AN202

Low Pin-Count LCD Interface

1. Introduction

This application note provides an example interface for a C8051F330 device with an example LCD. First, this application note describes how an LCD works and then describes the two types of LCDs: direct drive and multiplexed drive. Next, the software interface and structure are explained. Finally, this note describes how to modify the software example to work with other LCDs.

The code accompanying this application note was originally written for C8051F33x devices. The code can also be ported to other devices in the Silicon Labs microcontroller range.

2. Key Points

- The software provided translates ASCII characters into 7-segment digits, compatible with the *printf()* standard library function.
- The LCD used in this example has 19 pins (4 Common and 15 Segment). Seven pins are used on the microcontroller: four for the Common pins and three that serve as a serial interface to a pair of 74HC595 8-bit, latched shift registers which are the segment drivers.
- The refresh rate of the LCD is chosen to minimize power consumption as well as minimize flickering.

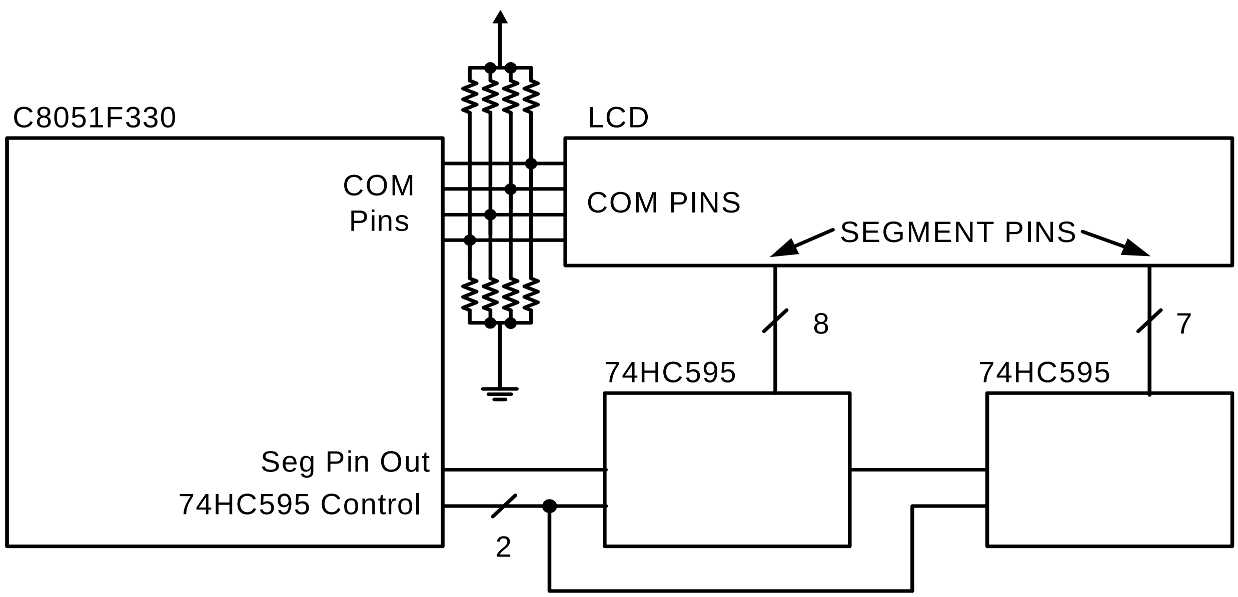


Figure 1. LCD Interface Block Diagram

3. LCD

The following sections describe how an LCD works.

3.1. Components of an LCD

An LCD consists of a collection of segments that are individually controlled. When there is no voltage across a segment, it is turned OFF and assumes the color of the background of the LCD. Applying an ac voltage across a segment causes it to turn ON and it will look darker than the background of the LCD. The root mean square (rms) value of the voltage across the segment must be greater than a certain threshold for the segment to turn on. This threshold is determined by the LCD manufacturer.

Figure 2 shows a diagram of a single segment. Each segment in an LCD has two terminals: a Backplane terminal and a Segment terminal. Electrically the segment looks like a capacitor. Multiple segments can connect to the same backplane. The collection of Segment Drivers is called the Frontplane.

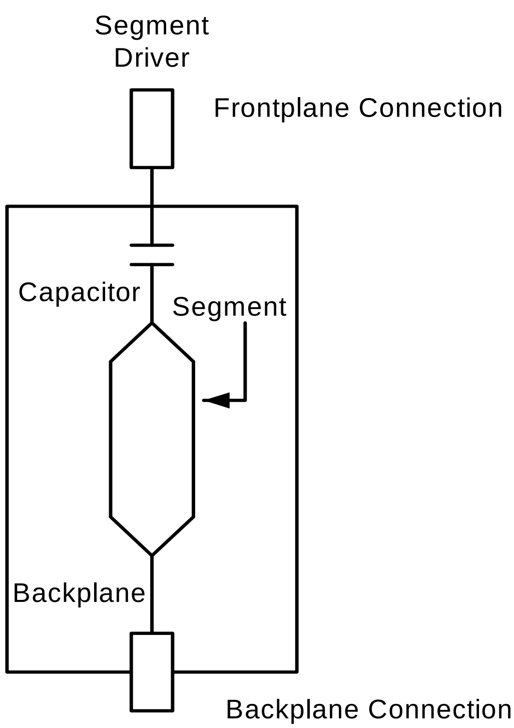


Figure 2. Single Segment in an LCD

3.2. Direct Drive LCD

In a direct drive LCD, each segment on the LCD is mapped to its own Segment pin. Another pin called the Common pin (COM) provides the voltage to the backplane. A direct drive LCD with N segments requires a total of N + 1 pins. Figure 3 below shows a diagram of an direct-drive LCD with seven segments.

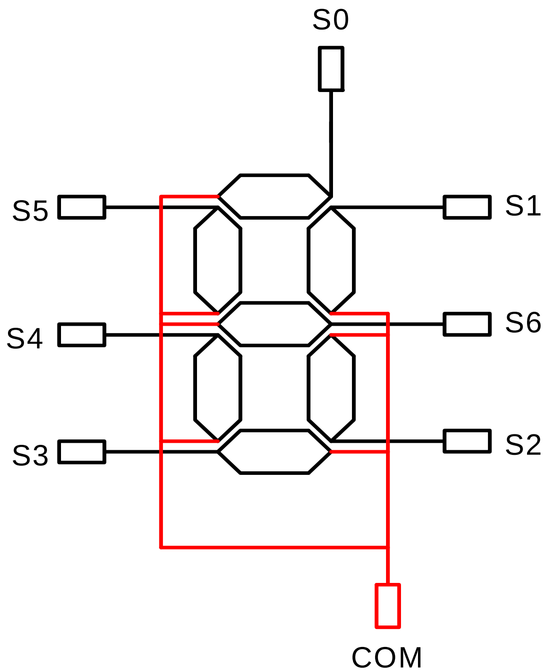


Figure 3. Connections for a 7-Digit Segment

Figure 4 shows a sample timing diagram that illustrates how a single segment is turned on and off. The microcontroller drives all the inputs of a direct drive LCD to either V_{DD} or GND. To turn a segment on, a voltage difference is applied between the backplane and the segment pin for that specific segment.

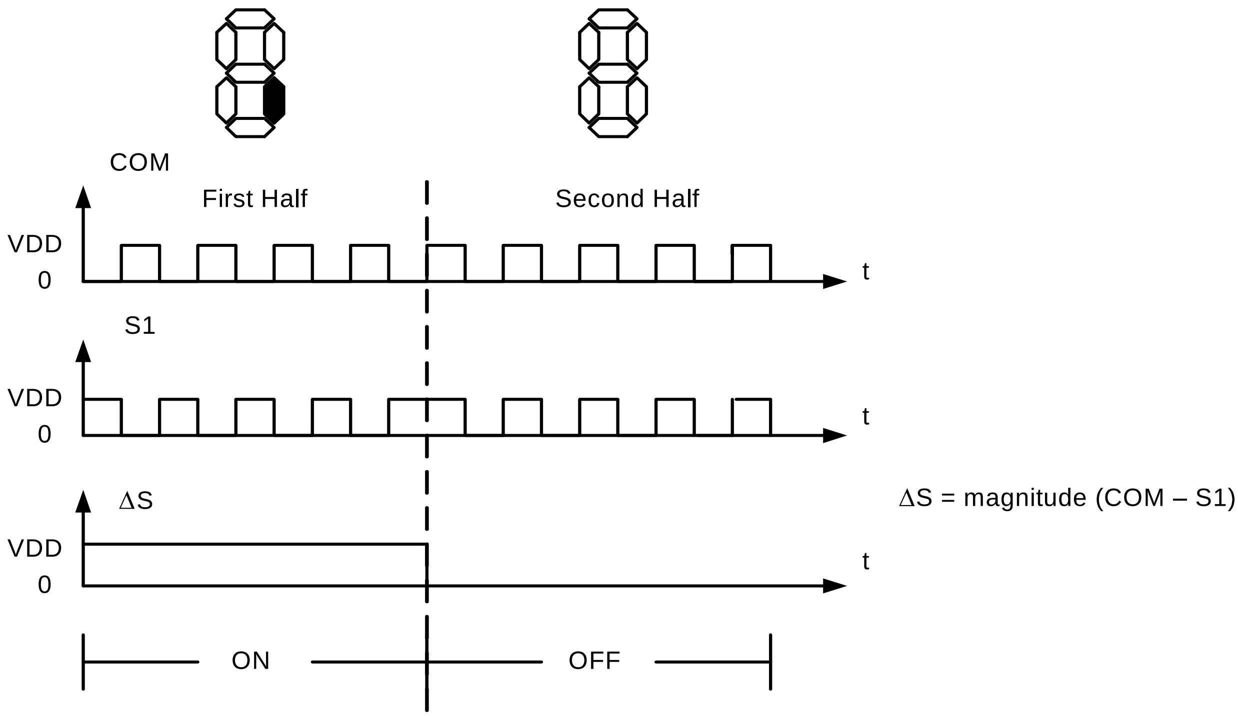


Figure 4. Sample Timing Diagram for a Single Segment

To achieve the best contrast and lowest power consumption, the COM pin should be driven with a 50% duty-cycle square wave.

In the first half of the timing diagram, the segment pin S1 is driven to a value opposite the value driven on the COM pin. This leads to a voltage difference across the segment, the magnitude of which is shown by ΔS, and the segment is turned ON for this length of time. In the second half of the timing diagram, the segment pin S1 is driven to the same value as the output on the COM pin. This leads to no potential difference across the segment and thus the segment is turned OFF for this period of time.

In summary, to turn a segment ON in a direct drive LCD, drive the corresponding segment pin to the value opposite the value of the COM pin. To turn a segment OFF, drive the segment pin to the same value as the COM pin.

Note: An ac excitation waveform is required to turn the segment ON, and the rms value of the voltage across the segment must be above a certain threshold for the segment to change color.

3.3. Multiplexed LCD

A multiplexed LCD has more than one backplane, and a corresponding COM pin for each of those backplanes. In a M-way multiplexed LCD, there are M separate backplanes and M COM pins.

The segments share Segment pins as well as COM pins. The segments are divided equally between the Segment pins, with each segment possessing a unique combination of Segment and COM pins. If an LCD has M COM pins and N segment pins, it can support up to M x N segments. For example, and LCD with 4 COM pins and 15 Segment pins can have up to 60 segments.

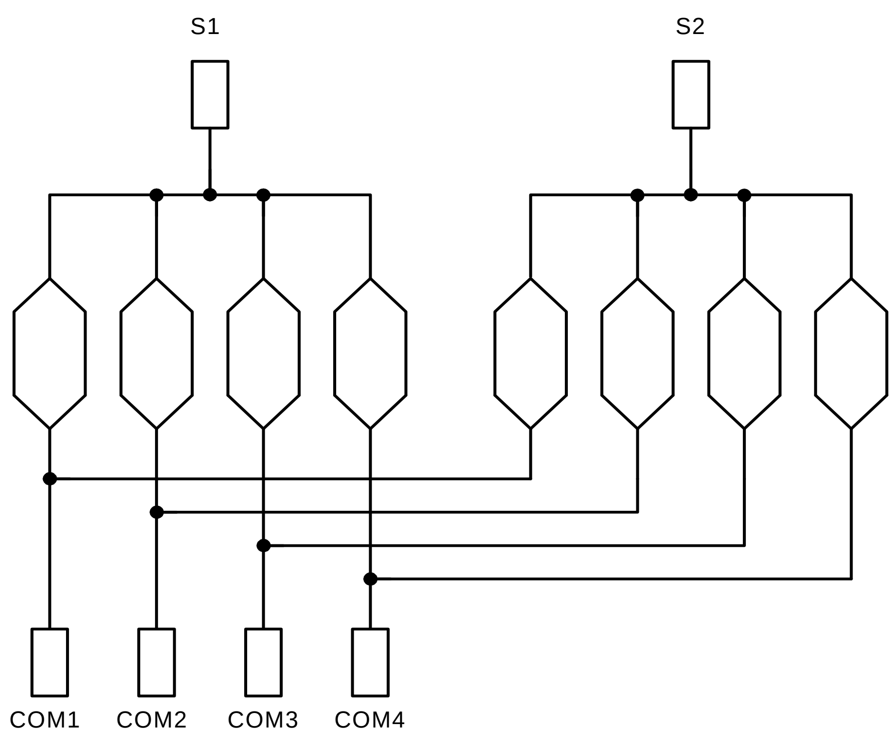


Figure 5. Connections for 8 Segments in a 4-way Multiplexed LCD

The LCD discussed in this application note can support up to 60 segments and is 4-way multiplexed. This means it has 15 groups of 4 pins each. Each group shares a single segment pin. This means that 19 pins are needed to interface with this LCD (15 for each segment group + 4 COM pins).

If the same LCD was designed as a direct drive LCD, it would require 61 pins (60 pins for each segment and 1 COM pin).

The diagram in Figure 5 shows the connections for a sample 4-way multiplexed LCD.

The COM pins in a multiplexed LCD are driven to one of three voltage levels: V_{DD}, $V_{DD}/2$, or GND. At any one moment, only one of the COM pins is driven to either V_{DD} or GND. All other COM pins are driven to $V_{DD}/2$. The segment pins are still only driven only to V_{DD} or to GND.

In a 4-way multiplexed LCD, each refresh cycle or period is separated into 8 phases. During the first four phases of the period, each COM pin is alternately driven to V_{DD}, while the other COM pins are held at $V_{DD}/2$. In the last 4 phases, each COM pin is alternately driven to GND, while the other COM pins are held at $V_{DD}/2$. The value of the COM pins during the last 4 phases is an inverse of the values in the first four phases. This is known as “1/4 duty cycle.”

Determining the segment pin value to turn a segment ON is similar to the method used for a direct drive LCD. Each segment is connected to one Segment pin and one of the COM pins. When the respective COM driver is high (in one of the first four phases of the period), the segment pin must be driven low to turn ON the segment and driven high to turn OFF the segment. The value of the segment pin is not relevant to a segment if its COM pin is not driven to V_{DD} or GND.

Driving the pin low when the respective COM pin is high creates a voltage difference across the segment whose rms value is greater than the threshold necessary to turn on the segment. Whenever a COM pin is set to $V_{DD}/2$, and the segment pin is set to V_{DD} or GND, the rms voltage is below the threshold to turn on the segment. As long as the rms value of the voltage across the segment over the four phases is above a certain threshold, the segment will remain ON for those four phases. If the rms value is below the threshold, the segment will remain OFF for those four phases.

The Segment pin value over the last four phases of the period must be the inverse of the value over the first four phases. This will cause the segments to remain in the same state (ON/OFF) that they were in the first four phases.

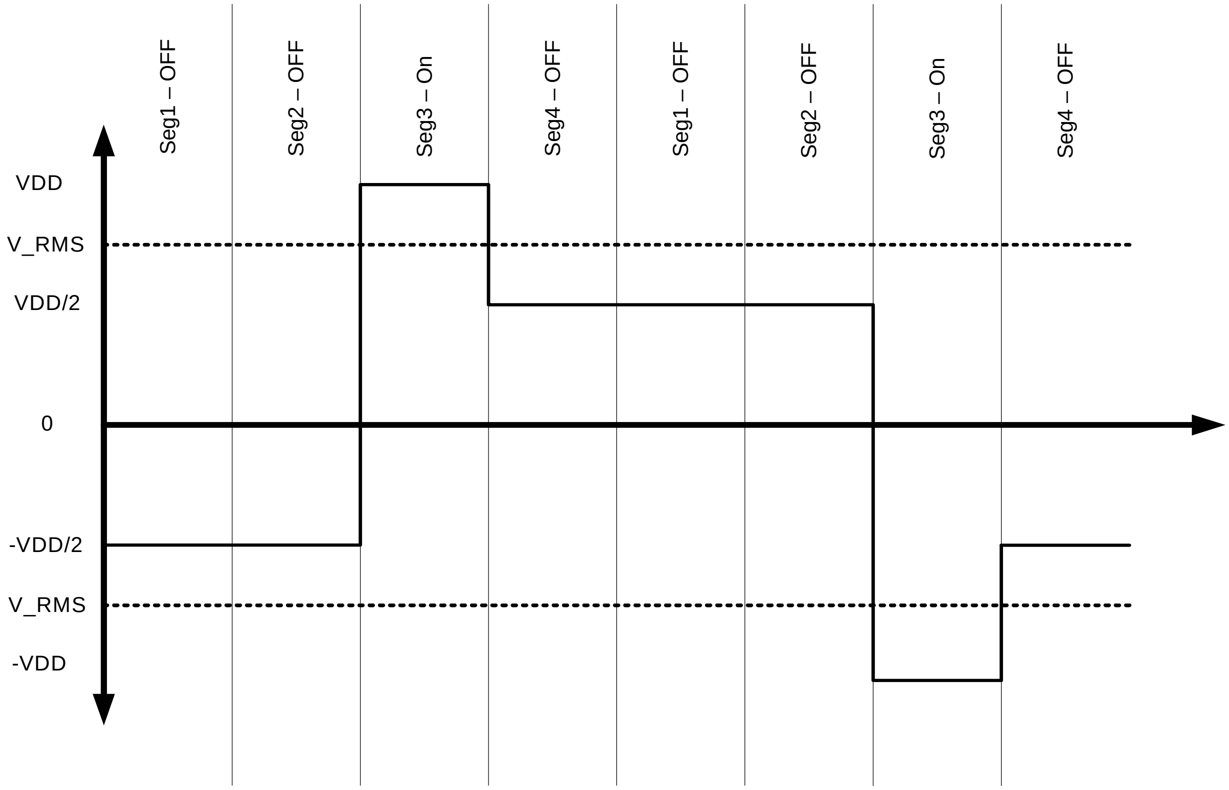


Figure 6. RMS thresholds for LCD segments

Figure 6 is a graph of the voltage difference over time between a specific Segment pin and four COM pins. Whenever the magnitude of the voltage difference is less than V_RMS, the segment that is between the Segment pin and the active COM pin is OFF. Whenever the magnitude of the voltage difference is greater than V_RMS, the segment that is between the Segment pin and the active COM pin is ON.

In Figure 6, the voltage difference is greater than V_RMS only during phase 3 (and the corresponding phase 7) and thus only segment 3 is ON. The other segments will remain off during this refresh cycle.

To create a rms value greater than the threshold for a certain segment, set the segment pin low whenever the corresponding COM pin is high, and set the segment pin high whenever the corresponding COM pin is low.

Figure 7 shows the waveforms for the COM signals, a single Segment pin, and the delta values for the four segments that the Segment pin is connected to. The delta values are the voltages across the segments. Figure 7 also shows which segments are ON and OFF during each period.

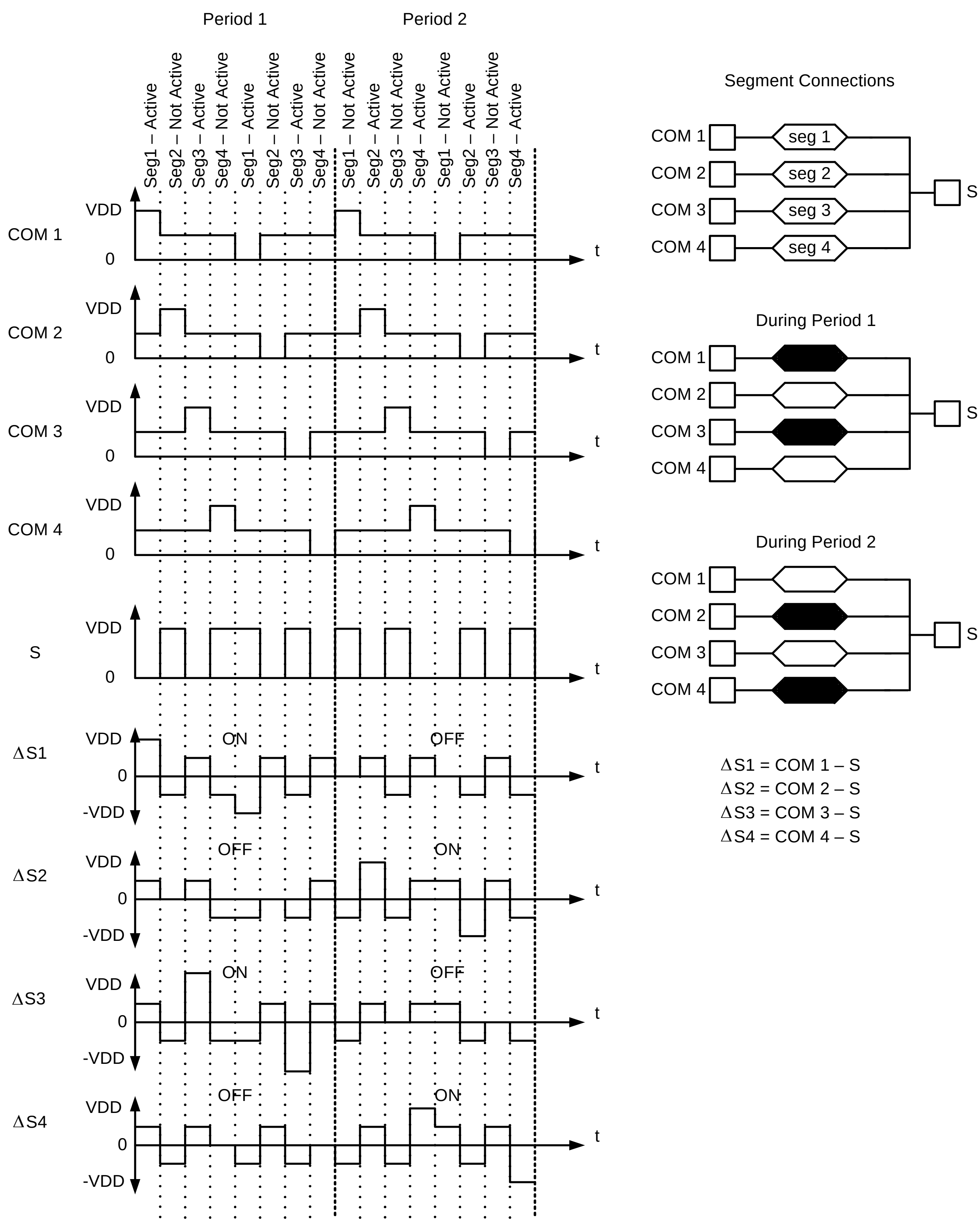


Figure 7. Timing Diagram for a Sample 4-Way Multiplexed LCD

4. How to Generate Segment Values for a Particular Digit

Figure 8 shows a single, 7-segment digit from a 4-way multiplexed LCD. Part A of Figure 8 shows the connections between the segment and the Segment pins S1 and S2. Part B of Figure 8 shows the connections between the segment and backplane COM pins COM1, COM2, COM3, and COM4.

This example shows how to generate the digit “5”. From Figure 8, segments 0, 2, 3, 5, and 6 need to be turned ON. Segments 1 and 4 need to be turned OFF. Figure 9 shows the timing diagram for one refresh cycle necessary to generate the digit “5”.

The ∆Sn show whether a segment is ON or OFF. The delta values for segments 0, 2, 3, 5, and 6 all have an rms value greater than $V_{DD}/2$. This means that the segments are on for as long as this S pattern is presented to the LCD.

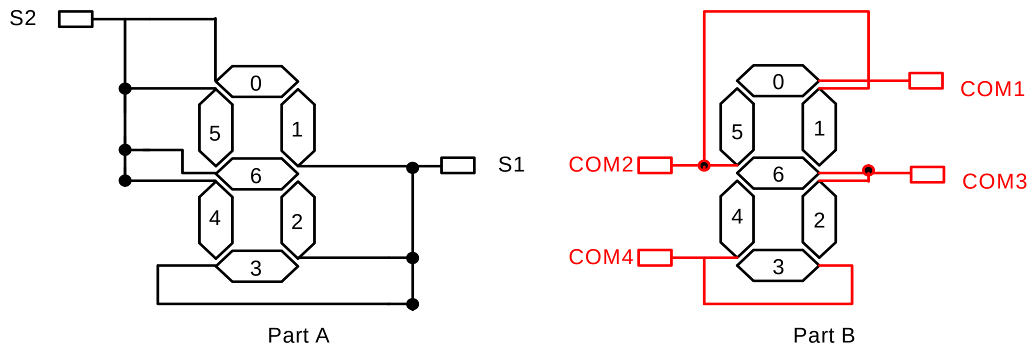
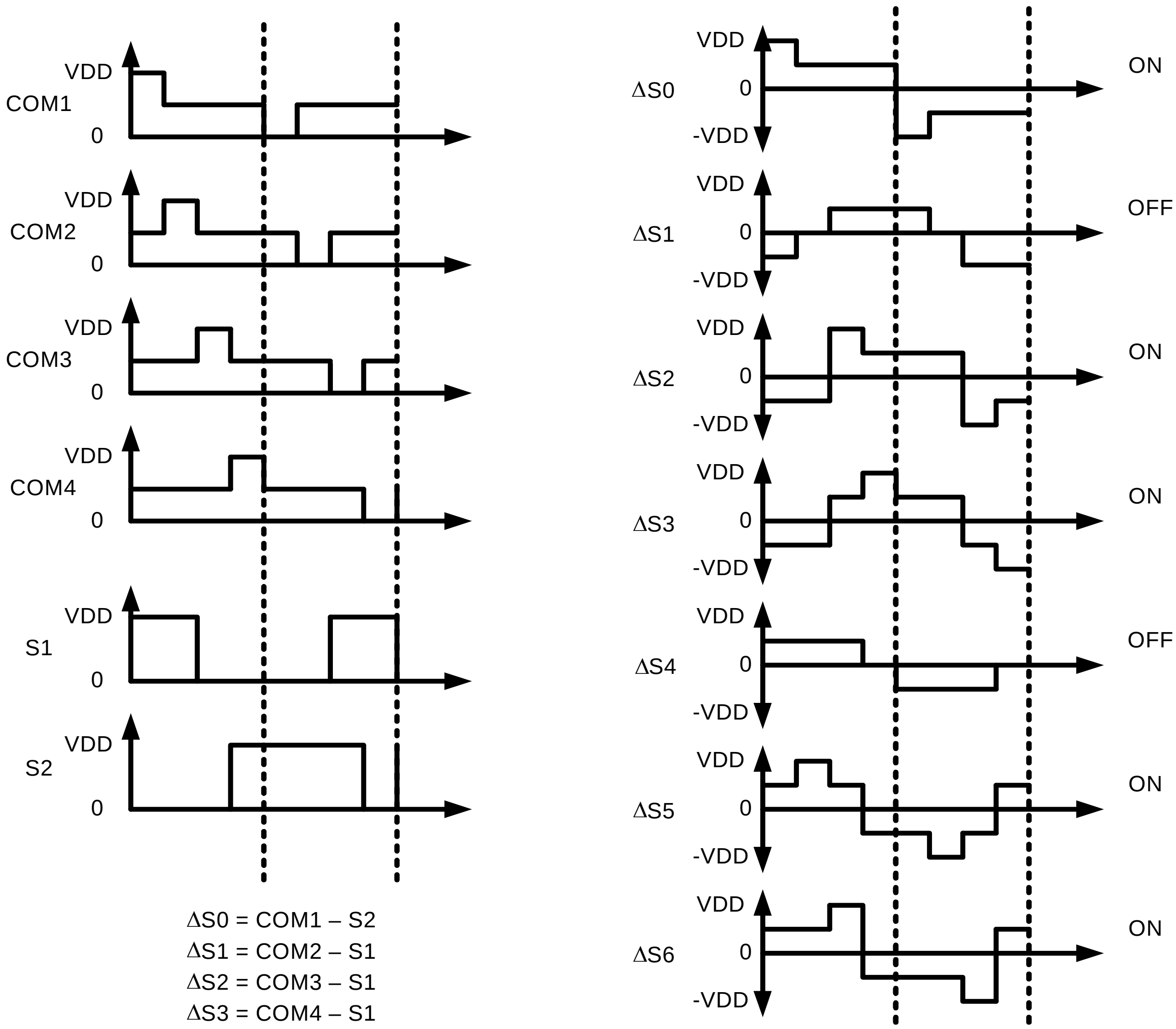


Figure 8. Sample Connections for a 7-Segment, 4-way Multiplexed LCD



ΔS0 = COM1 – S2
ΔS1 = COM2 – S1
ΔS2 = COM3 – S1
ΔS3 = COM4 – S1
ΔS4 = COM4 – S2
ΔS5 = COM2 – S2
ΔS6 = COM3 – S2

Figure 9. Timing Diagram Showing How to Display the Digit “5”

5. Software Example

This section describes how the user can interface to the LCD using the *putchar()* function.

5.1. Software Interface

The software provided in "10. Software Example Source Code" on page 16 provides an interface for a C8051F330 to the LCD. This LCD has six 7-segment digits. The four backplanes and 15 segment pins allow for 60 segments, but this LCD only makes 42 segments visible.

The example LCD library overloads the standard library *putchar()* function. The flow chart for the *putchar()* function is shown below in Figure 10.

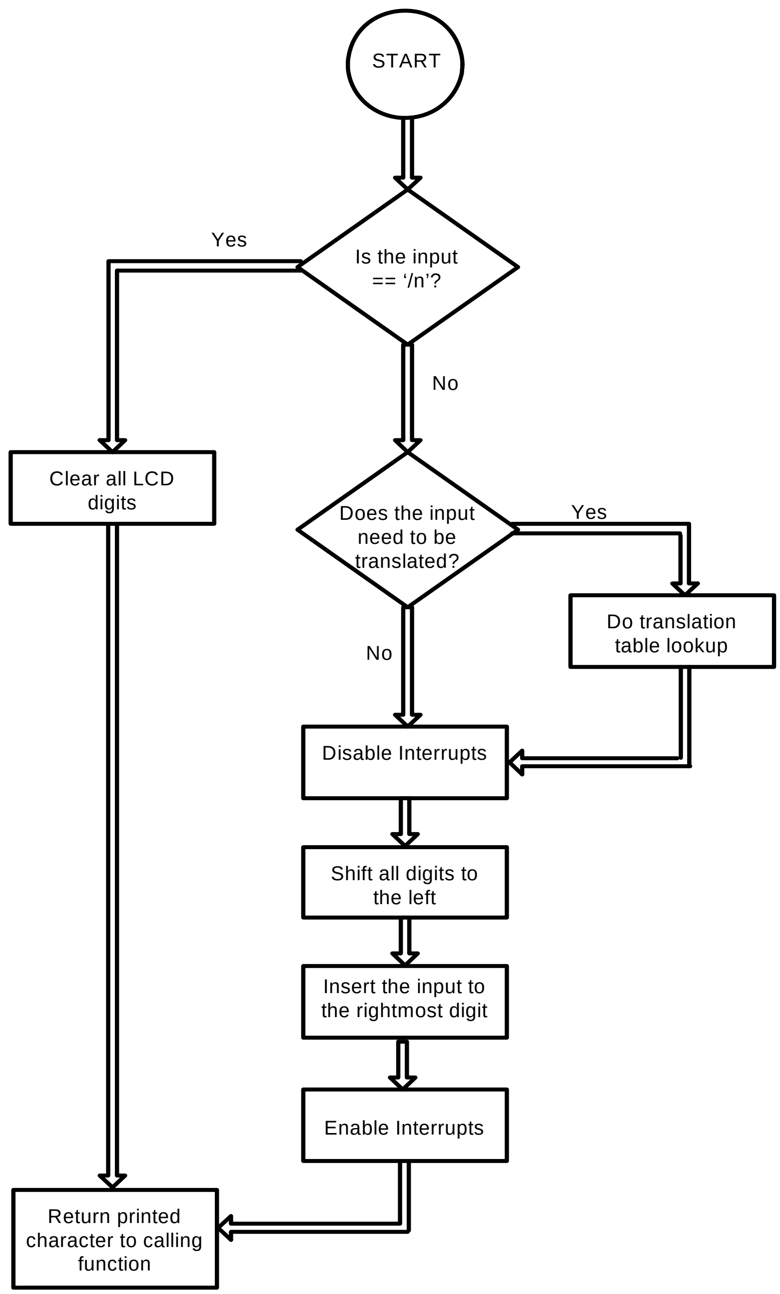


Figure 10. Flowchart for *putchar()*

The software overloads the standard *putchar()* function to print its output to the LCD. The function *putchar()* will insert the character in the rightmost digit on the LCD. Multiple calls to *putchar()* will shift the text to the left and insert the newest character in the rightmost digit's place. Interrupts are disabled when the display is updated to avoid flickering.

The most significant bit (MSB) of the 8-bit character passed to *putchar()* determines if *putchar()* uses a translation table or directly displays the value. If an 8-bit character whose ASCII value is between 0 and 127 (MSB is 0) is passed to *putchar()*, a translation will be made using the translation table. If a value from 128 to 255 (MSB is 1) is passed to *putchar()*, the digit will be directly displayed on the LCD.

If the bit is directly displayed, the 7 lower bits are translated as follows: if the bit is “0”, the corresponding segment is ON; if the bit is “1”, the corresponding segment is OFF. Figure 11 shows which segments are mapped to which bits. This allows the user to easily create every combination possible with seven segments. The function *putchar()* also clears the six LCD digits when it is passed the newline character, “\n”, whose ASCII value is 10. The translation table is described in detail in "7. LCD driver" on page 13.

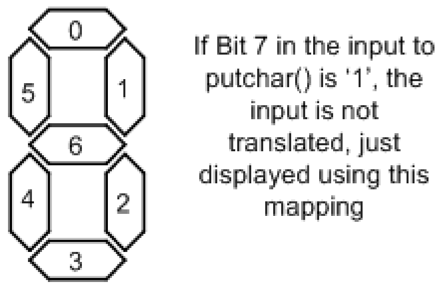


Figure 11. Bit Mapping Between Segments in a Digit and the *putchar()* Input Value

Table 1. Pin Map for the Example LCD

Pin#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
COM1					A1		A2		A3		A4		A5		A6				COM1
COM2					F1	B1	F2	B2	F2	B3	F4	B4	F5	B5	F6	B6		COM2	
COM3					G1	C1	G2	C2	G3	C3	G4	C4	G5	C5	G6	C6	COM3		
COM4	COM4				E1	D1	E2	D2	E3	D3	E4	D4	E5	D5	E6	D6			

6. Interpreting the LCD Data Sheet

The LCD data sheet provides the mapping between the segment pins and Segment pins and the COM pins. Table 1 shows the mapping for the example LCD.

Pins 1, 17, 18, and 19 are the COM pins. Pins 2–16 are the Segment pins. The intersection between the segment pin number and the COM pin is the segment connected between those pins. The blank spaces in columns 2-16 indicate that there is no segment between that Segment pin and COM pin. There are blank spaces in columns 1, 17, 18, and 19 because are the COM pins.

Table 1 indicates which bits should be shifted out to the shift register in each state of the *LCDrefresh_ISR()*. For example, when COM2 is active, the segment status bits for segments F1 through B6 (row 1) need to be shifted to the shift register. Any value can be shifted to 2, 3, and 4 on the LCD while COM2 is high because there is no segment that can be activated on those pins during the COM2 phase.

Table 2. Example LCD Specifications

Electrical / Optical Characteristics	Value	Units
Operating Temperature Range	–10°–60°	°C
Operational Voltage, RMS	2.5–3.5	V-RMS
Drive Frequency	60–300	Hz
Current Consumption	15	nA/mm^2
Turn On Time	80	ms
Turn Off Time	80	ms

The specification relevant to the firmware design is the drive frequency. The drive frequency determines how many times the *LCDrefresh_ISR()* should be triggered.

7. LCD driver

There are two main components to the LCD driver: the LCD refresh state machine and the ASCII translation table.

The LCD refresh state machine is located in *LCDrefresh_ISR()*. This function is responsible for driving the 4 COM signals and the 15 segment pins. This function is executed every time Timer2 overflows which is scheduled to happen 2000 times a second. This leads to a refresh rate of 250 Hz, which is in the ideal range for the LCD.

Each state in the 8-state state machine represents which COM driver is active at the time. Because there are eight phases in each period, this function is called eight times each refresh cycle. Each of the 42 segments has a bit in memory that holds its state. These bits are stored in the variable *LCD_digits*. During each run of the function, the ISR shifts the 15 bits that hold the state for the segments that are active this phase to the shift registers. These values are then passed in parallel to the LCD Segment pins. After four such phases, four more phases follow with all the output values inverted to match the associated polarity reversal of the COM signals. The flowchart for the ISR is shown in Figure 12.

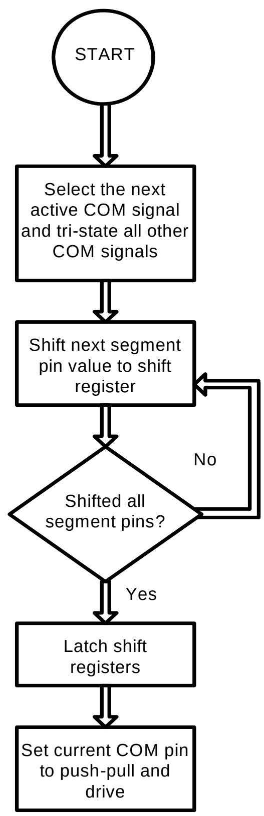


Figure 12. Flowchart for the *LCDrefresh_ISR*

The ASCII translation table is used whenever *putchar()* is called with an ASCII character as its parameter. The ASCII table is used to translate the ASCII character to a 7-segment digit. The table indicates which segments should be OFF and ON to best represent that character. If the character cannot be translated properly, the translation table displays a space, which is represented by 0xFF in the table. Figure 13 shows the digit mapping for each ASCII character. The first 32 characters generate spaces, so they are not part of the table. The top left number in each box is the value stored in the translation table. The bottom left number is the ASCII value. Both of these numbers are shown in hexadecimal notation. The bottom right character is the character being translated. If the box is empty, there is no translation available for that ASCII character and the LCD will display a space instead.

Figure 13. Mapping for the Basic Set of ASCII Characters (0–127)

8. Implementation Notes

To generate the V_{DD}/2 necessary for the backplane, a voltage divider is created using two equal sized resistors. Whenever the COM output pin on the microcontroller is set to “analog in” (high impedance), the voltage divider will provide the necessary V_{DD}/2 voltage to the LCD. Whenever the COM output pin is set to digital output, 1 (V_{DD}) or 0 (GND) will be sent to the LCD. See "11. Schematic" on page 27 for further details.

It is also important to note that increasing the refresh rate of the LCD to remove flickering also increases the power consumption. The refresh rate should be set to the minimum amount necessary to prevent flickering. This ideal refresh rate will vary for each manufacturer's LCD.

The shift registers (74HC595) are used to reduce the number of pins required on the microcontroller. It is important to choose shift registers that also provide a latching capability. In each phase, all the segment pin values should be shifted to the shift registers before latching those values to the LCD. This will prevent flickering on the LCD.

9. How to Customize the Software Example for a Different Multiplexing LCD

There are two parts of the code that need to change to accommodate different LCDs.

The Port I/O configuration has to change if the number of backplanes changes. For each backplane, a COM port pin must be allocated. If the number of Segment pins changes, more pins need not be allocated because the bits are shifted out serially. However, the number of shift registers depends directly on how many Segments pins are on the LCD.

The *LCDrefresh_ISR()* must be changed to accommodate the number of backplanes as well as the number of segment pins. This involves changing the number of states if the number of backplanes is different. The structures that store the segment state information must be modified to match the segments on the new LCD. The mapping between which segments are connected to which backplanes will determine which bits are sent to the LCD during each phase.

10. Software Example Source Code

This section contains the source code for the software example.

```
//-----------------------------------------------------------------------------
// LCDInterface.c
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories, Inc.
//
// AUTH: GP
// DATE: 19 NOV 04
//
// This program interfaces a C8051F330 device with an example LCD.
//
// Target: C8051F33x
// Tool Chain : Keil
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051F330.h>
#include <stdio.h>

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F3xx
//-----------------------------------------------------------------------------

sfr16 TMR2RL   = 0xca;                 // Timer2 reload value
sfr16 TMR2     = 0xcc;                 // Timer2 counter

sfr16 TMR3RL   = 0x92;                 // Timer3 reload value
sfr16 TMR3     = 0x94;                 // Timer3 counter

//-----------------------------------------------------------------------------
// Structures, Unions, Enumerations, and Type Definitions
//-----------------------------------------------------------------------------

// The translation table provides the mapping between ASCII characters
// and the segment pin values

// The first 32 characters (except 10) just produce a space;
// Character 10 (newline) clears the LCD digits
// Characters that can't be translated produce a space
// The MSB in the byte is meaningless because there are only 7 segments

// If the bit is low, the corresponding bar in the digit is active
// The 'diagram' below shows which bit corresponds to which bar in the LCD digit

//      0
//      _
//   5 | | 1        the middle inner bar is bit 6
//      -
//   4 | | 2
```

```
//      -
//      3


unsigned char code translation_table[128] = {

0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,     // 0 - 7
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,     // 8 - 15
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,     // 16 - 23
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,     // 24 - 31

0xFF, 0xCF, 0xDD, 0xFF, 0x92, 0x9B, 0xFF, 0xDF,     // 32 - 39
0xC6, 0xF0, 0xFF, 0xFF, 0xDF, 0xBF, 0xEF, 0xFF,     // 40 - 47
0x40, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x02, 0xF8,     // 48 - 55
0x00, 0x10, 0xF9, 0xF9, 0x9E, 0xB7, 0xBC, 0x18,     // 56 - 63

0xFF, 0x08, 0x00, 0xC6, 0x21, 0x06, 0x0E, 0x02,     // 64 - 71
0x09, 0xF9, 0xE1, 0x07, 0xC7, 0xFF, 0x48, 0x40,     // 72 - 79
0x0C, 0x18, 0xAF, 0x92, 0xF8, 0x41, 0xE3, 0xFF,     // 80 - 87
0x09, 0x11, 0xA4, 0x06, 0xFF, 0xF0, 0xDC, 0xF7,     // 88 - 95

0xDF, 0x20, 0x03, 0xA7, 0x21, 0x04, 0x0E, 0x10,     // 96 - 103
0x0B, 0xFB, 0x61, 0x07, 0xF9, 0xFF, 0xAB, 0x23,     // 104 - 111
0x0C, 0x18, 0xAF, 0x92, 0x07, 0xE3, 0xE3, 0xFF,     // 112 - 119
0x09, 0x11, 0xA4, 0xC6, 0xF9, 0xF0, 0xBF, 0xFF,     // 120 - 127
};

//-----------------------------------------------------------------------------
// Global Constants
//-----------------------------------------------------------------------------

#define SYSCLK       24500000          // SYSCLK frequency in Hz
#define TIMER2_RATE  1000              // Timer 2 overflow rate in Hz
#define TIMER3_RATE  2000              // Timer 3 overflow rate in Hz
#define PULSE_LENGTH 25

//-----------------------------------------------------------------------------
// Port names
//-----------------------------------------------------------------------------

sbit SRCLK = P1^1;                     // shift register clock
sbit RCLK  = P1^2;                     // shift register latch
sbit SER   = P1^3;                     // shift register serial in

sbit COM1  = P1^4;                     // COM1 pin on LCD
sbit COM2  = P1^5;                     // COM2 pin on LCD
sbit COM3  = P1^6;                     // COM3 pin on LCD
sbit COM4  = P1^7;                     // COM4 pin on LCD


//-----------------------------------------------------------------------------
// Global LCD Variables
//-----------------------------------------------------------------------------
```

```
unsigned char com_cycle = 1;           // start at COM 1
unsigned char com_invert = 0;          // start with positive cycle

// Below are the bit maps for each of the bars on the LCD;  If the bit is low
// then the bar is opaque (ON). If the bit is high, the bar is transparent (OFF).

// one char per digit on the LCD; initialized to OFF
unsigned char bdata LCD_digits[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

// The naming scheme: D1A means the A segment of digit 1

// Digit 1 (D1) is the leftmost digit on the LCD
sbit D1A = LCD_digits[0] ^ 0;
sbit D1B = LCD_digits[0] ^ 1;          // D1 is controlled by S1 and S2
sbit D1C = LCD_digits[0] ^ 2;
sbit D1D = LCD_digits[0] ^ 3;
sbit D1E = LCD_digits[0] ^ 4;
sbit D1F = LCD_digits[0] ^ 5;
sbit D1G = LCD_digits[0] ^ 6;

sbit D2A = LCD_digits[1] ^ 0;          // D2 is controlled by S3 and S4
sbit D2B = LCD_digits[1] ^ 1;
sbit D2C = LCD_digits[1] ^ 2;
sbit D2D = LCD_digits[1] ^ 3;
sbit D2E = LCD_digits[1] ^ 4;
sbit D2F = LCD_digits[1] ^ 5;
sbit D2G = LCD_digits[1] ^ 6;

sbit D3A = LCD_digits[2] ^ 0;          // D3 is controlled by S5 and S6
sbit D3B = LCD_digits[2] ^ 1;
sbit D3C = LCD_digits[2] ^ 2;
sbit D3D = LCD_digits[2] ^ 3;
sbit D3E = LCD_digits[2] ^ 4;
sbit D3F = LCD_digits[2] ^ 5;
sbit D3G = LCD_digits[2] ^ 6;

sbit D4A = LCD_digits[3] ^ 0;          // D4 is controlled by S7 and S8
sbit D4B = LCD_digits[3] ^ 1;
sbit D4C = LCD_digits[3] ^ 2;
sbit D4D = LCD_digits[3] ^ 3;
sbit D4E = LCD_digits[3] ^ 4;
sbit D4F = LCD_digits[3] ^ 5;
sbit D4G = LCD_digits[3] ^ 6;

sbit D5A = LCD_digits[4] ^ 0;          // D5 is controlled by S9 and S10
sbit D5B = LCD_digits[4] ^ 1;
sbit D5C = LCD_digits[4] ^ 2;
sbit D5D = LCD_digits[4] ^ 3;
sbit D5E = LCD_digits[4] ^ 4;
sbit D5F = LCD_digits[4] ^ 5;
sbit D5G = LCD_digits[4] ^ 6;

sbit D6A = LCD_digits[5] ^ 0;          // D6 is controlled by S11 and S12
sbit D6B = LCD_digits[5] ^ 1;
```

```
sbit D6C = LCD_digits[5] ^ 2;
sbit D6D = LCD_digits[5] ^ 3;
sbit D6E = LCD_digits[5] ^ 4;
sbit D6F = LCD_digits[5] ^ 5;
sbit D6G = LCD_digits[5] ^ 6;

//-----------------------------------------------------------------------------
// Function Prototypes
//-----------------------------------------------------------------------------

void SYSCLK_Init (void);
void Port_IO_Init();
void Timer2_Init (int);
void Timer3_Init (int);
char putchar(char);
void Strobe();
void wait_one_ms(unsigned int);

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main(void)
{
   PCA0MD &= ~0x40;                    // WDTE = 0 (clear watchdog timer enable)

   SYSCLK_Init();                      // initialize the oscillator
   Port_IO_Init();                     // initialize the ports
   Timer2_Init (SYSCLK / TIMER2_RATE); // enable timer to interrupt at some Hz
   Timer3_Init (SYSCLK / TIMER3_RATE); // enable timer to overflow at some Hz

   // We first configure the COM ports to analog inputs.  This allows us
   // to set them to high impedance if we write a 1 to the COM port.  Along with
   // some external resistors, we can then create a VDD/2 voltage.  When it is
   // time for the corresponding COM cycle, we can set the pin to push-pull and
   // drive the output to VDD or GND.  These 3 levels (VDD, VDD/2, GND) are
   // necessary only for the backplane (Common) pins on the LCD

   COM1  = 1;                          // high impedance
   COM2  = 1;                          // high impedance
   COM3  = 1;                          // high impedance
   COM4  = 1;                          // high impedance
   RCLK  = 0;                          // don't output anything to LCD
   SRCLK = 0;                          // don't shift anything to registers yet

   EA = 1;                             // enable global interrupts

   while (1)
   {
      printf ("Hello");

      wait_one_ms (1000);

      printf ("\n");
```

```
      wait_one_ms (1000);
   }
}


//-----------------------------------------------------------------------------
// Init Functions
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal 24.5MHz
// oscillator as its clock source.  Also enables missing clock detector reset.
//
void SYSCLK_Init (void)
{
   OSCICN |= 0x03;                     // Configure internal osc to max freq
   RSTSRC  = 0x04;                     // Enable missing clock detector
}


//-----------------------------------------------------------------------------
// Port_IO_init
//-----------------------------------------------------------------------------
//
// This routine initializes the ports and enables the crossbar
//
void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Skipped,     Push-Pull,  Digital    SRCLK for 74HC595
    // P1.2  -  Skipped,     Push-Pull,  Digital    RCLK for 74HC595
    // P1.3  -  Skipped,     Push-Pull,  Digital    SER for 74HC595
    // P1.4  -  Skipped,     Open-Drain, Digital    COM1 for LCD
    // P1.5  -  Skipped,     Open-Drain, Digital    COM2 for LCD
    // P1.6  -  Skipped,     Open-Drain, Digital    COM3 for LCD
    // P1.7  -  Skipped,     Open-Drain, Digital    COM4 for LCD

    P0MDOUT   = 0x80;
    P1MDOUT   = 0x0E;                  // configure above pins to Push-Pull
    P1MDIN    = 0x0F;                  // configure Pins 1.4 - 1.7 to analog in
    P1SKIP    = 0xFE;                  // skip pins 1.1 to 1.7
```

```
    XBR1      = 0x40;                    // enable crossbar

}

//-----------------------------------------------------------------------------
// Timer2_Init
//-----------------------------------------------------------------------------
//
// The timer overflows at a rate of TIMER2_RATE times a second
// The interrupt generated in handled by the LCD_refresh ISR
//
void Timer2_Init (int counts)
{
   TMR2CN = 0x00;                       // STOP Timer2; Clear TF2H and TF2L;
                                        // disable low-byte interrupt; disable
                                        // split mode; select internal timebase
   CKCON |= 0x10;                       // Timer2 uses SYSCLK as its timebase

   TMR2RL  = -counts;                   // Init reload values
   TMR2    = TMR2RL;                    // Init Timer2 with reload value
   ET2 = 1;                             // enable Timer2 interrupts
   TR2 = 1;                             // start Timer2
}

//------------------------------------------------------------------------------------
// Timer3_Init
//------------------------------------------------------------------------------------
//
// Configure the Timer to overflow without interrupts
// The overflow will be used in the wait function
//
void Timer3_Init (int count)
{
   TMR3CN = 0x00;                       // STOP Timer3; Clear TF3H and TF3L;
                                        // disable low-byte interrupt; disable
                                        // split mode; select internal timebase

   CKCON |= 0x40;                       // Timer3 uses SYSCLK as its timebase

   TMR3RL  =  -count;                   // Init reload values
   TMR3    =  TMR3RL;                   // Init Timer3 with reload value
   EIE1    &= 0x7F;                     // disable Timer3 interrupts

   TMR3CN  |= 0x01;                     // start Timer3
}

//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------

// LCDrefresh is triggered on a Timer2 Overflow

// Takes what is in the LCD bar bits and shift them into the two 74HC595
// shift registers depending on the COM cycle; The most signficant
```

```
// LCD pin (pin 16) gets shifted out first; Only 15 bits get shifted each
// COM cycle;

void LCDrefresh_ISR (void) interrupt 5
{
   int i = 0;

   if (com_cycle == 1)
   {
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = D6A  ^ com_invert; Strobe();
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = D5A  ^ com_invert; Strobe();
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = D4A  ^ com_invert; Strobe();
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = D3A  ^ com_invert; Strobe();
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = D2A  ^ com_invert; Strobe();
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = D1A  ^ com_invert; Strobe();
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = 1    ^ com_invert; Strobe();         // non-existent segment
      SER = 1    ^ com_invert; Strobe();         // non-existent segment

      RCLK = 1;                        // put shifted data to LCD - rising edge
      for (i=0; i<PULSE_LENGTH; i++);  // keep clock high for a while
      RCLK = 0;                        // turn off clock

      P1MDIN  &= ~0x80;                // configure COM4 to ANALOG_IN;
      P1MDIN  |= 0x10;                 // and COM1 to digital

      P1MDOUT &= ~0x80;                // make COM4 an open-drain
      P1MDOUT |= 0x10;                 // make COM1 a push-pull
      COM4    = 1;                     // set COM4 to high impedance
      COM1    = 1 ^ com_invert;        // start the COM1 cycle
      com_cycle = 2;                   // next state
   }

   else if (com_cycle == 2)
   {
      SER = D6B  ^ com_invert;     Strobe();
      SER = D6F  ^ com_invert;     Strobe();
      SER = D5B  ^ com_invert;     Strobe();
      SER = D5F  ^ com_invert;     Strobe();
      SER = D4B  ^ com_invert;     Strobe();
      SER = D4F  ^ com_invert;     Strobe();
      SER = D3B  ^ com_invert;     Strobe();
      SER = D3F  ^ com_invert;     Strobe();
      SER = D2B  ^ com_invert;     Strobe();
      SER = D2F  ^ com_invert;     Strobe();
      SER = D1B  ^ com_invert;     Strobe();
      SER = D1F  ^ com_invert;     Strobe();
      SER = 1    ^ com_invert;     Strobe();      // non-existent segment
```

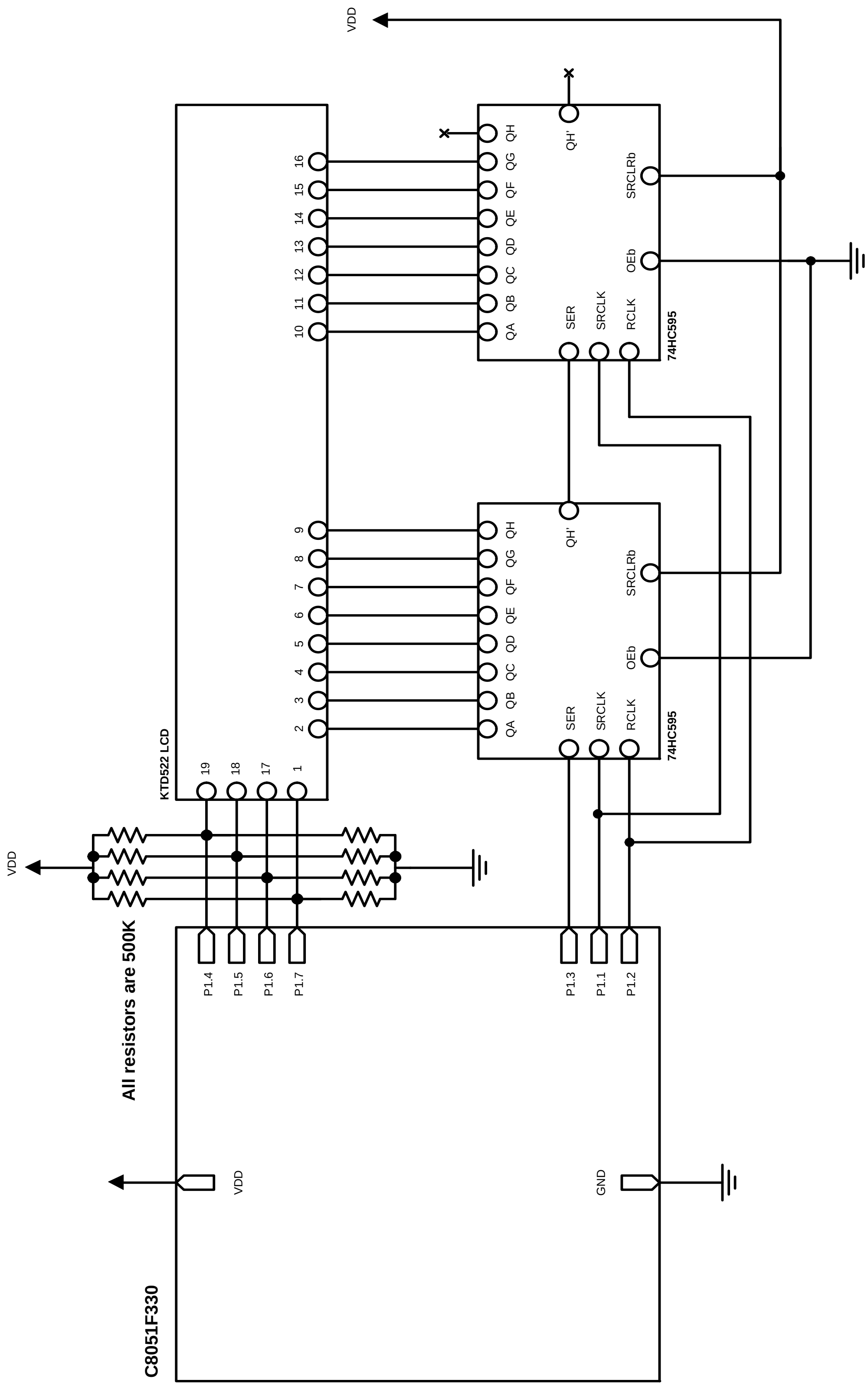
```
        SER = 1    ^ com_invert;     Strobe();      // non-existent segment
        SER = 1    ^ com_invert;     Strobe();      // non-existent segment

        RCLK = 1;                         // put shifted data to LCD - rising edge
        for (i=0; i<PULSE_LENGTH; i++);   // keep clock high for a while
        RCLK = 0;                         // turn off clock

        P1MDIN  &= ~0x10;                 // configure COM1 to ANALOG_IN;
        P1MDIN  |= 0x20;                  // and COM2 to digital

        P1MDOUT &= ~0x10;                 // make COM1 an open-drain
        P1MDOUT |= 0x20;                  // make COM2 a push-pull
        COM1    = 1;                      // set COM1 to high impedance
        COM2    = 1 ^ com_invert;         // start the COM2 cycle
        com_cycle = 3;                    // next state
    }

    else if (com_cycle == 3)
    {
        SER = D6C ^ com_invert;     Strobe();
        SER = D6G ^ com_invert;     Strobe();
        SER = D5C ^ com_invert;     Strobe();
        SER = D5G ^ com_invert;     Strobe();
        SER = D4C ^ com_invert;     Strobe();
        SER = D4G ^ com_invert;     Strobe();
        SER = D3C ^ com_invert;     Strobe();
        SER = D3G ^ com_invert;     Strobe();
        SER = D2C ^ com_invert;     Strobe();
        SER = D2G ^ com_invert;     Strobe();
        SER = D1C ^ com_invert;     Strobe();
        SER = D1G ^ com_invert;     Strobe();
        SER = 1   ^ com_invert;     Strobe();      // non-existent segment
        SER = 1   ^ com_invert;     Strobe();      // non-existent segment
        SER = 1   ^ com_invert;     Strobe();      // non-existent segment

        RCLK = 1;                         // put shifted data to LCD - rising edge
        for (i=0; i<PULSE_LENGTH; i++);   // keep clock high for a while
        RCLK = 0;                         // turn off clock

        P1MDIN  &= ~0x20;                 // configure COM2 to ANALOG_IN;
        P1MDIN  |= 0x40;                  // and COM3 to digital

        P1MDOUT &= ~0x20;                 // make COM2 an open-drain
        P1MDOUT |= 0x40;                  // make COM3 a push-pull
        COM2    = 1;                      // set COM2 to high impedance
        COM3    = 1 ^ com_invert;         // start the COM3 cycle
        com_cycle = 4;                    // next state
    }

    else if (com_cycle == 4)
    {
        SER = D6D ^ com_invert;     Strobe();
        SER = D6E ^ com_invert;     Strobe();
        SER = D5D ^ com_invert;     Strobe();
```

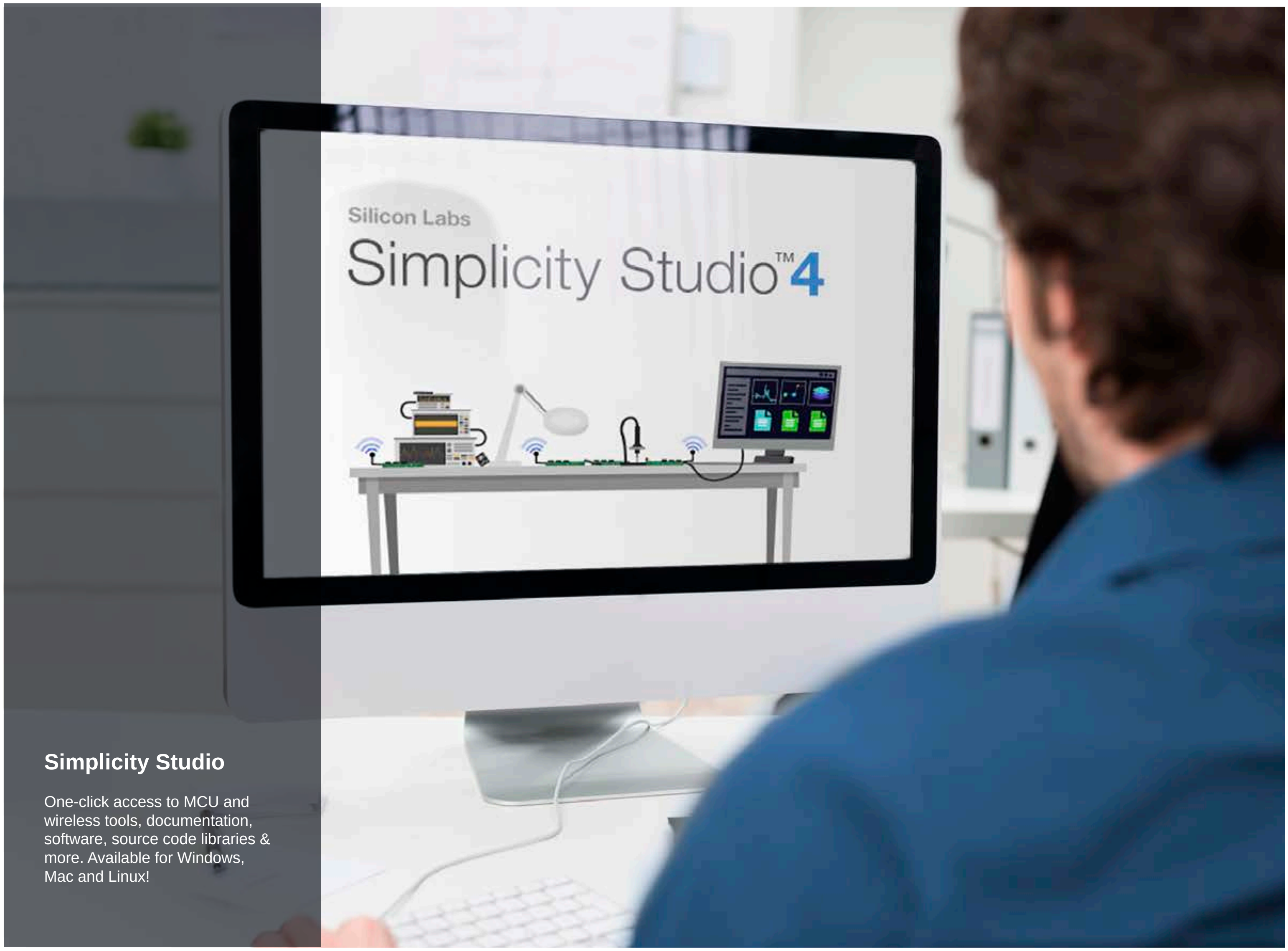
```
      SER = D5E ^ com_invert;     Strobe();
      SER = D4D ^ com_invert;     Strobe();
      SER = D4E ^ com_invert;     Strobe();
      SER = D3D ^ com_invert;     Strobe();
      SER = D3E ^ com_invert;     Strobe();
      SER = D2D ^ com_invert;     Strobe();
      SER = D2E ^ com_invert;     Strobe();
      SER = D1D ^ com_invert;     Strobe();
      SER = D1E ^ com_invert;     Strobe();
      SER = 1   ^ com_invert;     Strobe();         // non-existent segment
      SER = 1   ^ com_invert;     Strobe();         // non-existent segment
      SER = 1   ^ com_invert;     Strobe();         // non-existent segment

      RCLK = 1;                         // put shifted data to LCD - rising edge
      for (i=0; i<PULSE_LENGTH; i++);   // keep clock high for a while
      RCLK = 0;                         // turn off clock

      P1MDIN  &= ~0x40;                 // configure COM3 to ANALOG_IN;
      P1MDIN  |= 0x80;                  // and COM4 to digital

      P1MDOUT &= ~0x40;                 // make COM3 an open-drain
      P1MDOUT |= 0x80;                  // make COM4 a push-pull
      COM3    = 1;                      // set COM3 to high impedance
      COM4    = 1 ^ com_invert;         // start the COM4 cycle
      com_cycle = 1;                    // next state

      com_invert = com_invert ^ 1;      // toggle com_invert
   }

   TF2H = 0;                            // clear TF2

} // end LCDrefresh_ISR

//-----------------------------------------------------------------------------
// Strobe
//-----------------------------------------------------------------------------
//
// Strobe is used to clock the data into the 74HC595 shift registers
//
void Strobe()
{
   int i = 0;

   SRCLK = 1;
   for (i = 0; i < PULSE_LENGTH; i++);          // wait a few cycles
   SRCLK = 0;
   for (i = 0; i < PULSE_LENGTH; i++);          // wait a few cycles

}

//-----------------------------------------------------------------------------
// wait_one_msec
//-----------------------------------------------------------------------------
//
```

```
// Assumes Timer3 overflows once every 500 usec
//
void wait_one_ms(unsigned int count)
{
   count = count * 2;                  // overflows once every 500 usec
                                       // so double that is 1 ms

   TMR3CN &= ~0x80;                    // Clear Timer3 overflow flag
   TMR3 = TMR3RL;
   TMR3CN  = 0x04;                     // Start Timer3

   while (count--)
   {
      while (!(TMR3CN & 0x80)) {}      // wait for overflow
      TMR3CN &= ~0x80;                 // clear overflow indicator
   }

   TMR3CN &= ~0x04;                    // Stop Timer3
}

//-----------------------------------------------------------------------------
// LCD functions
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// putchar
//-----------------------------------------------------------------------------
//
// putchar only handles the digit components on the LCD screen.
// This functions shifts the digit values to the left, shifting out the
// left-most digit.  This function has 3 potential actions based on the input:
//
// 1. Any input whose ASCII code is between 0 and 127 gets translated
//    according to the translation table above
//
// 2. Any input whose ASCII code is between 128 and 255 is directly sent to
//    the LCD. The lower 7 bits indicate which of the seven segments are lit.
//
// 3. Passing a newline char '\n' to this function clears all 6 digits
//
// This function, unlike standard putchar, does not have any error return msgs.
//
// This function will not cause an interrupt to force output.  The input char
// will be displayed on the screen on the next refresh cycle

char putchar(char charIN)
{
   unsigned char iter = 0;

   if (charIN != '\n')                                  // not a new line
   {
      if ((charIN & 0x80) == 0) {                       // translation necesssary
        charIN = translation_table [charIN]; }          // quick lookup
```

```
      EA = 0;                                         // prevent partial display

      for (iter = 0; iter < 5; iter++) {              // shift the digits left
         LCD_digits[iter] = LCD_digits[iter+1]; }

      LCD_digits[5] = charIN;                         // new digit is rightmost

      EA = 1;                                         // enable interrupts again
   }

   else                                               // input is a newline
   {
      EA = 0;                                         // disable interrupts
      for (iter = 0; iter < 6; iter++) {
         LCD_digits[iter] = 0xFF; }                   // clear all digits
      EA = 1;                                         // enable interrupts
   }

   if (charIN == 0xFF) {                              // couldn't interpret OR space
      charIN = ' '; }                                 // return space

   return charIN;                                     // just like putchar
}
```

11. Schematic





IoT Portfolio	SW/HW	Quality	Support and Community
www.silabs.com/IoT	*www.silabs.com/simplicity*	*www.silabs.com/quality*	*community.silabs.com*

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com