Application Note

An LCD Drive for the Z8 Encore!® MCU

AN016203-0708

Abstract

Directly driving an LCD can be accomplished without a dedicated LCD driver on the microcontroller, and requires few additional resources. There are two types of LCDs—static and multiplexed. This Application Note discusses the programming of the multiplexed type of LCD in conjunction with the Zilog's Z8 Encore!® microcontroller.

Note: *The source code file associated with this application note,* `AN0162-SC01.zip` *is available for download at* www.zilog.com.

Z8 Encore! Flash Microcontrollers Overview

Zilog's Z8 Encore! products are based on the new eZ8™ CPU and introduce Flash Memory to Zilog's extensive line of 8-bit microcontrollers. Flash Memory in-circuit programming capability allows for faster development time and program changes in the field. The high-performance register-to-register based architecture of the eZ8 core maintains backward compatibility with Zilog's popular Z8® MCU.

The Z8 Encore! MCUs combine a 20 MHz core with Flash Memory, linear-register SRAM, and an extensive array of on-chip peripherals. These peripherals make the Z8 Encore! MCU suitable for various applications including motor control, security systems, home appliances, personal electronic devices, and sensors.

Discussion

A static LCD features a separate pin for each segment of the LCD and a common backplane pin. A requirement for illuminating a segment is to bias the segment in opposition to the backplane. An additional requirement is that LCDs cannot allow Direct Current (DC) present on a segment.

To prevent DC on a segment, the backplane is driven with a low-frequency square wave, and the segments are toggled with respect to the backplane.

Multiplex LCDs feature multiple backplanes, and a single segment pin is shared among multiple segments. To illuminate a particular segment, the segment pin is driven in opposition to the backplane, and the unused backplanes remain in an IDLE state. The backplanes are again driven with a low-frequency square wave to prevent DC bias on the segments.

The Challenge

Programming a multiplexed LCD can be a difficult task due to the multiplexed arrangement of the segments. Multiplexed LEDs usually feature a separate backplane for each LED digit, However, multiplexed LCDs arrange their backplanes across the top, middle, and bottom of the digit. This arrangement can make the decoding process very complicated, but it is important to mention that even microcontrollers that feature dedicated LCD drivers still require a difficult decoding process (see Figure 1 on page 2).

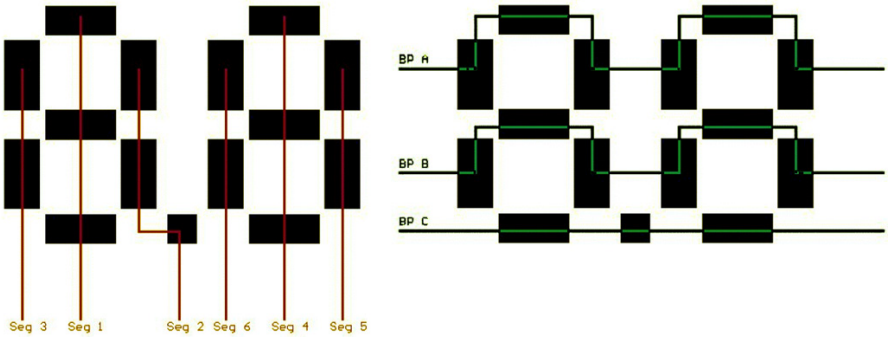

Figure 1. Segment Arrangement

The next engineering task is related to the voltage required to illuminate a segment. The ON drive level of a multiplexed segment is reduced because the segment spends most of its time in an IDLE state and is only asserted 25% of the time. In effect, this statement means that at lower operating voltages, a segment may not illuminate.

A further complication is that the segment can perceive a voltage potential while in its OFF state as the shared segment pin is being asserted by the currently active backplane. These contrasting problems may become worse as more backplanes are added to the display, because with each additional backplane, the available ON voltage is reduced, and the residual OFF voltage is increased.

Figure 2 displays how contrast decreases with each backplane due to the fact that there is less difference between an ON segment and an OFF segment. In Figure 2, a static display with one backplane receives 100% of its available V_{CC} for an ON voltage and 0% for an OFF voltage. In the 3-plane example, one-half of the V_{CC} is available for ON voltage and an OFF segment receives one-quarter of the V_{CC}. LCDs vary from one manufacturer to the next, but the typical threshold voltage is 2.3 V RMS.

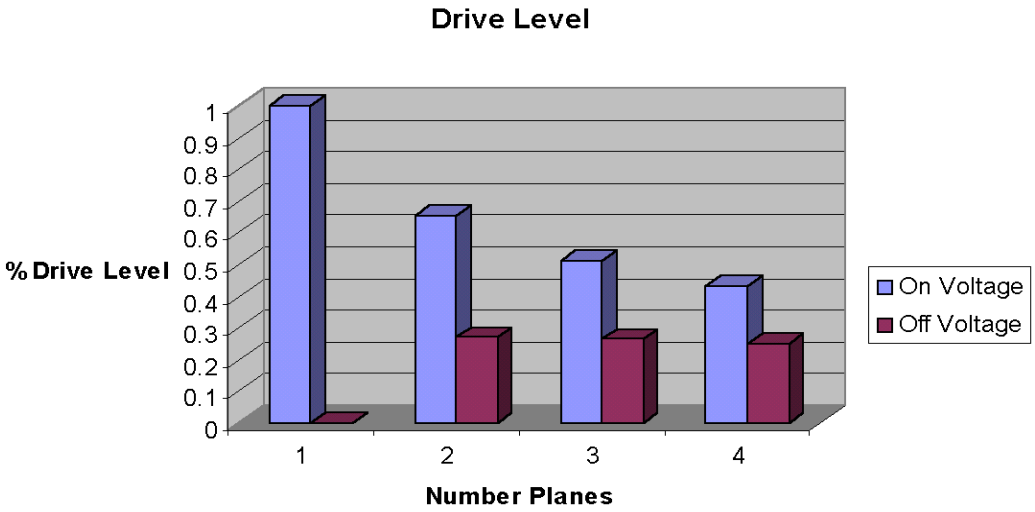

Figure 2. Drive Level

With only one-half of the V_{CC} available for ON voltage, it is easy to see that a 3.3 V microcontroller is unable to directly drive a multiplex LCD. The purpose of this Application Note is to show that you can drive a multiplexed LCD on the 3 V Z8 Encore!® MCU.

Hardware Architecture

To drive a multiplexed LCD with a 3 V MCU, the drive level must be boosted. To reduce gate count and complexity, only the backplanes are boosted. Segment drive voltages swing above and below $^1/_2V_{CC}$; therefore, a boosted backplane signal must perform in the same manner by using the Z8 Encore! MCU's port pins as two charge pumps referenced at $^1/_2V_{CC}$. IC1, the 4050 buffer, is used to provide the level-shifting function. For more details, see the schematic in Appendix A—Schematic Diagram on page 9.

Each backplane is driven High and idled while the other planes are driven. The process is inverted to remove any DC component, as shown in Figure 3.

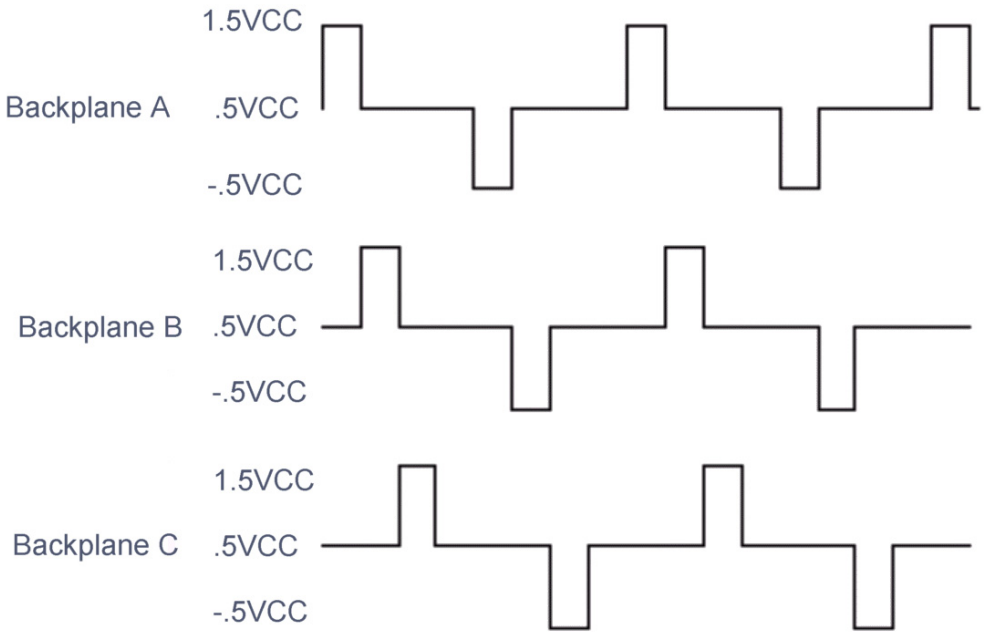

Figure 3. Backplane Waveforms

Segments are turned ON by driving the segment pin in the opposite direction of the active backplane, and OFF by driving the pin in the direction of the active backplane. During the backplane's IDLE state, the voltage on any segment is below the threshold voltage. As a result, the voltage on such a segment remains unlit.

Software Implementation

Due to the additional speed and memory of the Z8 Encore! family, it is assumed that development occurs in the 'C' programming language. Application written in C can be easily ported to different environments, if the software is written to allow easy porting.

With this aspect, macros are used for the I/O-specific operations so that the bulk of the software will remain untouched if the code is ported to another device.

The code segment provided below maintains the charge pumps.

```
/*Charge pump definitions
The charge pumps boost the segment drive voltage and are serviced each timer
interrupt. The Positive pump is pulled Low to charge and floated to an input
state. The negative pump is floated when charging. The cap is referenced at 1/
2VCC and the charge on the capacitor appears to be VCC +/- the reference. The
macro also initializes the port mode for the port so it is always refreshed.
*/

#defineChargePumpsPDADDR=PxHD; PDCTL|= B3|B4; PDOD&=~B3;
PDOD|=B4; PDADDR=PxOC;
PDCTL&=~(B3|B4); PDADDR=PxDDR;
PDCTL&=~(B3|B4)
#defineFloatPumpsPDADDR=PxDDR; PDCTL|=B4|B3
```

The macros listed below manage the backplane drive. These macros are complex as only one pin is required per backplane, but two pin states are required for each backplane.

```
/*Backplane drives require three states: an ON, an OFF, and an IDLE. By mixing
BP1 with BP2, BP2 with BP3, and BP3 with BP1 it's possible to get all three
states on each plane without requiring additional pins.

PlaneX123

_______________
BP11101
BP20110
BP31011
`BP10010
`BP21001
`BP30100

*/

#define SetUpBackplanePDADDR=PxOC; PDCTL&=~(B0|B5|B6);
PDADDR=PxDDR; PDCTL&=~(B0|B5|B6)
#define BP1PDOD&=~B6; PDOD|=B0|B5
#define BP2PDOD&=~B5; PDOD|=B0|B6
#define BP3PDOD&=~B0; PDOD|=B6|B5

#define NotBP1PDOD&=~(B0|B5); PDOD|=B6
#define NotBP2PDOD&=~(B0|B6); PDOD|=B5
#define NotBP3PDOD&=~(B5|B6); PDOD|=B0
Finally, the macros for driving the segments.
```

```
/*This next macro takes the individual segments stored in the display buffer
and places them on the ports. It could have been done without the macro but
this makes it more generic. There will be six planes with two buffers because
there are more than 8 segments*/

#define DisplaySegmentsPAOD&=~0xF8;
PAOD|=(buffer[plane]&0x00F8); PCOD=0;
PCOD|=((buffer[plane]&0x7F00)>>8)
```

As mentioned earlier, the difficult programming task involved in multiplex LCDs is a rather unusual multiplexing scheme. The above macro simply places the previously-decoded segments from the buffer and on to the ports. As the decoding process is so involved, it is not performed in the Interrupt Service Routine (ISR). ISRs must be kept as short as possible; all that is required in the ISR, set the backplanes and drive the segments. The buffer is an integer array that holds the 12 segments used in our display. The single dimension in this array is the plane. There are six planes in this dimension—one for each backplane state: A, B, C, A', B', and C'. When the decoding process is complete, the buffer must be loaded with the 12 segments of data across the 6 planes.

The first step in decoding is to define how the characters are displayed. This definition is universal for all 7 segment displays. Therefore, the below code segment can be reused.

```
#defineDig_0Seg_a | Seg_b | Seg_c | Seg_d | Seg_e | Seg_f
#defineDig_1Seg_b | Seg_c
#defineDig_2Seg_a | Seg_b | Seg_g | Seg_e | Seg_d
#defineDig_3Seg_a | Seg_b | Seg_g | Seg_c | Seg_d
#defineDig_4Seg_f | Seg_g | Seg_b | Seg_c
#defineDig_5Seg_a | Seg_f | Seg_g | Seg_c | Seg_d
#defineDig_6Seg_a | Seg_f | Seg_g | Seg_c | Seg_d | Seg_e
#defineDig_7Seg_a | Dig_1
#defineDig_8Seg_g | Dig_0
#defineDig_9Seg_a | Seg_f | Seg_g | Seg_b | Seg_c
#defineDig_ASeg_a | Seg_b | Seg_c | Seg_g | Seg_e | Seg_f
#defineDig_bSeg_f | Seg_e | Seg_g | Seg_d | Seg_c
#defineDig_CSeg_a | Seg_f | Seg_e | Seg_d
#defineDig_dSeg_b | Seg_c | Seg_d | Seg_e | Seg_g
#defineDig_ESeg_a | Seg_f | Seg_e | Seg_d | Seg_g
#defineDig_FSeg_a | Seg_f | Seg_e | Seg_g
#defineDig_gSeg_a | Seg_f | Seg_g | Seg_b | Seg_c | Seg_d
#defineDig_hSeg_g | Seg_c | Seg_e | Seg_f
#defineDig_IDig_1
#defineDig_JDig_1 | Seg_d
#defineDig_LSeg_d | Seg_e | Seg_f
#defineDig_nSeg_c | Seg_e | Seg_g
#defineDig_ODig_0
#defineDig_PSeg_g | Seg_a | Seg_b | Seg_e | Seg_f
#defineDig_rSeg_g | Seg_e | Seg_f
#defineDig_SSeg_g | Seg_a | Seg_c | Seg_d | Seg_f
#defineDig_tSeg_g | Seg_e | Seg_f | Seg_d
#defineDig_USeg_b | Seg_c | Seg_d | Seg_e | Seg_f
```

The segments are scattered across three separate backplanes, and must be arranged in a single byte as three packets of three bits. The last bit is unused. This arrangement mirrors the physical layout of a display digit (see Table 1).

Table 1. Segment Assignment

Backplane	A	B	C
Segment 1	a	g	d
Segment 2	b	c	decimal
Segment 3	f	e	Not Used

```
#define Seg_a1
#define Seg_g2
#define Seg_d4

#define Seg_b8
#define Seg_c16
#define Seg_dp32

#define Seg_f64
#define Seg_e128
```

Our goal is to place the segment data into three integers—one for each display backplane. The software must split the single byte representing the 7-segment character into the three segments by three planes. Table 2 indicates how the integer array stores the individual segment bits.

Table 2. Segment Array

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PlaneA					4f	4b	4a	3f	3b	3a	2f	2b	2a	1f	1b	1a
PlaneB					4e	4c	4g	3e	3c	3g	2e	2c	2g	1e	1c	1g
PlaneC					NC	4dp	4d	NC	3dp	3d	NC	2dp	2d	NC	1dp	1d

As each digit of the LCD requires three segments, addressing the correct segment pin requires shifting the data over three bits for each digit. Finally, the correct physical pin of the microcontroller must be addressed. The assignments in the code segment below can change based upon the board layout and other resources.

```
#define Seg_1AGD8//PAOD|=B3
#define Seg_1BCDP16//PAOD|=B4
#define Seg_1FE32//PAOD|=B5
#define Seg_2AGD64//PAOD|=B6
#define Seg_2BCDP128//PAOD|=B7
```

```
#define Seg_2FE256//PCOD|=B0

#define Seg_3AGD512//PCOD|=B1
#define Seg_3BCDP1024//PCOD|=B2
#define Seg_3FE2048//PCOD|=B3

#define Seg_4AGD4096//PCOD|=B4
#define Seg_4BCDP8192//PCOD|=B5
#define Seg_4FE16384//PCOD|=B6
```

The code segment below determines which segments turn ON for a particular character.

```
for (digit=0, shift=9; digit<4; digit++, shift-=3)
{
    segments[0]|=(0x07 & CharTbl[que[digit]])<<shift;
    segments[1]|=((0x38 & CharTbl[que[digit]])>>3)<<shift;
    segments[2]|=((0x1C0 & CharTbl[que[digit]])>>6)<<shift;
}
```

Storing the correct segment to turn ON requires testing each individual bit, but the advantage is that the code becomes very portable, as shown below:

```
for(plane=0;plane<3;plane++)
{
if (segments[plane]&B0)
    buffer[plane]|=Seg_1AGD;

if (segments[plane]&B1)
    buffer[plane]|=Seg_1BCDP;

if (segments[plane]&B2)
    buffer[plane]|=Seg_1FE;

if (segments[plane]&B3)
    buffer[plane]|=Seg_2AGD;

etc.
}

buffer[0]=tempbuffer[0]; // We have an assignment here rather than
buffer[1]=tempbuffer[1]; // setting a single variable in the
                          // statements above because a timer IRQ
buffer[2]=tempbuffer[2]; // occurs while the segments are being
buffer[3]=~buffer[0];    // gathered and that will cause the display
buffer[4]=~buffer[1];    // to flicker. These are simply complements
buffer[5]=~buffer[2];    // of the first three.
```

Summary

The actual code required for directly driving an LCD is not very complex. The time required to decode the individual segment planes in this C example is only 141 µs. The advantage is the ability to drive very large displays directly without an additional LCD driver, or the use of a microcontroller with a dedicated driver. The only disadvantage is the additional pins required for the charge pumps and backplane drive, but in most cases, the additional pins are cheaper than a dedicated driver.

Appendix A—Schematic Diagram

Figure 4 displays a schematic for an LCD drive using the Z8 Encore!® MCU.

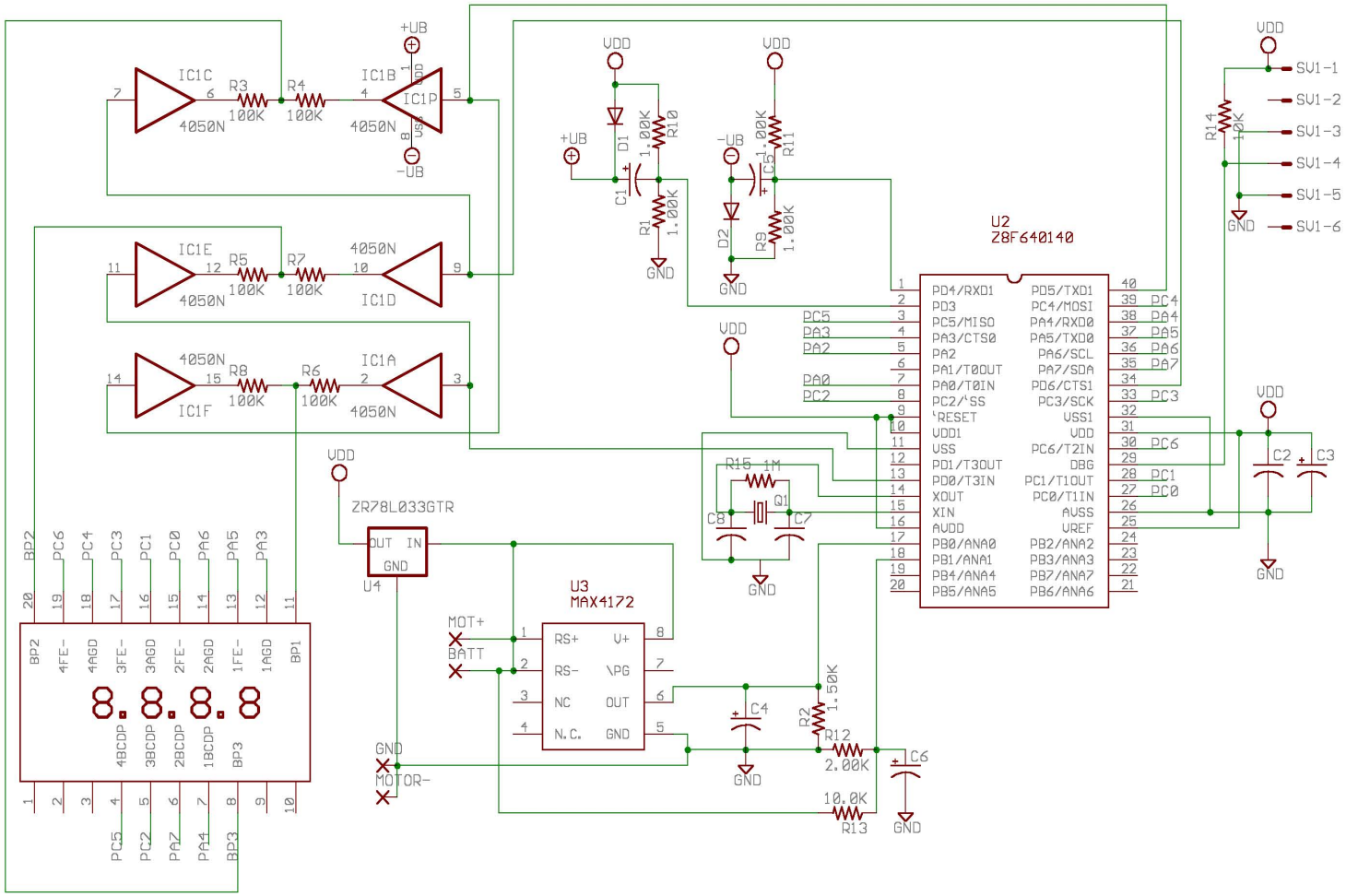

Figure 4. Schematic Diagram

Warning: DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.