is done by the function **lcd_write** in **lcd.c**. This function sends a single byte to the LCD:

```
/*
 *  CAB202 Teensy Library (cab202_teensy)
 *      lcd.c
 *
 *      Michael, 32/13/2015 12:34:56 AM
 *
 */
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <util/delay.h>

#include "lcd.h"
#include "ascii_font.h"
#include "macros.h"

/*
 * Function implementations
 */
void lcd_init(uint8_t contrast) {
        // Set up the pins connected to the LCD as outputs
        SET_OUTPUT(DDRD, SCEPIN);
        SET_OUTPUT(DDRB, RSTPIN);
        SET_OUTPUT(DDRB, DCPIN);
        SET_OUTPUT(DDRB, DINPIN);
        SET_OUTPUT(DDRF, SCKPIN);

        CLEAR_BIT(PORTB, RSTPIN);
        SET_BIT(PORTD, SCEPIN);
        SET_BIT(PORTB, RSTPIN);

        lcd_write(LCD_C, 0x21); // Enable LCD extended command set
        lcd_write(LCD_C, 0x80 | contrast ); // Set LCD Vop (Contrast)
        lcd_write(LCD_C, 0x04);
        lcd_write(LCD_C, 0x13); // LCD bias mode 1:48

        lcd_write(LCD_C, 0x0C); // LCD in normal mode.
        lcd_write(LCD_C, 0x20); // Enable LCD basic command set
        lcd_write(LCD_C, 0x0C);

        lcd_write(LCD_C, 0x40); // Reset row to 0
        lcd_write(LCD_C, 0x80); // Reset column to 0
}

void lcd_write(uint8_t dc, uint8_t data) {
        // Set the DC pin based on the parameter 'dc' (Hint: use the WRITE_BIT macro)
        WRITE_BIT(PORTB,DCPIN,dc);

        // Pull the SCE/SS pin low to signal the LCD we have data
        CLEAR_BIT(PORTD,SCEPIN);

        // Write the byte of data using "bit bashing"
        for(int i = 7; i >= 0; i--) {
                CLEAR_BIT(PORTF, SCKPIN) ;
                if((data>>i) & (1 == 1)) {
                        SET_BIT(PORTB, DINPIN);
                } else {
                        CLEAR_BIT(PORTB, DINPIN);
                }
                SET_BIT(PORTF, SCKPIN);
        }

        // Pull SCE/SS high to signal the LCD we are done
        SET_BIT(PORTD, SCEPIN);
}

void lcd_clear(void) {
```

```
        // For each of the bytes on the screen, write an empty byte
        // We don't need to start from the start: bonus question - why not?
        for (int i = 0; i < LCD_X * LCD_Y / 8; i++) {
                lcd_write(LCD_D, 0x00);
        }
}

void lcd_position(uint8_t x, uint8_t y) {
        lcd_write(LCD_C, (0x40 | y )); // Reset row to 0
        lcd_write(LCD_C, (0x80 | x )); // Reset column to 0
}
```

We can see a direct correlation between the code and the diagram:

- Parameters are:
  - **`dc`** – a byte, which should be either 0 or 1.
  - **`data`** – and arbitrary byte. It may contain either a command (when **`dc == 0`**), or pixel data (when **`dc == 1`**).
- Before we send any bits, the D/C pin is assigned the value.
  - LCD reads the D/C pin while the last (8th) bit is being read (Data sheet, p11).
  - Setting it early does no harm… the value is ready in plenty of time.
- Next, 0 is written into the Chip Select pin, telling the LCD that data is on the way.
- The 8 bits are then sent, one at a time, starting with the most significant bit (bit 7). For each bit:
  - 0 is written to the Serial Clock pin.
  - The current bit is written to the Serial Data Input pin.
  - 1 is written to the Serial Clock pin. This tells the LCD to read the bit.
- After the bits are written, the Chip Select pin is set back to 1 to tell the LCD there is no more data.

Programming the LCD

The LCD instruction set is set out in Section 8 of the data sheet (pages 14–16). We reproduce Tables 1 and 2 here for convenience. Unused operations have been eliminated from Table 1.

INSTRUCTION	D/C	COMMAND BYTE								DESCRIPTION
		DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
(H = 0 or 1) *Either instruction set*										
NOP	0	0	0	0	0	0	0	0	0	no operation
Function set	0	0	0	1	0	0	PD	V	H	power down control; entry mode; extended instruction set control (H)
Write data	1	D7	D6	D5	D4	D3	D2	D1	D0	writes data to display RAM
(H = 0) *Basic instruction set*										
Display control	0	0	0	0	0	1	D	0	E	sets display configuration
Set Y address of RAM	0	0	1	0	0	0	Y2	Y1	Y0	sets Y-address of RAM; 0 ≤ Y ≤ 5
Set X address of RAM	0	1	X6	X5	X4	X3	X2	X1	X0	sets X-address part of RAM; 0 ≤ X ≤ 83
(H = 1) *Extended instruction set*										
Temperature control	0	0	0	0	0	0	1	TC1	TC0	set Temperature Coefficient (TCx)
Bias system	0	0	0	0	1	0	BS2	BS1	BS0	set Bias System (BSx). *Determined by number of multiplexed planes. This unit has MUX == 1:48, so the appropriate value for Bias is 3. See page 16 of data sheet.*
Set VOP	0	1	VOP6	VOP5	VOP4	VOP3	VOP2	VOP1	VOP0	write VOP to register; sets the

										operating voltage; $0 \le VOP \le 127$ == 0x7f. *Provides adjustable contrast*.

Interpretation:

BIT	0	1
PD	chip is active	chip is in Power-down mode
V	horizontal addressing	vertical addressing
H	use basic instruction set	use extended instruction set
D and E 00 10 01 11	 display blank normal mode all display segments on inverse video mode	
TC1 and TC0 00 01 10 11	 VLCD temperature coefficient 0 VLCD temperature coefficient 1 VLCD temperature coefficient 2 VLCD temperature coefficient 3	

The information from this table has been distilled into **`lcd_model.h`**. The enumerated types and macros will probably be helpful fro everybody; the Nokia5110 structure is there for people who are interested in creating software emulations of the LCD.

```
/*
** Nokia 5110 LCD logical model.
**
**      Enumerated types and macros to support learning about the LCD.
**
**      Lawrence Buckingham, 10 Sep 2017.
**      (C) Queensland University of Technology, Brisbane, Australia.
*/

#pragma once
#include <stdint.h>

#if ! defined(LCD_X)
#define LCD_X 84
#endif

#if ! defined(LCD_Y)
#define LCD_Y 48
#endif

// Issue a command to the Nokia5110.
#define LCD_CMD( op_code, args ) lcd_write(0,op_code|args)

// Send a byte of pixel data to the Nokia5110.
#define LCD_DATA( args )         lcd_write(1,args)

// Valid op_codes for Nokia5110.
typedef enum lcd_op_code_t {
        lcd_nop              = 0,
        lcd_set_function     = 1 << 5,

        // When lcd_instr_basic has been selected, these op_codes are available.
        lcd_set_display_mode = 1 << 3,
        lcd_set_y_addr       = 1 << 6,
        lcd_set_x_addr       = 1 << 7,

        // When lcd_instr_extended has been selected, these op_codes are available.
        lcd_set_temp_coeff   = 1 << 2,
        lcd_set_bias         = 1 << 4,
        lcd_set_contrast     = 1 << 7,
} lcd_op_code_t;

// Arguments for lcd_set_function
typedef enum lcd_power_mode_t {
        lcd_chip_active = 0,
        lcd_power_down  = 1 << 2,
} lcd_power_mode_t;

// Arguments for lcd_set_function
typedef enum lcd_addressing_mode_t {
        lcd_addr_horizontal = 0,
        lcd_addr_vertical   = 1 << 1,
} lcd_addressing_mode_t;
```

```c
// Arguments for lcd_set_function
typedef enum lcd_instruction_mode_t {
        lcd_instr_basic    = 0,
        lcd_instr_extended = 1
} lcd_instruction_mode_t;

// Arguments for lcd_set_display_mode
typedef enum lcd_display_mode_t {
        lcd_display_all_off = 0b000, // decimal 0
        lcd_display_all_on  = 0b001, // decimal 1
        lcd_display_normal  = 0b100, // decimal 4
        lcd_display_inverse = 0b101, // decimal 5
} lcd_display_mode_t;

// Arguments for lcd_set_y_addr: numeric values from 0 to 5.
// Arguments for lcd_set_x_addr: numeric values from 0 to 83.
// Arguments for lcd_set_temp_coeff: numeric values from 0 to 3.
// Arguments for lcd_set_bias: numeric values from 0 to 7. Ref Data Sheet.
// Arguments for lcd_set_contrast: numeric VOP values from 0 to 127.

/*
**      A logical model of the LCD state machine.
**
**      !!! This is not for use in Teensy programs      !!!
**      !!! But it can be used to implement an emulator !!!
**
*/

typedef struct Nokia5110_t {
        // PowerDown mode
        lcd_power_mode_t powerMode;

        // Instruction set
        lcd_instruction_mode_t instructionSet;

        // LCD contrast (V_{OP} setting. Valid values 0 .. 0x7f
        uint8_t contrast;

        // LCD Bias. Valid values 0 .. 7
        uint8_t bias;

        // LCD Temperature Coefficient. Valid values 0.. 0x03.
        uint8_t temperatureCoefficient;

        // Display mode.
        lcd_display_mode_t displayMode;

        // Addressing mode (cursor direction control)
        lcd_addressing_mode_t addressing;

        // Current horizontal position of data cursor (0..83).
        // When a data byte is received, the byte is inserted at pixels[x][y],
        // after which x and/or y are updated according to the addressing mode.
        uint8_t x;

        // Current vertical position of data cursor (0..5)
        // When a data byte is received, the byte is inserted at pixels[x][y],
        // after which x and/or y are updated according to the addressing mode.
        uint8_t y;

        // Pixel data 2D array
        uint8_t pixels[LCD_X][LCD_Y / 8];

} Nokia5110_t;
```

Every LCD command (other than writing pixel data) is an 8-bit value with two parts:

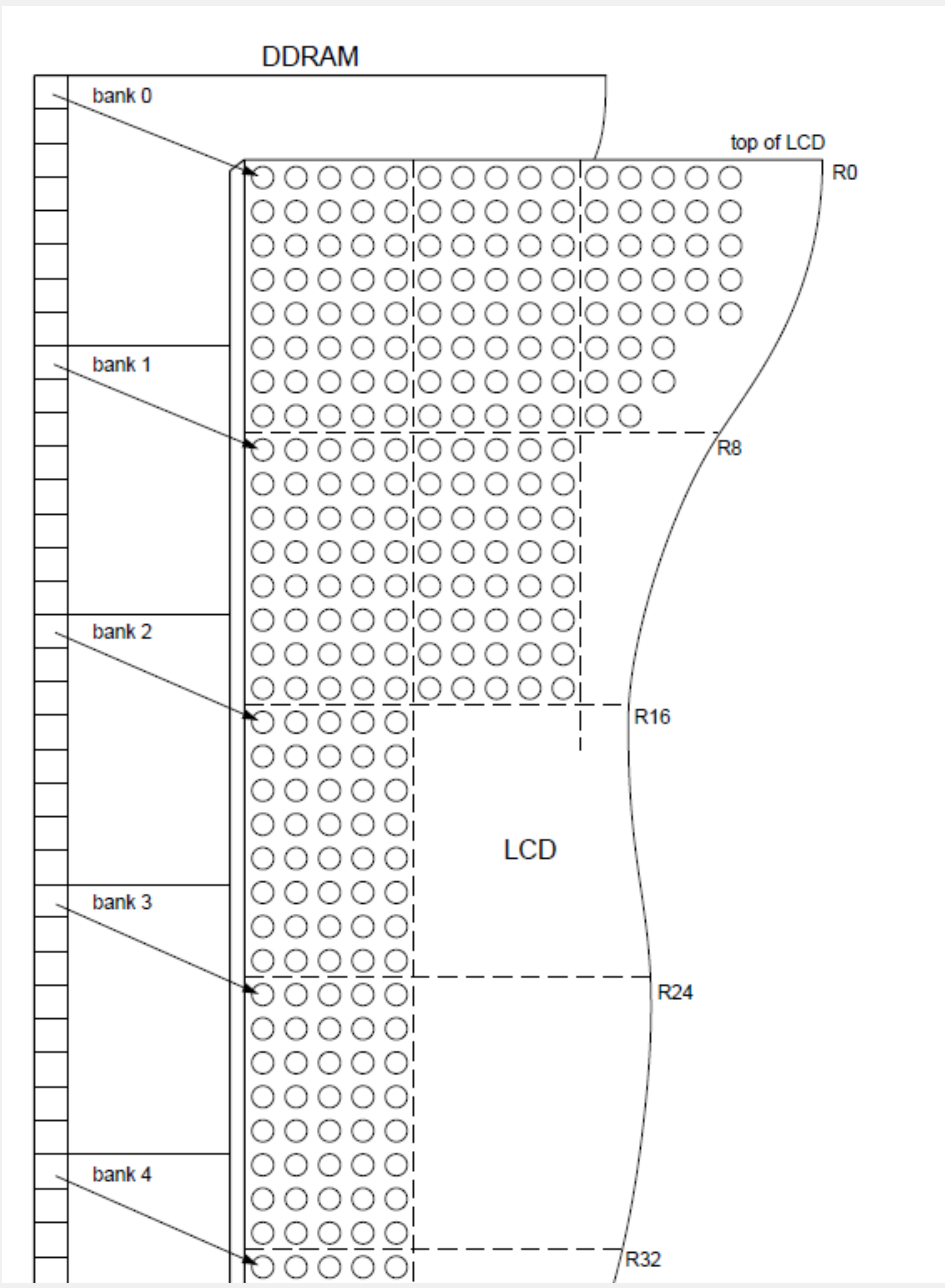
- `COMMAND = OP_CODE | ARGS` or perhaps `COMMAND = OP_CODE | ARG1 | ARG2 | …`
- `OP_CODE == ` ***`operation code`***.
- PCD8544 recognises 8 operation codes – defined in `lcd_op_code_t`.
- The operation codes are classified as *general*, *basic*, or *extended*.
  - General commands can be sent at any time. They are `lcd_nop` and `lcd_set_function`.
  - Basic commands are `lcd_set_display_mode`, `lcd_set_y_addr`, and `lcd_set_x_addr`. These are called frequently.
  - Extended commands are `lcd_set_temp_coeff`, `lcd_set_bias`, and `lcd_set_contrast`. These are called less frequently.
- Each operation code has a particular set of values that are meaningful. They are modelled where possible by an enumerated type.

- `lcd_nop` → nothing.
- `lcd_set_function` → up to three arguments: `lcd_power_mode_t`, `lcd_addressing_mode_t`, and `lcd_instruction_mode_t`.
- `lcd_set_display_mode` → `lcd_display_mode_t`
- `lcd_set_y_addr` → `0..5`
- `lcd_set_x_addr` → `0..83`
- `lcd_set_temp_coeff` → `0..3`
- `lcd_set_bias` → `0..7`
- `lcd_set_contrast` → `0..127`

Pixel data storage

The LCD contains 504 bytes of display data RAM which is organised as 6 *banks*, each of which has 84 bytes of storage.

- Each bank contains a horizontal band of pixels which stretches from the left to right side of the display.
- The pixels are arranged in vertical blocks of 8, and each block of 8 pixels is packed into a byte.
- Every time we write data to the LCD display we replace a complete block of 8 pixels.



- Although reality may be different, it is useful to think of the LCD pixel data storage as a system with three variables:
  `uint8_t x; // Horizontal cursor position, ranging from 0 to 83`
  `uint8_t y; // Vertical cursor position, ranging from 0 to 5`
  `uint8_t pixels[6][84]; // Pixel data, indexed by (y,x)`

- Here **x** and **y** specify where the next byte of pixel data will be placed when it arrives.
  - `(y,x) == (0,0)` is the top left corner of the display.
  - The values of **x** and **y** can be set via `lcd_set_y_addr` and `lcd_set_x_addr` commands.
  - After each byte of pixel data arrives, **x**, **y**, or possibly both, change.
  - If horizontal addressing is active, then **x** increments after each byte is drawn. When **x** reaches 84, it wraps back to 0, and **y** increments. When **y** reaches 6, it wraps back to 0.
  - If vertical addressing is active, then **y** increments after each byte is drawn. When **y** reaches 6, it wraps back to 0, and **x** increments. When **x** reaches 84, it wraps back to 0.
  - If 504 bytes are written, the cursor will return to its original location (because it will have travelled over every block in every bank in the display).
- 8 pixels are packed into each byte of display memory.
  - Pixel data is sent in batches of 8 bits, so 8 pixels are overwritten each time pixel data is sent.
  - The pixels in a block are laid out vertically on the physical display, with the pixel corresponding to the least significant bit (the "ones") appearing at the top of the bank. So a block containing bits:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

will be drawn to the LCD as below:

b_0
b_1
b_2
b_3
b_4
b_5
b_6
b_7

(Remember, ($b_i \in \{0, 1\}$).)

  - There is no mechanism to combine new pixels with old.
    - If you want to blend new content in with old, all processing must be done before the pixels are sent to the LCD display.
    - Depending on the kind of drawing tasks that must be done, it may be necessary to keep a screen buffer in microcontroller RAM, and periodically flush it to the LCD.
    - This is what we do with `cab202_teensy/graphics.c`.
  - To write an 8-bit block of pixel data `(pixel_block)` at screen coordinates `(px,py)`:
    - Get the cursor position:

```
x = px;
y = py / 8;
```

    - Move LCD internal cursor:

```
LCD_CMD(lcd_set_function, lcd_instr_basic | lcd_addr_horizontal);
LCD_CMD(lcd_set_x_addr, x);
LCD_CMD(lcd_set_y_addr, y);
```

    - Write the byte value:

```
LCD_DATA(pixel_block);
```

Case study 1: `lcd_init`

The `lcd_init` function in the library was written with no regard for readability. In the following code listing (`TeensyLines.c`) it is rewritten using more comprehensible notation.

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>

#include <graphics.h>
#include <macros.h>

#include "lcd_model.h"

void new_lcd_init(uint8_t contrast) {
	// Set up the pins connected to the LCD as outputs
	SET_OUTPUT(DDRD, SCEPIN); // Chip select -- when low, tells LCD we're sending data
	SET_OUTPUT(DDRB, RSTPIN); // Chip Reset
	SET_OUTPUT(DDRB, DCPIN);  // Data / Command selector
	SET_OUTPUT(DDRB, DINPIN); // Data input to LCD
	SET_OUTPUT(DDRF, SCKPIN); // Clock input to LCD

	CLEAR_BIT(PORTB, RSTPIN); // Reset LCD
	SET_BIT(PORTD, SCEPIN);   // Tell LCD we're not sending data.
	SET_BIT(PORTB, RSTPIN);   // Stop resetting LCD

	LCD_CMD(lcd_set_function, lcd_instr_extended);
	LCD_CMD(lcd_set_contrast, contrast);
	LCD_CMD(lcd_set_temp_coeff, 0);
	LCD_CMD(lcd_set_bias, 3);
	LCD_CMD(lcd_set_function, lcd_instr_basic);
	LCD_CMD(lcd_set_display_mode, lcd_display_normal);
	LCD_CMD(lcd_set_x_addr, 0);
	LCD_CMD(lcd_set_y_addr, 0);
}

void setup(void) {
	set_clock_speed(CPU_8MHz);
	new_lcd_init(LCD_DEFAULT_CONTRAST);
	clear_screen();
	show_screen();
}

void process(void) {
	static double fraction = 0.0;
	clear_screen();



	 for ( int y = 0; y < LCD_Y; y++ ) {
		draw_line(0, y, LCD_X - 1, y, FG_COLOUR);
	}


	int x1 = LCD_X * fraction;
	int x2 = LCD_X - x1;
	int y1 = LCD_Y * fraction;
	int y2 = LCD_Y - y1;

	fraction += 0.01;

	if ( fraction >= 0.5 ) fraction = 0.0;

	draw_line(x1, y1, x2, y1, BG_COLOUR);
	draw_line(x1, y2, x2, y2, BG_COLOUR);
	draw_line(x1, y1, x1, y2, BG_COLOUR);
	draw_line(x2, y1, x2, y2, BG_COLOUR);

	show_screen();
}

int main(void) {
	setup();

	for ( ;; ) {
		process();
		// _delay_ms(10);
	}
}
```

Worked example 2: Changing Contrast

Due to physical variations in the LCD it is a good idea to make it possible for the user to select a contrast level that suits the ambient lighting and temperature. In `ContrastDemo` we see how to adjust the contrast of the display.

```
#include <stdint.h>
```

```c
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>
#include <stdio.h>

#include <graphics.h>
#include <macros.h>

#include "lcd_model.h"

void new_lcd_init(uint8_t contrast) {
	// Set up the pins connected to the LCD as outputs
	SET_OUTPUT(DDRD, SCEPIN); // Chip select -- when low, tells LCD we're sending data
	SET_OUTPUT(DDRB, RSTPIN); // Chip Reset
	SET_OUTPUT(DDRB, DCPIN);  // Data / Command selector
	SET_OUTPUT(DDRB, DINPIN); // Data input to LCD
	SET_OUTPUT(DDRF, SCKPIN); // Clock input to LCD


	CLEAR_BIT(PORTB, RSTPIN); // Reset LCD
	SET_BIT(PORTD, SCEPIN);   // Tell LCD we're not sending data.
	SET_BIT(PORTB, RSTPIN);   // Stop resetting LCD

	LCD_CMD(lcd_set_function, lcd_instr_extended);
	LCD_CMD(lcd_set_contrast, contrast);
	LCD_CMD(lcd_set_temp_coeff, 0);
	LCD_CMD(lcd_set_bias, 3);

	LCD_CMD(lcd_set_function, lcd_instr_basic);
	LCD_CMD(lcd_set_display_mode, lcd_display_normal);
	LCD_CMD(lcd_set_x_addr, 0);
	LCD_CMD(lcd_set_y_addr, 0);
}

void setup(void) {
	set_clock_speed(CPU_8MHz);
	new_lcd_init(LCD_DEFAULT_CONTRAST);
	clear_screen();
	draw_string( 10, 10, "Hello Cab202!", FG_COLOUR );
	show_screen();
}

char buffer[10];

void draw_int(uint8_t x, uint8_t y, int t) {
	snprintf( buffer, 10, "%d", t );
	draw_string( x, y, buffer, FG_COLOUR );
}

void process(void) {
	static uint8_t contrast = 0;
	contrast ++;
	if ( contrast > 127 ) contrast = 0;

	draw_string( 10, 20, "          ", FG_COLOUR );
	draw_int( 10, 20, contrast );
	show_screen();

	LCD_CMD( lcd_set_function, lcd_instr_extended );
	LCD_CMD( lcd_set_contrast, contrast );
	LCD_CMD( lcd_set_function, lcd_instr_basic );
}

int main(void) {
	setup();

	for ( ;; ) {
		process();
		_delay_ms(50);
	}
}
```

Worked example 3: Direct screen write to the LCD

There may be situations where it makes sense to draw objects directly to the screen.

For example, some applications require text output only. Since sophisticated graphical effects are not required, we can use the memory occupied by the screen buffer for other purposes.

Another case would be where (for performance reasons) we wish to selectively update a small part of the screen but leave the remainder unchanged. This might be applicable for example if a sprite is moving over a static background.

`DirectDemo.c` shows how to write bit patterns directly to the LCD.

```c
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>
#include <stdio.h>
#include <stdlib.h>

#include <macros.h>
#include <lcd_model.h>
#include <ascii_font.h>
#include <graphics.h>

/*
**  Initialise the LCD display.
*/
void new_lcd_init(uint8_t contrast) {
    // Set up the pins connected to the LCD as outputs
    SET_OUTPUT(DDRD, SCEPIN); // Chip select -- when low, tells LCD we're sending data
    SET_OUTPUT(DDRB, RSTPIN); // Chip Reset
    SET_OUTPUT(DDRB, DCPIN);  // Data / Command selector
    SET_OUTPUT(DDRB, DINPIN); // Data input to LCD
    SET_OUTPUT(DDRF, SCKPIN); // Clock input to LCD

    CLEAR_BIT(PORTB, RSTPIN); // Reset LCD
    SET_BIT(PORTD, SCEPIN);   // Tell LCD we're not sending data.
    SET_BIT(PORTB, RSTPIN);   // Stop resetting LCD

    LCD_CMD(lcd_set_function, lcd_instr_extended);
    LCD_CMD(lcd_set_contrast, contrast);
    LCD_CMD(lcd_set_temp_coeff, 0);
    LCD_CMD(lcd_set_bias, 3);

    LCD_CMD(lcd_set_function, lcd_instr_basic);
    LCD_CMD(lcd_set_display_mode, lcd_display_normal);
    LCD_CMD(lcd_set_x_addr, 0);
    LCD_CMD(lcd_set_y_addr, 0);
}

uint8_t smiley_original[8] = {
    0b00111100,
    0b01000010,
    0b10100101,
    0b10000001,
    0b10100101,
    0b10011001,
    0b01000010,
    0b00111100,
};

uint8_t smiley_direct[8];
uint8_t x, y;

/*
**  Convert smiley into vertical slices for direct drawing.
**  This will need to be amended if smiley is larger than 8x8.
*/
void setup_smiley(void) {
    // Visit each column of output bitmap
    for (int i = 0; i < 8; i++) {

        // Visit each row of output bitmap
        for (int j = 0; j < 8; j++) {
            // Kind of like: smiley_direct[i][j] = smiley_original[j][7-i].
            // Flip about the major diagonal.
            uint8_t bit_val = BIT_VALUE(smiley_original[j], (7 - i));
            WRITE_BIT(smiley_direct[i], j, bit_val);
        }
    }

    // Choose any random (x,y)
    x = rand() % (LCD_X - 8);
    y = rand() % (LCD_Y - 8);
}

/*
**  Draw smiley face directly to LCD.
**  (Notice: we cheat on the y-coordinate.)
*/
void draw_smiley(void) {
    LCD_CMD(lcd_set_function, lcd_instr_basic | lcd_addr_horizontal);
    LCD_CMD(lcd_set_x_addr, x);
    LCD_CMD(lcd_set_y_addr, y / 8);

    for (int i = 0; i < 8; i++) {
        LCD_DATA(smiley_direct[i]);
    }
}

/*
**  Draw character directly to LCD.
**  Bypasses the screen buffer used by <lcd.h>.
```

```
**
**      Parameters:
**      x, y:   Integer coordinates at which the character will be placed.
**              The y-position will be truncated to the nearest multiple of 8.
**      ch:     The character to draw.
**      colour: The colour. If FG_COLOUR, the characters are rendered normally.
**              If BG_COLOUR, the text is drawn as a negative (inverse).
**  Returns:
**      No result is returned.
*/
void draw_char_direct(int x, int y, char ch, colour_t colour) {
    // Do nothing if character does not fit on LCD.
    if (x < 0 || x > LCD_X - CHAR_WIDTH || y < 0 || y > LCD_Y - CHAR_HEIGHT) {
        return;
    }

    // Move LCD cursor to starting spot.
    LCD_CMD(lcd_set_function, lcd_instr_basic | lcd_addr_horizontal);
    LCD_CMD(lcd_set_x_addr, (x & 0x7f));
    LCD_CMD(lcd_set_y_addr, (y & 0x7f) / 8);

    // Send pixel blocks.
    for (int i = 0; i < CHAR_WIDTH; i++) {
        uint8_t pixelBlock = pgm_read_byte(&(ASCII[ch - ' '][i]));

        if (colour == BG_COLOUR) {
            pixelBlock = ~pixelBlock;
        }

        LCD_DATA(pixelBlock);
    }
}

/*
**      Draw string directly to LCD.
**  Bypasses the screen buffer used by <lcd.h>.
**
**      Parameters:
**      x, y:   Integer coordinates at which the character will be placed.
**              The y-position will be truncated to the nearest multiple of 8.
**      str:    The text to draw.
**      colour: The colour. If FG_COLOUR, the characters are rendered normally.
**              If BG_COLOUR, the text is drawn as a negative (inverse).
**  Returns:
**      No result is returned.
*/
void draw_string_direct(int x, int y, char * str, colour_t colour) {
    for (int i = 0; str[i] != 0; i++, x += CHAR_WIDTH) {
        draw_char_direct(x, y, str[i], colour);
    }
}

/*
**      Remove smiley from LCD.
*/
void erase_smiley(void) {
    LCD_CMD(lcd_set_function, lcd_instr_basic | lcd_addr_horizontal);
    LCD_CMD(lcd_set_x_addr, x);
    LCD_CMD(lcd_set_y_addr, y / 8);

    for (int i = 0; i < 8; i++) {
        LCD_DATA(0);
    }
}

void setup(void) {
    set_clock_speed(CPU_8MHz);
    new_lcd_init(LCD_DEFAULT_CONTRAST);
    lcd_clear();
    draw_string_direct((LCD_X - 6 * CHAR_WIDTH) / 2, 4 * CHAR_HEIGHT, "Direct", FG_COLOUR);
    draw_string_direct((LCD_X - 6 * CHAR_WIDTH) / 2, 5 * CHAR_HEIGHT, "Writer", FG_COLOUR);
    setup_smiley();
    draw_smiley();
}

void process(void) {
    erase_smiley();
    x = (x + 1) % (LCD_X - 8);
    draw_smiley();
}

int main(void) {
    setup();

    for (;; ) {
        process();
        _delay_ms(100);
    }
}
```

The program also shows some of the extra actions that need to be taken when rendering straight onto the screen.

- The left-to-right, top-down declaration of a bitmap that works well in a text editor is not suitable for the LCD due to the pixel layout.
- We create a second image of smiley which conforms to the LCD topology by transposing smiley.
    - The rows of original smiley become columns in transposed smiley, and vice-versa.
- Because pixels are transferred in batches of 8, it is somewhat more difficult to draw smiley at pixel y-coordinates that aren't divisible by 8.
    - Today, we cheat by just truncating the vertical position to the nearest multiple of 8 vertical pixels.
- *Something to look forward to*: In AMS exercises you will be guided through the process of creating larger bitmaps, and drawing objects at arbitrary screen positions.

The End
